Tartalomjegyzék

1. Bevezetés

A nagy adat (big data) napjainkban az egyik vezető cím az informatikai terminológiák körében, különböző címekkel ellátva, mint: *Adat, adat mindenhol* (Economist, 2016) vagy *Big data: Információ kinyerése adatokból* (Felice és R. 2008). Olyannyira megnőtt a kereslet a nagy adattal foglalkozó szakemberek iránt, hogy már az informatikán kívül is megjelent, Hal Varian –aki egyben Google vezető közgazdásza– szerint a *következő 10 éven belül az egyik legvonzóbb szakma lesz az adatokkal foglalkozó állás* (Davenport és Patil, 2016). De mit is jelent pontosan? Nincs explicit meghatározás a fogalomra, de Doug Laney 2001-es definíciója egy jó kiindulópontnak tekinthető: az adatok nagy mennyiségben (volume), gyorsan (velocity) és különböző formátumban (variety) jelennek meg (3V's) (Laney, 2015). Azonban, ma már kiegészíthetjük ezt a fogalmat még 4V-vel: bizonyosság (veracity, érték (value), adategyezés (variability) és megjelenítés (visualization).(Rijmenam, 2015) 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 (Emc.com, 2015) szerint az adatok mennyisége kétévente duplázódik, így 2020-ra a *forgalomban* lévő adatmennyiség elérheti a 44 zetabájtnyi (vagy 44 trillió gigabyte-nyi) 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 megnövelheti azáltal, hogy rejtett összefüggéseket mutat meg. (William, 2015)

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 jelentenek az informatikai iparnak, mivel érintik az fizikai manifesztációt (hardver), mind az ezt vezérlő és feldolgozó réteget (szoftver és algoritmus). A rendellenességek,–amelyek a tradicionális 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 (Sensmeier, 2015), „dolgok" internete (IoT) (Rijmenam, 2015) vagy alkalmazás monitoring (Google Analytics). 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, (Emmerich 2015) amelyek felépítésükből fakadóan sok lehetőség és költség jellemez, amelyeket a későbbiekben fogok kifejteni. A dolgozatomban használt Apache Flink (mely az Apache Software Foundation egyik legújabb és legmodernebb terméke) platform közel 40 millió elem feldolgozására képes egy 40 magos architektúrán másodpercenként. (Flink 2015).

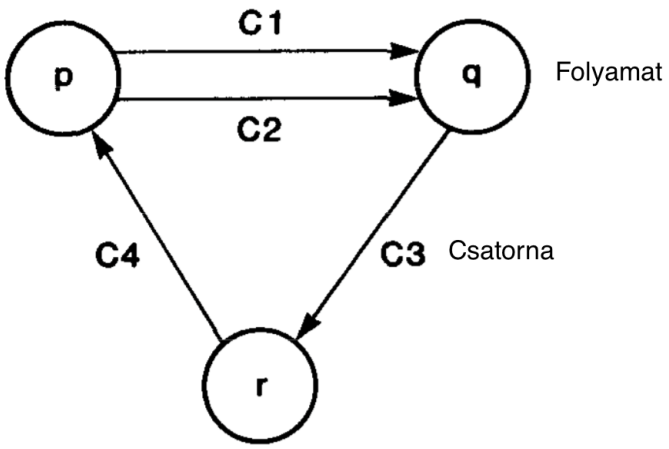
Ahhoz, hogy ezt az adatmennyiséget ki tudjuk elemezni és ajánlásokat tudjunk adni, gépi tanulásra van szükségünk. A gépi tanulás az informatikának és a matematikának egy olyan ága, amely az adatok folyamatos betáplálása során új ismereteket szolgáltat, megpróbál előrejelzéseket adni anélkül, hogy explicit módon be lenne erre programozva.(SAS 2015). A gépi tanulás egy olyan fajtáját fogom alkalmazni, ahol a termékek és felhasználóktól szerzett adatokból a lehető legpontosabban próbáljuk megjósolni, hogy milyen termék szerepelhet a felhasználó preferencialistája elején.

A választott metódus a sztochasztikus gradiens leszállás (SGD, stochastic gradient descent) (Bottou 2012), amely egy olyan egyszerűsítési illetve optimalizációs eljárás, ahol adott célfüggvény gradiensét folyamatosan, iteratív módon számoljuk ki. Dolgozatomban megtervezem Apache Flinkben az SGD algoritmust, megkezdem a szükséges módosítások implementálást és kijelölök hosszú távú fejlődési lehetőségeket.

3. Elosztottság

A megnövekedett fizikai modellezési és katonai problémák miatt már 1950-es évektől kezdve felmerült az a kérdés, hogy hogyan lehetne összetett számítási feladatokat minél gyorsabban, egyszerűbben elvégezni. (Cocke és Slotnick 1958) Rájöttek, hogy a problémákat olyan részproblémákra kell bontani, melyek megoldása egymástól független, így párhuzamosan is megoldhatóak, így jött létre a párhuzamos programozás. Párhuzamos programozás következő lépéseként jött létre az elosztott rendszerekre való igény, mely esetében a számolások már nem csak az adott számítási egységben különülnek el, hanem jóval több, akár különböző fizikai területen levő számítógép is ugyanannak a problémának egy számítását végzi. Azonban az ily módon elért számítási teljesítmény növekedés a komplexitás megnövekedésével is jár. Ezt a komplexitást különböző keretrendszerek elfedik, így biztosítva a lehetőséget elosztott alapon működő alkalmazások fejlesztésére. A komplexitás növekedése mellett a lehetőségeink is korlátozottak egy elosztott rendszer esetén.

A klasszikus megfogalmazás szerint (Chandy és Lamport 1985) egy elosztott rendszer véges számú folyamatot és csatornát tartalmaz, amik valamilyen módon kapcsolatban vannak egymással. A csatornáktól elvárjuk, hogy végtelen bufferként szolgáljanak, hiba mentesek legyenek a rajtuk átküldött üzenetek és sorrendben továbbítsák az adatot. Egy csatorna állapotán az átküldött üzenetek csoportját értjük, a folyamatokra pedig jellemző az állapotok sorozata a kezdeti állapot és események sorozata.



1. ábra. Elosztott rendszer felépítése p, q, r folyamatoknál és $c1, c2, c3, c4$ csatornák esetében (Chandy és Lamport 1985) alapján.

Az e esemény a p folyamatban atomi műveletet képvisel ami –lehet, hogy– módosítja p-t, és maximum 1 csatorna c-t. Globális állapotnak nevezzük a rendszer összes folyamatának és csatornájának állapotát.

3.1. Hibatűrés

Az informatikai rendszerek összetettségének növekedésével megnő annak a valószínűsége, hogy a rendszerünk (vagy az alkalmazásunk) nem fogja a tőle elvárt, korrekt és pontos kimenetet biztosítani. Ezek a hibafaktorok lehetnek hardver (tároló lemez kiesés) vagy szoftver (rossz adat particionálás) esetleg a köztes szoftverből (middleware) származóak, de a közös bennük, hogy általánosak és gyakori az előfordulásuk. A hibák a feldolgozó csomópontnál, vagy a kommunikációs hálózatnál jelenhetnek meg folyamatos lekérdezést –continuous query, kapcsolt irányított körmentes gráfja a kérés operátoroknak (query operator)– okozva, ami hibás vagy hiányos eredményeket produkálhat Balazinska, Hwang és Shah 2009. Ahhoz, hogy megbízható rendszert építsünk, lényeges, hogy a tervezett rendszer a különböző komponensek hibáját (közel) maximálisan tudja kijavítani. Az elosztott rendszerek kliens-szerver architektúra szerint vannak felépítve, ahol távoli eljáráshívásoknál (RPC) kommunikálunk. Különböző mechanizmusokkal kell garantálnunk a hívás megérkezését, a hibák kijavítását és korrigálását. Ezek a különböző mechanizmusok lehetnek Shenoy 2013:

- Kliens nem találja a szervert, ezért a szervernek tájékoztatnia kell a klienst a kimaradásról.
- Kliens és szerver is megfelelően működik, viszont hálózati hiba miatt a hívás nem jut el a szerverig, esetleg a szerver válasza nem jut el a klienshez. Ilyen esetekben időkorlát (timeout) bevezetése egy jó megoldás, ami során újra küldjük a hívást.
- Hívásoknak idempotensnek kell lennie, tehát a többszöri azonos hívás futtatás sem fog hibát okozni a rendszerben.

Amikor a szerver összeomlik, olyanfajta szemantikákat kell alkalmaznunk, amely biztosítja a klienst a szerver állapotáról, illetve a szerver felépülés esetén tájékoztatja a klienst a jelenlegi státuszról.
Mivel az igény a hibamentességre egyre nagyobb, különböző hibatűrési szinteket vállalhatunk a hibatűrő rendszerünkkel (Apache Akka, 2015):

- Nincs semmilyen garancia az üzenet megérkezésére.
- At-most once (legfeljebb egyszer): Minden üzenet legfeljebb egyszer kézbesítésre kerül, de ezeknek az üzeneteknek egy rész nem jut el a címzetthez (üzenetek elveszhetnek).

- At-least once (legalább egyszer): Minden üzenet legalább egyszer kézbesítésre kerül, de az üzenetek többször is eljuthatnak egy címzetthez, ami duplikált rekordokhoz vezethet (de semmiképpen nem veszik el).
- Exactly once (Pontosan egyszer): Minden üzenet pontosan egyszer kerül kézbesítésre és fogadásra, nincs elveszett és duplikált üzenet sem.
- Garantáljuk az üzenet elküldését pontosan egyszer, és ezek az üzenetek helyes sorrendben érkeznek meg.

Ahhoz, hogy hiba-toleráns rendszert alkossunk, valamilyen szintű *redundanciára* van szükségünk, amik a rendszer vagy állapot visszaállításért felelősek. Elmondhatjuk, hogy a rendszerek tipikusan egy előre definiált k egy időben történő hibát képesek kezelni. Két (Balazinska, Hwang és Shah 2009) fő fajtáját különböztetjük meg a replikációnak és a koordinációnak abban az esetben, ha feltételezzük a determinisztikusságát a számításoknak (tehát két nem hibás számítás mindig helyes értékű és sorrendű kimenetet fog produkálni fix bemenetre, más néven **konzisztensek számítások**).

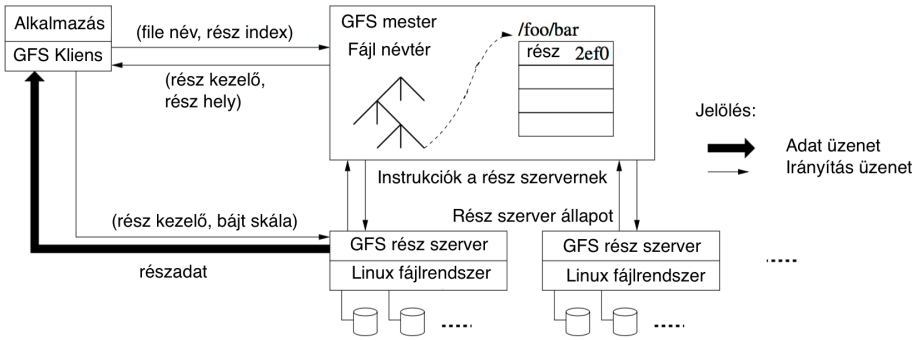
1. State-machine: a számítást $k+1 \geq 2$ egymástól független csomóponton tároljuk és mindegyik másolat az eredetivel megegyező módon kapja a bemeneteket. Előnye, hogy nagyon gyorsan megtörténhet a helyreállítás, hátránya, hogy $k+1$ erőforrást kíván a használata.
2. Rollback recovery: a rendszer periodikusan menti a számítások állapotát egy ellenőrző pontra (checkpoint), ami egy független csomóponton vagy adattárolón lesz. A tárolópontok közötti időben a rendszer automatikusan naplóállományt készít (log), és hiba esetén a legkésőbbi ellenőrzőpont és a saját log alapján felállítja a rendszert. Előnye az alacsony erőforrás igény, hátránya a lassabb visszaállítás.

Nem minden alkalmazás vagy szolgáltatás szegmensnek van szüksége arra, hogy mindig maximális pontosságú kimenetet adjon, ez az ún. részleges hiba tolerancia.

3.2. Google fájl rendszer

Google 2003-ban megalkotta a google fájl rendszert (GFS, Google File System) (Ghemawat, Gobioff és Leung 2003), ami napjaink elosztott, hibatűrő rendszereinek az alapja. Tervezésénél fontos szempont volt, hogy skálázható, megbízható, elérhető és magas teljesítménnyel rendelkező legyen mindezt úgy, hogy adat-intenzív applikációk alapját fogja szolgálni. Figyelembe vették, hogy a komponens (mind adat, mind

hardver oldalon) hibák inkább általánosak, mint kivételek, így az architektúra tervezésénél ez különös figyelmet kapott. E mellett relatív nagy fájlokra szabták (64 megabyte) és fájlok felülírása helyett (overwrite) hozzáfűzték (append) az adatokat a létező fájlokhoz. Az elrendezés könyvtár alapú, a fájlokat névtér és fájl név alapján lehet azonosítani. Támogatja a megszokott létrehozás, törlés, megnyitás, olvasás és írás operátorokat, de bevezet kettő újat is a legoptimálisabb működés miatt: pillanatkép (snapshot) aminek segítségével a fájlrendszer pillanatnyi állapotról lehet mentést készíteni és párhuzamos hozzáfűzés (record append) ami lehetővé teszi, hogy egyszerre több kliens is tudjon egy fájlt írni.



2. ábra. Google fájl rendszer felépítése (Ghemawat, Gobioff és Leung 2003)

A fájlokat 64 megabyte-os részekre (chunk) osztják, amit egy 64 kilobyte-os fájl (chunk handle) kezel. Különböző szerverek (chunk server) tárolják a részeket, és minden fájlból három másolat készül annak érdekében, hogy az adatok teljességét biztosítani tudják. Az állapotkezelést és utasításokat mester-szolga (master-slave) módon építették fel, mester tárolja memóriában a metaadatokat (névtér, fájl helyek és fájl -> chunk leképzést). Ezen kívül biztosítva van az automatikusan, pár másodperc alatt feléledő a rendszer, kritikus, infrastruktúra hiba esetén.

3.3. CAP tétel

Az elosztott rendszerek tervezésekor és használatakor három képességet várunk el elsősorban: legyen konzisztens (consistency), elérhető (availability) és particionálástűrő (partition tolerance).

- Konzisztencia: Bármelyik csomópontból kérdezem le az adatokat, mindig helyes választ kapok. Elosztott rendszereknél a konzisztencia megtartásához elengedhetetlen, hogy a csomópontok kommunikáljanak egymással.

- Elérhető: Bármelyik időpillanatban kapok választ egy kérés küldése után.

- Particionálás-tűrő: Ha –a teljes hálózati kieséstől eltekintve– egy csomópont eltűnik a hálózatról, akkor is kapok választ a rendszertől.

A CAP tétel kimondja (Gilbert és Lynch 2002), hogy ebből a háromból legfeljebb kettő lehet igaz egy adott időpillanatban. Azonban a valóság nem ennyire letisztult, különböző rendszerek különböző prioritással veszik figyelembe a három alapképességet, így korlátozottan megtalálhatjuk a három alapképességet egy modern elosztott rendszerben. Brewer 2012

3.4. Párhuzamos kötegelt adatfeldolgozás

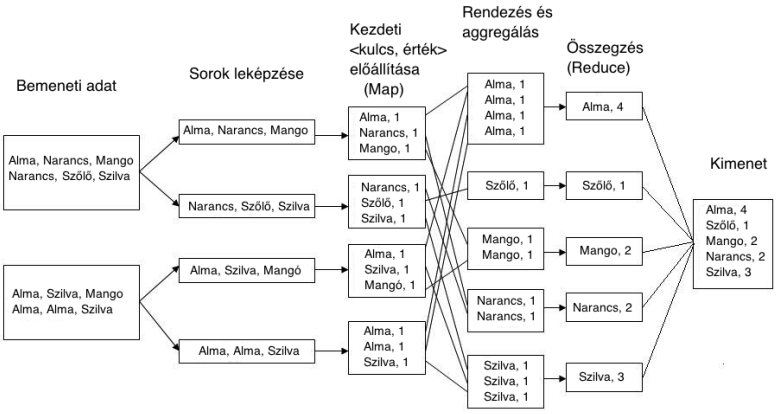
A kétezres évek elején és közepén, a megfelelő számítási kapacitás hiányában a párhuzamos adatfeldolgozás általában *kötegelt* (batch) módon történt. Felhasználói interakció nélkül, megadott időközönként történik a nagy mennyiségű adat feldolgozása. Előnye, hogy akkor futhat a program, amikor a rendszer leterheltsége alacsony, ezzel biztosítva az egyenletes kihasználtságot. Mivel a parancsok automatikusan futnak le, ezért kisebb az üres, nem számítással töltött idő. IBM 2010.

3.5. MapReduce

A Google által fejlesztett MapReduce volt az első olyan szélesebb körben is elismert programozási modell, ami lehetővé tette, hogy nagymennyiségű adatot is lehessen feldolgozni párhuzamosan, elosztott módon (Dean és Ghemawat 2004). Az adatokat <kulcs, érték> párokra bontjuk, ahol a kulcs egy referencia, ami hivatkozik az adatra, míg az érték maga az adat. A bemenet az adat, míg a kimenet a köztes <kulcs, érték> pár. Ezt az ún. Map függvény valósítja meg, amit a felhasználónak kell biztosítania. A köztes <kulcs, érték> párt a felhasználó által írt Reduce függvény dolgozza fel. A Reduce függvényben a köztes <kulcs, érték> párok összegzése zajlik, tehát megkeressük a vizsgált kulcshoz az összes értéket.
Az egyik klasszikus példa a szószámolás, ahol meg kell számolnunk azt, hogy a bemeneti adatunk szavai milyen mennyiséggel fordulnak elő.

Elsőként a bemeneti adatunkat (szövegfájl), sorokra bontjuk, majd a sorokat szavakra és a hozzájuk tartozó előfordulási értékekre (Map). Az így előállt <kulcs, érték> párokat aggregálom, majd összefésülöm (Reduce). A végső kimenet pedig ezeknek a összesített pároknak az összessége.

A ma használt keretrendszerek, az egyszerű ötletnek és magas absztrakciós szintnek köszönhetően lehetővé teszik, hogy a felhasználónak csak a konkrét adatfeldolgozó kóddal kelljen dolgoznia, mivel a keretrendszer elfedi az elosztott számítási komplexitást. Hadoop 2016



3. ábra. MapReduce alkalmazása szószámoló példán keresztül KS 2011 alapján.

3.6. Adatfolyam alapú feldolgozás

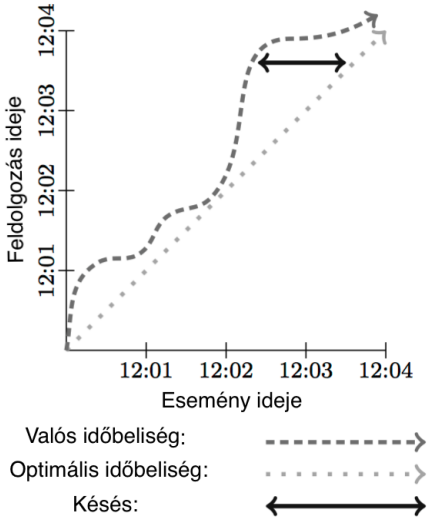
Mit is tekinthetünk adatfolyamnak? Olyan adatfeldolgozó motort, mely arra van tervezve, hogy folytonos (végtelen) és rendezetlen adatsorokat dolgozzon fel (Tyler, 2015). Eddig az ilyen rendszereket alapvetően alacsony pontossággal és/vagy megbízhatatlansággal vádolták, mely csak spekulálni tud az egyes adatok valódi értékéről. Jobb megértést tud biztosítani az, hogy ha a végtelen adatunkat valós időben dolgozzuk fel, mivel az esetek többségében az adatunk elavul, csak egy limitált időkorláton belül tudjuk értelmezni, illetve hasznos információkat kinyerni.
A rendezetlen (unorder), végtelen (unbounded) és teljes globális skálázású rendszerek kiszolgálását csak úgy lehet megoldani, ha olyan rendszert fejlesztünk, ami ténylegesen ezekre a problémákra ad megoldást (Tyler, 2015). E mellett egyéb előnyei is vannak az adatfolyam alapú rendszereknek:

- Ki tud elégíteni olyan valós idejű üzleti igényeket, mint pl.:anomália keresés, csalásfelderítés, hirdetéselhelyezés.
- Hosszú távon az erőforrás eloszlás kiegyenlítettebb lesz, mivel mindig (majdnem) akkor kerül feldolgozásra, amikor létrejön.

Ha két dolgot szem előtt tartunk a rendszer tervezésnél, akkor túl tudunk lépni a fejlettebb micro-batch rendszerek (Damji, Jules S., 2015) lehetőségein. Az első a konzisztens tárolás, ami azt jelenti, hogy hosszú idő után is, esetleges gépi hiba esetén is megmaradjanak az adatok helyes 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 időhorizont megegyezik, tehát közvetlenül akkor dolgozzuk fel az adatot, amikor az létrejött. Sajnos, azonban a bemeneti forrás késése, a feldolgozó motor hibája vagy hardver üzemszünet miatt nem lehetséges.



4. ábra. Adatfolyam feldolgozás a valóságban (Akidau et al. 2015).

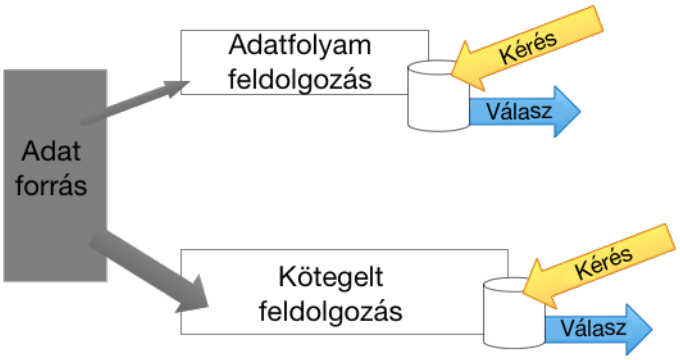
3.7. Lambda architektúra

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

- Robusztus, hiba toleráns legyen, mind hardver, mind szoftver oldalon.
- Széles körű felhasználhatóság biztosítson, alacsony válaszidővel.
- Horizontálisan skálázható legyen (több "általános célú" gép használatával lehessen növelni a teljesítményt).
- Bővíthető legyen.

Ezen feltételek mellett egy három rétegű architektúrát alkotott meg Nathan Marz (Hausenblas és Bijnens, 2015). 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 lefuttatja 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 lekérdezhetőségüket.
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.



5. ábra. Lambda architektúra felépítése Nathan Marz alapján.

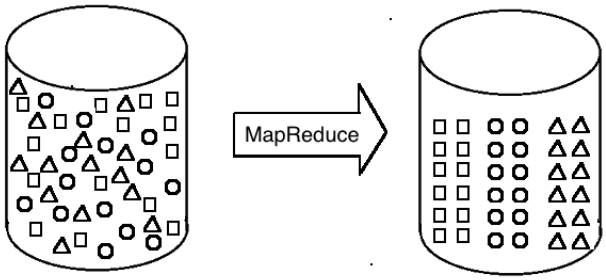
Az architektúra a maga idejében egy rendkívül jó megoldás volt, biztosítva az alacsony válaszidő és a pontosság egyvelegét. Ahogy azonban fejlődtek a technológiai megoldások, egyre jobban kiütköztek a hátrányok is. A többszörös adatfeldolgozás miatt egyszerre két infrastruktúrát kell fent tartani, ami 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 infrastruktúra fenntartása okoz. Másrészt pedig csak olyan technikai megoldásokat használhatunk, amely a két feldolgozó motor metszéspontjában szerepel.

3.8. Adatfeldolgozó mintázatok

Ahhoz, hogy tudjuk mivel dolgozik egy modern, adatfolyam alapú feldolgozó egység, részleteznünk kell az eddig használtakat (Tyler, 2015).

3.8.1. Kötegelt feldolgozás véges adaton

Az adatfeldolgozási folyamat nagyon egyszerű, vesszük a különböző típusú adatokat meghatározott időközönként és egy adatfeldolgozó architektúra (pl.: MapReduce (Dean és Ghemawat 2004)) segítségével átalakítjuk strukturált adatokká.



6. ábra. Bal oldalon található entrópikus adatokból MapReduce segítségével strukturált, információval rendelkező adatokat generálunk (Akidau 2015 cikk alapján).

3.8.2. 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 terjedt el, a rögzített ablak (windowing) és a munkamenet (session).

Rögzített ablak arra vonatkozik amely során fix méretű átmeneti időblokkokat vezetünk be, és adatainkat ezekbe az időblokkokba particionáljuk be, feldolgozó algoritmusunkat pedig itt hajtjuk végre. 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 (addicionális logikával) vagy a válaszidő növelésével(növelem a session idejét) tudjuk figyelembe venni.

3.8.3. 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.

Közelítő algoritmusok mint például Top-n (Ghosh, 2014) 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.

Ablakok létrehozásánál megkülönböztetünk az adat születése és feldolgozása felett működő modellt. A feldolgozás alapú ablakoknál a fő problémánk az, hogy az adataink nem sorrendben érkeznek meg a feldolgozó motorhoz, így előfordulhat, hogy egyes adatok lemaradnak, így rontva a feldolgozás eredményességét. Az adat születése alapú ablaknál viszont tekintettel vagyunk arra, hogy mikor születik az adat, így az azonos időben született adatokat együtt tudjuk feldolgozni.

3.9. Watermark és Trigger

Ahhoz, hogy a végtelen, adatfolyam alapú feldolgozást a lehető legjobban el tudjuk végezni, két fogalmat kell még tisztáznunk. A *watermark* (Akidau et al. 2013) egy olyan szemantika, mely nyilvántartja az összes eseményt az elosztott rendszerben, így nagy valószínűséggel meg tudjuk határozni, hogy egyes adatok késnek, vagy nem lettek a rendszernek elküldve. Tehát azt vizsgálja, hogy mennyire tekinthetünk egy bemenetet teljesnek. Konceptuálisan nézve egy függvény $F(P) \rightarrow E$, ami vesz

egy rekordot feldolgozási időben (P) és visszaadja esemény időben (E).

Másik heurisztika, ami a feldolgozást segíti azt a problémát oldja meg, hogy mikor adhatjuk át egy ablaknak az adatokat, mivel az adatok rendezetlenek az esemény idejének tekintetében, külső jelre van szükségünk, ami biztosítja ezt az információt. A *trigger* (Akidau et al. 2015) mechanizmusa biztosítja ezt számunkra, amellett, hogy az idő elteltével lehet az ablakok méretét változtatni, ezzel növelve az adatok teljességét.

3.10. Összegzés

Az eddig elmondottak alapján láthatjuk, hogy a megnövekedett számítási kapacitást és tárolási igényt csak akkor tudjuk kielégíteni, ha elosztott rendszereket használunk. Az elosztott rendszer alkalmazása egy gépi tanulási eljárás során fog megvalósulni, amit a következő fejezetben fejtek ki részletesen.

4. Gépi tanulás

Amikor tanulunk, a célunk, 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. Ebből következően minél több bemeneti adatunk és tapasztalatunk (adat) van, annál jobb okosabb és pontosabban fog előrejelezni a használni kívánt algoritmus. Gépi tanulást használunk például:

- Web keresés
- Spam szűrés
- Ajánló rendszerek
- Online hirdetések

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.

4.1. Felügyelt tanítás

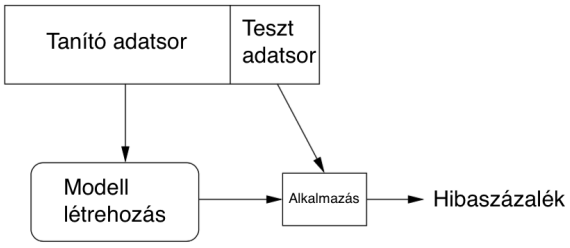
Általánosan fogalmazva, az adat amit betáplálunk a gépi tanulás modellünkbe, tréning példáknak (training set) nevezzük. A tréning példák x, y párokat tartalmaznak, ahol x az érték vektor (feature vector). Minden x érték: kategorikus (diszkrét értékek sorozatából származik, pl.: {kék, piros, sárga}) vagy numerikus (az érték egész vagy valós szám). y a címke (label), ami kategorizáló érték x-re nézve.A célunk 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 (Leskovec et al, 2014, 444.o).



7. ábra. Felügyelt tanulás általános modellje (Leskovec et al, 2014, 444.o).

4.2. Nem felügyelt tanítás

Nem felügyelt tanítás esetén adottak: $(x_1, x_2, ..., x_n)$ adataink, és nincs célfüggvényünk, vagy elvárt kimenetünk. Alapvetően nem strukturált *zajból* próbálunk mintázatot keresni, olyan modellt létrehozni, ami jól reprezentálja adatok valószínűségi eloszlását. 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 egyöntetűen származnak a $P(x)^2$-ból.Ghahramani 2004

4.3. Ajánlórendszerek

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 *hosszú farok*-nak (Wharton University, 2009), 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.

Az ajánlórendszereket definiálhatjuk olyan, főként webes rendszernek, 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
- Amazon, ahol az a cél, hogy a felhasználónak olyan termékeket ajánljunk, amit a hozzá hasonló felhasználók (k-legközelebbi analízis segítségével) is preferálnak.

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

4.3.1. Rating mátrix

Az ajánlórendszerek alapja az ún. rating mátrix, ami tartalmazza a felhasználókat és hozzájuk kapcsolt elemek adatait. Minden felhasználó, elem 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.

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

8. ábra. Rating matrix, ahol az A, B, C, D felhasználók a Star Wars, Harry Potter és Twilight filmeket értékelték(Leskovec et al, 2014, 322.o).

Célunk 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 kedvelni fogja.

4.3.2. Együttműködésen alapuló szűrés

Az együttmőködésen alapuló szűrésnél az eddig megtörtént tranzakciók illetve termék értékelések elemzése alapján dolgozunk(Koren, Bell és Volinsky 2009). Kapcsolatot elemzünk a felhasználók és a termékek között és kölcsönös függőségeket keresünk, amely során új felhasználó, termék kapcsolatot fedezhetünk fel, amit majd ajánlhatunk. Az eddigi tapasztalatok alapján az ilyen megoldások pontosabb előrejelzést tudnak adni, ha rendelkezünk megfelelő historikus adatokkal. Előfordulhat azonban, hogy olyan terméket kellene értékelnünk, amit még nem értékelt senki. Ebben az esetben historikus adat hiánya lévén, semmilyen prekoncepcióval nem rendelkezünk a tárgyalt termékkel kapcsolatban, ezt nevezik *cold-start* problémának.

Két fő fajtáját különböztetjük meg az együttműködésen alapuló szűrésnek:

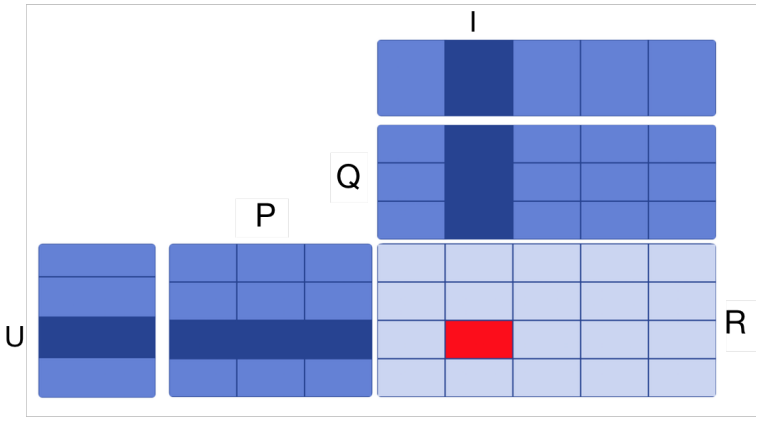
- Szomszédos elemek keresése (neighborhood methods): Ilyenkor megvizsgáljuk az egyes felhasználók és termékek közötti *távolságot*, és úgy adunk ajánlást, hogy a jól értékelt tárgyakhoz legjobban hasonlítható elemeket ajánljuk.
- Rejtett faktorok keresése (latent factor models): Próbálunk rejtett karakterisztikákat megvizsgálni az adott terméken (pl.: film esetén: mennyi erőszakot tartalmaz, mennyire drámai a cselekmény) amik specifikusan jellemzik az adott domaint. Ezek a faktorok nem egyértelműek, nehéz meghatározni a hatásukat a filmre, ezért a célunk, hogy matematikai modellekkel próbáljuk az összefüggéseket feltárni.

4.3.3. Mátrix faktorizáció

A mátrix faktorizáció a rejtett faktorokon alapuló jól skálázható, magas előrejelzési pontossággal bíró eljárás. Lényege, hogy magas fokú, alacsony sűrűségű rating mátrixunkat felbontjuk két kisebb mátrix szorzatára. Legyen $M \in \mathbb{R}$ a rating mátrixunk, amiben szerepel az összes felhasználó és hozzá tartozó értékelés. Határozzuk meg $U \in \mathbb{R}$ és $I \in \mathbb{R}$ részmátrixot, ahol U tartalmazza a felhasználókat, I pedig az termékeket és $M = |U| \cdot |I|$. Feladat, hogy megtaláljuk azt a két mátrixot $P(|U| \cdot k)$ és $Q(k \cdot |I|)$ amelyek szorzata a lehető legpontosabban megegyezik M-el:

$$M \approx P \cdot Q^T = \hat{M}$$

Ebben a formában, a mátrix faktorizációs modell leképzi a felhasználókat és a termékeket egy közös rejtett faktor f dimenzióba, ahol a felhasználó-termék interakciók a mátrix szorzatból fakadnak. Ennek megfelelően minden egyes i termék kapcsolatban van egy $q_i \in M^f$ vektorral, és minden felhasználó kapcsolatban van $p_u \in R^f$



9. ábra. Mátrix faktorizáció (DataBricks 2014 cikke alapján).

vektorral. Egy i terméknél, q_i megadja, hogy az i milyen mértékben rendelkezik az adott faktorok tulajdonságával (pozitív, vagy negatív irányban). A kapott skalárszorzat $q_i^T p_u$ rögzíti u felhasználó és i termék közötti kapcsolatot, a felhasználó átfogó érdeklődését a termék karakterisztikáiban. Ez a megközelítés alapján u felhasználó i termékhez való érdeklődését megkaphatjuk: $\hat{r}_{ui} = q_i^T p_u$.

4.3.4. Rating mátrix feltöltése

Rating mátrix nélkül nem lehetséges a felhasználóknak ajánlattétel együttműködésen alapuló szűrés esetén (cold-start probléma). Ahhoz, hogy feltudjuk tölteni, két általános megközelítés létezik.

1. *Explicit:* 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 rating mátrixot. (Pronin és Kugler 2006)

2. *Implicit:* Historikus adatokat elemzünk, azt vizsgáljuk, hogy a felhasználó megtekintette/megvette (implicit feedback) 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 rating 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 0, 1 intervallum skálán dolgozunk.

4.3.5. ALS

Mikor ajánlásokat adunk a felhasználónak, a célunk az, hogy a rating 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 egy fajtája az ALS (Alternating Least Squares).
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. A célunk tehát, hogy találjunk két olyan U és I részmátrixot, ahol U x $I^T = \hat{M}$, ahol $\hat{M}$ nagyjából megegyezik M-el.

Ha egyszerre változtatjuk a két részmátrix elemeit, akkor az NP-nehéz probléma, ezért mindig csak a P részmátrixon iterálunk végig amíg P értékei rögzítve 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.

Alapvető probléma a ma létező ajánló rendszereknél az explicitség (felhasználó által megadott értékelés) hiánya, mely az előbb említett ALS-en pár változtatást kényszerülünk tenni Hu, Koren és Volinsky 2008:

- Implicit ajánlást adunk, tehát a kapcsolatot U és I között, nem a kapcsolat milyenségét.
- Nincs negatív értékelés, ezért az értékelés hiányát negatív súllyal kell számításba vennünk.

AT&T (Hu, Koren és Volinsky 2008) által kidolgozott metódus alapján (implicit ALS) bevezetünk P_{ui} változót, ami a pozitív (1) vagy negatív (0) kapcsolatot fejezi ki, illetve C_{ui} változót, ami a P_{ui}-be "vetett" bizalom. Legyen $P_{ui} = 1$, ha $r_{ui} \geq 1$ egyébként 0. C_{ui} pedig növekvő függvénye r_{ui}-nek, pl.: $C_{ui} = 1 + \alpha * r_{ui}$.

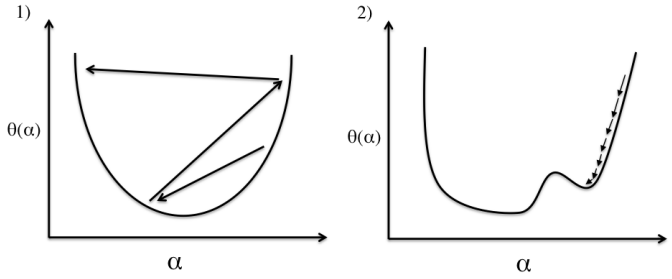
4.3.6. Gradiens leszállás

A dolgozatomban vizsgált gradiens leszállás egy olyan metódus, aminek a feladata, hogy egy általában konvex célfüggvény lokális (vagy optimális esetben 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:

$$\theta_j := \theta_j - \alpha \sum_{i=1}^{n} \frac{\delta}{\delta\theta_j} J(\theta)$$

CS229 Lecture notes 2012 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. Ezután kiválasztok egy lépésközt (tanulási ráta), ami megadja, hogy mekkora mértékben változtatom a paramétereket. Addig ismétlem a folyamatot, míg konvergenciára nem jutok vagy elérem az előre definiált ϵ küszöböt. 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.



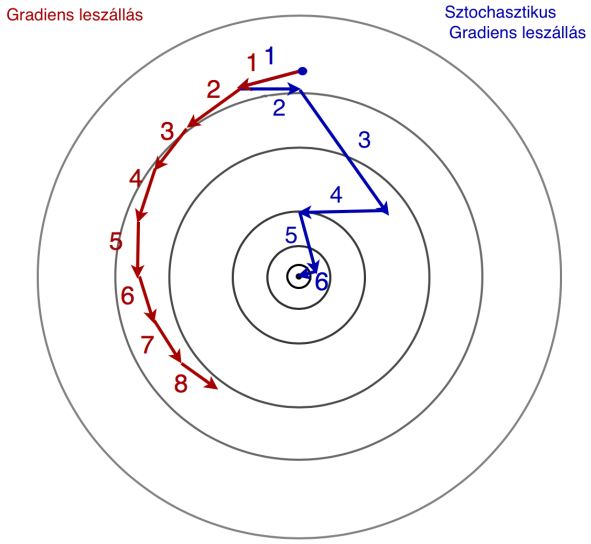
10. ábra. Konvergencia eltévesztése túl nagy és túl kicsi tanulási rátánál (Raschka 2015).

4.3.7. Sztochasztikus Gradiens leszállás

Gradiens leszállás egy jól működő, egyszerű algoritmus, mellyel lineárisan tudunk konvergenciára jutni. Azonban alapvető probléma vele, hogy nagy adatmennyiségnél („big data") költséges a megvalósítása, mivel minden egyes paramétert ki kell számolnunk ahhoz, hogy a következő iterációra jussunk. Robbins és Monroe 1951-ben Robbins és Monroe 1951 írták a tanulmányukat, melyben az úgynevezett sztochasztikus optimalizációs eljárást vizsgálták. Ebben arra jutottak, hogy nem szükséges minden lépésben az összes paramétert kiszámolni, elég véletlenszerűen venni mintákat az adatsorból. Ezt nevezzük sztochasztikus gradiens leszállásnak (SGD), ami a gradiens algoritmus egy bővített, egyszerűsített változata.
Miért jó ez? Ha minden lépésünk csak 1 deriválttól függ (tehát 2 iteráció között

csak 1 paraméter változásával számolunk), az jelentősen megtudja gyorsítani (illetve a számítási költségeket csökkenteni) a folyamatot, mivel a gradienshez képest $1/n$-es komplexitásról beszélünk, ha n jelöli a tanuló adatsorunk számosságát. E mellett arra is alkalmas a véletlenszerű mintavétel, hogy a zajos és redundáns adatokat kiszűrje (Le Roux 2009).

A 11. ábrán láthatjuk, hogyan alakulhat a konvergencia elérés a két vizsgált algoritmusnál.



11. ábra. Gradiens és Sztochasztikus Gradiens konvergencia elérés minta (Recht és Wright 2013 alapján).

Természetesen a csökkent komplexitással hátrányok is járnak. A véletlenszerű mintavétel miatt, a konvergálás véletlenszerű lesz, a variancia pedig nagy ami nem lineáris, de gyors konvergenciához vezet. Ezzel szemben a gradiens leszállással, ahol lineáris, de lassú konvergenciával dolgozunk. A pontossága sem lesz mérhető a gradiens leszálláshoz, mivel ha el is éri a lokális (vagy globális) minimumot, oszcilláló mozgást fog végezni körülötte (hacsak nem csökkentjük le az α tanulási rátát közel 0-ra (Bottou 2010)) Az, hogy mikor éri meg SGD-t alkalmazni mindig az adott üzleti alkalmazástól függ, mérlegelnünk kell, hogy a pontosságot feláldozzuk-e a sebesség oltárán.

Az SGD ajánlórendszereknél használt modellje:

$$\sum_{u,i} = (r_{ui} - P_u Q_i^T)^2 + \lambda(\| P_u \|^2 + \| Q_i \|^2)$$

Vegyük észre a jobb oldali négyzetes kifejezést, az ún. regularizációs kifejezést, ami a túltanulást hivatott megakadályozni, λ a pedig a hozzátartozó regularizációs koefficiens. Az algoritmus a következőképpen működik egy iterációnál:

1.
$$(u, i) \leftarrow rand$$

2.
$$P_u \leftarrow P_u - \alpha((r_{ui} - P_u Q_u^T)Q_i - \lambda P_u)$$

3.
$$Q_i \leftarrow Q_i - \alpha((r_{ui} - P_u Q_u^T)P_u - \lambda Q_i)$$

4.3.8. Célfüggvény

A gradiens leszállásnál láthattunk egy példát a célfüggvényre. Az ajánlórendszereknél az egyik leggyakrabban használt célfüggvény (melynek a minimumát keressük) a Root Mean Square Error, ami a függvény mért és becsült adatok négyzetes átlagának hibáját vizsgálja. Előnye, hogy a mért és ajánlott értékeket egy dimenzióban vizsgálja, így megkönnyítve a kiértékelést (Melville és Sindhwani 2008):

$$RMSE = \sqrt{\frac{\sum_{\{u,i\}}(P_{u,i} - r_{u,i})^2}{N}}$$

5. Megvalósítás

5.1. Java, Scala

Dolgozatomban Java és Scala nyelvek alatt fogom implementálni a sztochasztikus gradiens leszállás algoritmust, mivel a Flink is Java alapokon íródott. Java nagyon erős támogatással bír az Apache Software Foundation részéről, a 370 támogatott projektből 222 Java alapú (Apache Software Foundation, 2016). E mellett a rendelkezik olyan tulajdonságokkal, ami a nagyobb, elosztott rendszerek fejlesztésének kezdeténél –'90-es évek közepe és vége– nagyon kevés nyelv rendelkezett:

- Objektum orientált alapok, mely a nagy komplexitású projektek felépítését megkönnyíti.
- A beépített függvénykönyvtár erős hálózati (TCP/IP, UDP, HTTP, stb) támogatással rendelkezik.
- Platformfüggetlenség.
- Kivétel és hibakezelés, try/catch.
- Hálózati hatás: a Java nyelv elterjedtsége nagyon magas, az egyik legnépszerűbb nyelv a világon (TIOBE 2016), ami azt jelenti, hogy nagyon kiterjedt és fejlett külső keretrendszer jellemzi.

Scala –Scalable Language kifejezésből származtatva Odersky 2016– az elmúlt pár évben lett közismert a nagy adatfeldolgozó, elosztott projektek kapcsán. A vegyes objektum-orientált és funkcionális megközelítés lehetővé teszi, hogy a programozók könnyen átálljanak a nyelvre, mégis kihasználjanak újfajta paradigmákat (first-class functions, immutable data structures, immutability over mutation). Java Virtuális Gép (JVM) alapokon íródott, ezért lehetővé teszi, hogy a megszokott Java könyvtárakkal is dolgozzunk, és emiatt közel azonos sebességre is képes. E mellett típus-biztos (type-safe), tehát a fordítási időben ellenőrzi a változók típusát és értékét, így elkerülve a futási időben keletkező hibákat (Toptal, 2014).

5.2. Flink

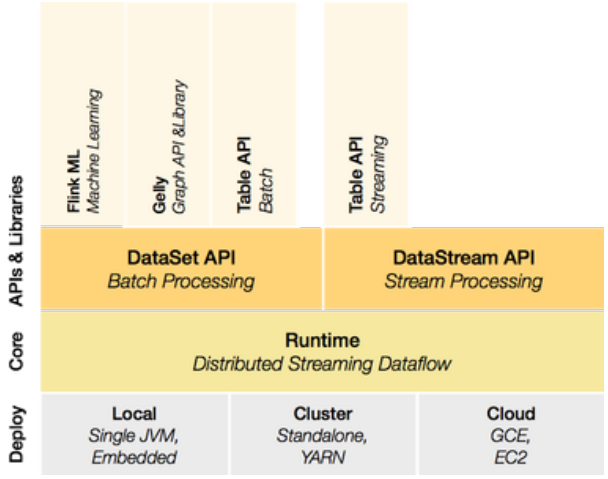
Az Apache Software Foundation által támogatott Flink egy olyan keretrendszer, ami biztosítja, hogy valós időben, elosztott rendszereken, *végtelen adaton* is kellő pontossággal tudjunk adatelemzést végezni, anélkül, hogy bonyolult, csak a feladatra tervezett architektúrát kellene létrehoznunk. Ezt az egyszerűséget és pontosságot következő tulajdonságaival éri el (Ewen és Tzoumas 2015)

- Egyszerűsített adatfeldolgozó csővezeték (pipeline), amely az adat megkapásától az adat feldolgozásáig tart
- Az adatot olyan módon modellezi és dolgozza fel, ahogy az létrejött, valósidőbeli események alapján
- Nem sorrendben érkezett adatokból származó hibák kiküszöbölése azzal, hogy esemény alapú ablakok létrehozását támogatja
- Biztosítja, hogy a lehető leghamarabb de szükségszerűen legkésőbb történjen az adatfeldolgozás, szem előtt tartva a még meg nem érkezett adatokat

Az eddig elterjedt megoldások többségében csak kötegelt (Apache Spark) vagy csak adatfolyam feldolgozásra (Storm 2016, IBM 2016) használhatóak. Flink alapvetően váltja meg ezt a paradigmát, egyszerre képes kötegelt és adatfolyam feldolgozásra is, úgy, hogy a programozási modell és futtató környezet is megegyezik a két esetnél. A megismert esemény és feldolgozási idő mellett egy új fogalmat is bevezet: belépési idő, (ingestion time) ami egy olyan hibrid megoldás(Ewen, 2016), ami azt mutatja, hogy mikor került a Flink-ben az adott esemény. Ez alacsonyabb késleltetést biztosít mint az esemény idő, és pontosabb eredményt mint a feldolgozási idő.

A kötegelt feldolgozást egy, az adatfolyam specifikus részének tekinti, így tudja megvalósítani a közös architektúrát. Négy fő részről beszélhetünk, a bevezetés (deployment), központ (core), API és könyvtárak. A központi részben találhatjuk meg az elosztott adatfolyam motort, ami végrehajtja a az adatfolyam programokat. Futásidőben, egy irányított körmentes gráf (DAG) állapottartó operátorokhoz csatlakozik az adatfolyamokon. Az API DataSet és DataStream részekre van osztva melyek a kötegelt és folytonos adatfeldolgozásért felelnek. Különböző domain specifikus nyelvekkel (DSL) oldották meg felsőbb rétegű függvénykönyvtárak alkalmazását: FlinkML (gépi tanulás), Gelly (gráf feldolgozás) és Table (SQL jellegű lekérdezések) könyvtárak állnak a felhasználók rendelkezésére. Futásidőben három típusú folyamatot kezelünk:

- Kliens: program kódot átalakítja adatfolyam gráffá, és elküldi a Job Managernek.
- Job Manager: felel az elosztottságért és nyilvántartja a program állapotát és előrehaladását minden operátornak és adatnak, e mellett új operátorok időzítéséért, ellenőrző pontok felállításáért és a rendszer felállításáért is felel.
- Task Manager: adatfeldolgozás a fő feladata.



12. ábra. Apache Flink felépítése http://flink.apache.org alapján.

5.3. Flink Streaming

Flink Streaming a központi API kiterjesztése, mely segítségével nagy áteresztő képességű (C. et al. 2015), alacsony várakozási idejű adatfeldolgozást hajthatunk végre. A rendszerbe különböző előre definiált (pl.: RabbitMQ, Twitter) vagy felhasználó által biztosított forrásokból csatlakoztathatunk adatfolyamokat. Az adatfolyamot magas szintű metódusok segítségével transzformálhatjuk és módosíthatjuk, úgy mint a nem Streaming feldolgozás esetében.

Flinkben a DataStream olyan adatfolyamot reprezentál, mely egyazon típusú adatokból áll fel, (közel) valós időben, általában előre nem tudható ideig közvetítünk a programunknak. Ilyen adatfolyam lehet például egy üzenetsor (message queue), socket illetve egy fájl is. Az eredményeket úgynevezett "sink"-ként közvetítjük, így adatot írva fájl-ba, vagy a standard kimeneten (parancssor) megjelenítve. Ahhoz, hogy tudjuk, hogy dolgozunk egy Flink programban, pár alapvető fogalmat ismertetnék Flink 2016a:

- Adatfolyam Transzformáció: Ezeknek a segítségével alakíthatunk át egy vagy több adatfolyamot egy újabb adatfolyamba. Ilyenek például a Map (egy elemet vesz inputnak, és egy elemet ad tovább), a $FlatMap$ (egy elemet vesz inputnak és nulla, egy vagy több kimenetet generál) illetve a $Filter$ (bemeneti elemre egy Bool függvényt futtat le, és azokat adja tovább amik Igaz értékkel térnek vissza) transzformátorok.

- Feladatláncolás (task chaining): Ahhoz, hogy minél optimálisabban fusson a Flink programunk, ún. feladatláncolásra van lehetőségünk, mely során az egymást követő transzformációkat egy szálon tudjuk futtatni. $someStream.filter(...).map(...).st$

esetében először egy $filter$ és map metódust valósítunk meg, míg ezután ugyanazon a szálon futtatva még egy map metódust.

- Adatforrások: A Flink lehetőséget ad többféle adatforrás csatlakoztatására is. Előre definiált lehetőségek esetén dolgozhatunk fájlokkal, socketről érkező információkkal, collection-el illetve egyedi forrással is, mely esetén egy külső adattároló megoldást csatlakoztatunk a Flinkhez (pl.: Apache Kafka, Metzger és Tzoumas 2015).

- Data Sinks: Egy adatfolyamot vár bemenetnek és kimenetként létrehozhat fájlt ($writeAsCsv(...)$), tovább küldheti az adatot egy socketnek ($writeToSocket$) illetve akár ki is irathatjuk az eredményt a konzolra ($print()$).

- Iterációk: Előfordulhat olyan eset, amikor a programunk egy részét szeretnénk *végtelenszer* lefuttatni, nem tudunk megadni egy véges számú iterációt. Ebben az esetben eldönthetjük, hogy programunk az adat mely részét továbbítja és mely részét küldjük újra az iterációba.

- Lusta kiértékelés (lazy evaluation): Amikor a program $main$ metódusa lefut, az adatbetöltés és a transzformáció nem fog megtörténni. E helyett a műveleteket a Flink feldolgozza, és akkor fognak lefutni, amikor explicit $execute()$ hívás történik a $StreamExecutionEnvironment$ objektumon függetlenül attól, hogy a program lokálisan vagy klaszteren fut. Ennek segítségével bonyolultabb programokat is holisztikusan, előre megtervezett egységként tudja kezelni a Flink.

- Adattípusok: Flink néhány megszorítással él az adatok típusát illetően azért, hogy a lehető legoptimálisabban tudjon a rendszer futni. A 6 fő kategória a Java Tuple, Scala Case osztály, Java POJO, Primitív típusok, egyéb generális osztályok, értékek, Hadoop típusok.

A következőkben egy szószámoló adatfolyam alkalmazás példája látható, ami 5 másodperces ablakokban aggregálja a web socketről érkező adatokat (Flink 2016b):

```
import java.util.concurrent.TimeUnit

//Flink futtatásához szükséges könyvtárak importálása
import org.apache.flink.streaming.api.scala._
import org.apache.flink.streaming.api.windowing.time.Time


object WindowWordCount {
  def main(args: Array[String]) {

    //adatfolyam létrehozása, melyen műveleteket végzünk
    val env = StreamExecutionEnvironment.getExecutionEnvironment

    //adatbemenet létrehozása
    val text = env.socketTextStream("localhost", 9999)

   //traszformáció, aggregálás elvégzése
    val counts = text.flatMap { _.toLowerCase.split("\\W+") filter {
        _.nonEmpty } }
      .map { (_, 1) }
      .keyBy(0)
      .timeWindow(Time.of(5, TimeUnit.SECONDS))
      .sum(1)

    counts.print

   //Program futtatása
    env.execute("Window Stream WordCount")
  }
}
```

5.4. FlinkML motiváció, tervezés

A Flinkben használt FlinkML (gépi tanulás, machine learning) egy olyan keretrendszer a Flinkhez, mely biztosítja a skálázható gépi tanulás eszközöket. Tervezése és megvalósítása során fő szempont, hogy minimálisra redukálják a "glue code"-ot, amely az end-to-end gépi tanulás rendszerekben lévő technológia megoldások összekötésénél járulékos. Ennek a glue code-nak a csökkentésével hozzájárulnak a manapság széles körben elterjedt technológiai adósság (technical debt) csökkentéséhez, amely

egy idő után feleslegesen megnöveli a projekt és kódbázis komplexitását (Sculley et al. 2014). E mellett biztosítják, hogy Flink tartalmazzon olyan adatkezelő lehetőségeket, melyek biztosítják az adatfeldolgozást, adatmanipulációt és eredmény kiértékelését a keretrendszer korlátain belül. Szempont volt még a fejlesztés során, hogy a szélesebb körben elterjedt, már ismert terminológiát és ötleteket használják, ezzel lecsökkentve a tanulási görbét. Ilyen keretrendszerek pl.: Apache Spark-ban használt MLlib illetve a Python-hoz használt tudományos körökben is elismert scikit-learn.

A képesség, hogy több műveletet egy sorozatba kapcsoljunk (pipeline) egy olyan elvárás, amit manapság minden modern gépi tanulással foglalkozó keretrendszernek tudnia kell. Ilyen művelet sorozat esetén a bemenő adatunk lehetnek adatok, vagy egy megelőző művelet sorozat kimenete. A kimenetünk pedig a transzformált adat, melyet használhatunk elemzéshez, vagy tovább adhatjuk egy következő feldolgozási rétegnek. A könnyű kezelhetőséget és alkalmazásprogramozási interfészt a scikit-learnből (Buitinck et al. 2013) ismert módon implementálták:

- Estimator, az ősosztály, mely tartalmazza a fit() metódust, ami a modell tanítását végzi.
- Transformer, amely az átalakító műveleteket végzi a bemeneten.
- Predictor, amely végrehajtja a tanulást, és az előrejelzést ad.

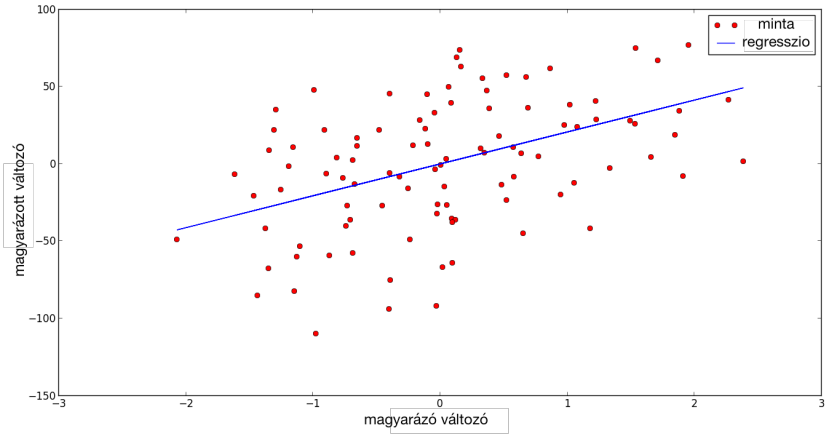
Egy lehetséges (Scala) megvalósítás a következő:

```
//Tanitó adat beolvasása
val input: DataSet[LabeledVector] = ...
// Teszt adat beolvasása
val unlabeled: DataSet[Vector] = ...

val scaler = StandardScaler()
val polyFeatures = PolynomialFeatures()
val mlr = MultipleLinearRegression()

// Pipeline létrehozása
val pipeline = scaler
  .chainTransformer(polyFeatures)
  .chainPredictor(mlr)

// Pipeline tanítása
pipeline.fit(input)
```

```
// Előrejelzés kiszámítása a teszt adaton
val predictions: DataSet[LabeledVector] = pipeline.predict(unlabeled)
```

5.5. Lineáris regresszió

A dolgozatomban használt függvény, melyet optimalizálok a lineáris regresszió. A lineáris regresszió széles körben használt a gépi tanulási problémáknál, segítségével egyszerűen meg tudunk olyan problémákat oldani, ahol a kapcsolat a magyarázó és a magyarázott változó között lineáris.



13. ábra. Lineáris regresszió minta

Tegyük fel, hogy egy egyenest szeretnénk a pontokra illeszteni. Ehhez az ismert $y = mx + b$ egyenes egyenletét használhatjuk, ahol m az egyenes meredeksége, b pedig megmutatja, hogy hol metszi az egyenes az y tengelyt. Ahhoz, hogy megtaláljuk a legpontosabban illeszkedő egyenest, a legpontosabb m és b értékeket keressük. Egy megszokott megközelítés a probléma megoldásához, ha létrehozunk egy célfüggvényt, ami méri az egyenes "pontosságát". A metódus (m, b) párokat fog bemenetnek várni, és visszatérési értékként megadja, hogy mennyire illeszkedik az egyenesünk a pontokra. A kiszámításhoz végigiterálunk az összes (x, y) adatpontunkon és összegezzük a négyzetes különbségeket az y pontok értéke és a vonal y értéke között: $\frac{1}{N}\sum_{i=1}^{N}(y_i - (mx_i + b^2))$.

Azok az egyenesek illeszkednek jobban a pontjainkra, ahol kisebb a célfüggvényben kiszámolt hiba. Tehát akkor találjuk meg a legpontosabb egyenest, ha a függvény minimumát keressük. Amikor gradiens leszállás műveletet végzünk el, véletlenszerű paraméterekkel kezdünk számolni. Ahhoz, hogy optimalizálni tudjunk, a függvény gradiensét kell kiszámolnunk. A gradiens egyfajta "iránytűként" fogja megmutatni, hogy milyen irányban kell változtatnunk a paramétert, hogy a függvény értéke csök-

kenjen. Mivel a függvényünk két paraméterből áll (m és b) ezért parciális deriválást kell végrehajtanunk, ami:

$$\frac{\delta}{\delta m} = \frac{2}{N}\sum_{i=1}^{N} -x_i(y_i - (mx_i + b))$$

illetve

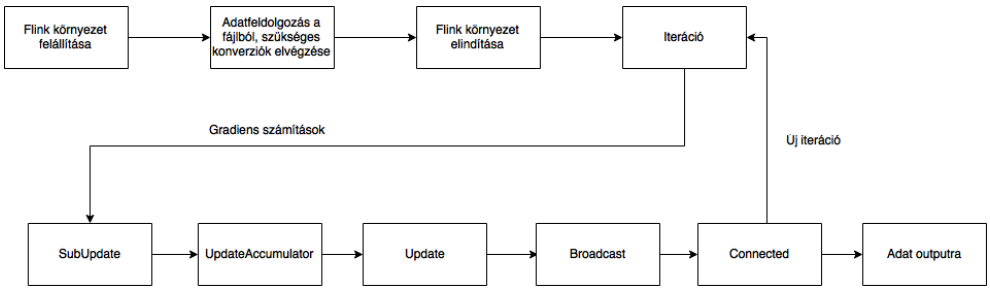
$$\frac{\delta}{\delta b} = \frac{2}{N}\sum_{i=1}^{N} -(y_i - (mx_i + b))$$

deriváltakat eredményezi.

Inicializálhatunk egy bármilyen m és b párt, és hagyjuk, hogy a gradiens leszállás folyamatosan csökkentse a célfüggvényt. Minden egyes iteráció során az m és b értéket úgy frissítjük, hogy az egyenesünk egy kicsit jobban illeszkedik a pontokra, mint az előző paramétereknél.

5.6. Elosztott sztochasztikus gradiens leszállás megvalósítása

A megvalósított sztochasztikus gradiens leszállás Java és Scala alapokon valósítottam meg. Mindkét osztály UML diagramja a függelékben található meg.



14. ábra. Program életciklusmodellje

Minden egyes Flink program esetén szükségünk van arra, hogy a Flink környezetet felállítsuk, ezzel megalapozva a későbbi működést. Ezt követően az adatfeldolgozást hajtjuk végre, amely során elvégezzünk a szükséges (String => Double) konverziókat, azért, hogy elkerüljük a futásidőben fellépő esetleges hibákat. Az adatfeldolgozás során egy statikus fájlból vesszük a bemeneti értékeket. Következő lépésként elindítjuk a Flink környezetet, és a feldolgozott adatot beolvassuk egy Flink Collectionbe, mely során megtörténik az adatok leképzése, és a fizikai elosztása.

A gradiens számítás megvalósításához 3 saját implementált metódussal dolgoztam. A *SubUpdate*(), az *UpdateAccumulator*() és az *Update*() felelős azért, hogy a paraméterek deriválása és frissítése megtörténjen minden egyes iteráció során. Végül

elvégezzük a particionálást ($Broadcast()$), hogy kihasználjuk a Flink elosztott rendszer előnyeit, és egy véletlen szám generátor segítségével eldöntjük, hogy a feldolgozott adatot a kimenetre, esetleg újabb iterációra küldjük ($Connected()$).

6. Fejlesztési lehetőségek

Továbbfejlesztési lehetőségeket több szempontból is meg lehet vizsgálni. Először is a már létező funkciók optimalizálásával lehet foglalkozni. A konvergencia elérése az algoritmusnál nem túl hatékony, ez nagy adatmennyiségnél jelentős erőforrás veszteségként jelenhet meg, mivel minden egyes iteráció után 50% az esélye, hogy tovább iterálunk. Ahhoz, hogy konvergenciára jussunk, a tanulási ráta (α) egy idő után folyamatosan csökken egészen 0-ig. Azonban ez az életciklus végén már nagyon lassan megy végbe, ezért úgynevezett új variancia változót lehetne bevezetni, mellyel a tanulási rátát optimálisabban tudnám változtatni. (Johnson és Zhang 2013).

Egyes publikációk szerint (Bottou 2012) annak ellenére, hogy a sztochasztikus gradiens leszállásnál véletlenszerűen járjuk végig az adatsorunkat, segíthet az algoritmus hatékonyságán, ha összekeverjük az adatokat, így az esetleges csoportosulások megbomlanak. Ezt segítve optimális megoldás lenne, ha az osztályunk rendelkezne azzal a képességgel, hogy feldolgozás előtt véletlenszerű sorrendbe rendezze az adatokat.

A jelenlegi állapotában az osztály nem képes külső adat feldolgozására, a *LinearRegressionDat* fájl segítségével dolgoztam, így a jövőben a bemeneti adat paraméterezhetővé tétele is egy jó kiindulópont.

7. Összefoglalás

Dolgozatomban egy elosztott rendszeren futó ajánlórendszer megvalósítását kezdtem el Apache Flinkben. A megírása előtt arra voltam kíváncsi, hogy milyen módon lehet az implementálást elvégezni, milyen matematikai, informatikai és közgazdasági háttérirodalom ismerete szükséges ahhoz, hogy valaki egy jól működő, a való élet problémáit megoldó rendszert készítsen.

A dolgozat első részében bemutattam, hogy mit is jelent az elosztottság és a hibatűrés, miért fontos az, hogy különböző technológiákkal és megoldásokkal megelőzzük azt, hogy az adatunk megsemmisüljön, vagy ellentétes esetben többször is eljusson a címzetthez. Ezt alátámasztva ismertettem CAP-tételt, mely során kifejtettem, hogy miért nem lehetséges az, hogy konzisztens, elérhető és particionálás-tűrő legyen egy adott időpillanatban a rendszer.

Az elméleti bevezetés után megmutattam a Google fájl rendszert, ami a modern adattárolás és adatelőhívás alapja. Ezt követően ugyancsak a Googlenek köszönhetően, megismerhettük a MapReduce technológiát, ami lehetővé tette, hogy az akkor még nagyon bonyolult, alacsony szintű, párhuzamos vagy elosztott programozást, egy magasabb absztrakciós szinten, akár egy nem specializáltan ezzel a témakörrel foglalkozó programozó is viszonylag egyszerűen kihasználja.

Fontosnak tartottam megemlíteni, hogy a korai kötegelt adatfeldolgozáson alapuló technológiáknak milyen hátrányai vannak, és milyen feltételeknek kell egy rendszernek ma megfelelnie, hogy túlnőhessen az architektúra adott korlátokon. Ezek után bemutattam egy már modernebb, jól működő, ám rengeteg felesleges fenntartási költséggel járó Lambda-architektúrát, mely teljesítményben már közel járt az optimálishoz, azonban a megoldás, hogy párhuzamosan futtatunk egy kötegelt és egy adatfolyam alapú architektúrát, a programozási és az infrastrukturális költségek jelentősen magasabbak az elvártnál. A legismertebb adatfeldolgozási mintázatok és heurisztikák megismerésével eljutottunk arra a pontra, hogy egy modern adatfeldolgozó rendszer alapjait lefektettük illetve megismertük, hogy milyen optimalizálási lehetőségeink vannak, ha egy ilyen rendszer építésébe kezdünk.

A dolgozat másik fő témája a gépi tanuláshoz köthető, vagyis az olyan rendszerek, ahol explicit programozás nélkül, csupán a historikus adatok segítségével tudunk bonyolult problémákat megoldani, melyeket a hagyományos eszközökkel nem tudnánk. Bemutattam a két fő kategóriáját, a felügyelt és a nem felügyelt tanítást, a ma ismert gépi tanulási tudománynak, amibe a ma ismert *mesterséges intelligencia* jellegű problémák nagy részét bele tudjuk illeszteni. Ezt követően az ajánlórendszerek témáját ismertettem, ahol világossá vált, hogy a manapság a végtelen információ idejében szükséges az, hogy a felhasználóknak valamilyen módon ajánlásokat adjunk, ami meg-

közelítőleg megfelel a preferenciájának, így elősegítve a választást a termékek közül. Az ajánlórendszerek megértéséhez szükséges tudni, hogy az egyes felhasználó-termék leképzéseket egy mátrixban tároljuk, és ezek a mátrixok ritkák (mivel a felhasználók az összes termék csak meglehetősen kis hányadát értékelik), így ezen mátrixok kitöltése erőforrásigényes.
Ennek a problémának a megoldására vázoltam fel az ALS algoritmus, mely során az úgynevezett *rating mátrixunkat* két, olyan U és I részmátrixokra bontjuk, amelyek szorzata hozzávetőlegesen megegyezik az eredeti mátrixunkkal. Egy másik közismert lehetőség az ajánlórendszerek létrehozására, a dolgozatomban is vizsgált (sztochasztikus) gradiens leszállás, mely során a cél, hogy minél kevesebb iteráció során konvergenciára jussak, tehát a hibát a valós és predikált eredmény között minimalizáljam. Ismertettem a minimalizálandó függvényt, ami az ajánlórendszereknél a mért és becsült adatok négyzetes hibájából adódik (RMSE).

A dolgozatom utolsó részében bemutattam, az implementáláshoz használt Java és Scala nyelvet, illetve az elosztottsághoz használt Apache Flink keretrendszert is. A dolgozat egy másik motivációja volt, hogy eddig ha ajánlórendszert szerettünk volna létrehozni Flink segítségével, csak ALS implementálását végezhettük el, azonban egyes esetekben az SGD hatékonyabban tud ajánlásokat adni. (Aberger R. 2014). Az implementáció elkészültével javaslatokat adtam arra, hogy milyen lehetőség van a program továbbfejlesztésére.

Az Apache Flink, dolgozatom megírásakor elérte az 1.0-s verziószámot, ami azt mutatja, hogy van létjogosultsága egy általános célú, nagy teljesítményű, alacsony késleltetési idővel dolgozó rendszerre, Az egyszerű kezelhetőség, a magas szintű alkalmazásprogramozási interfész, illetve a lehetőség, hogy (majdnem) azonos logikával hozhatunk létre kötegelt és adatfolyam alapú rendszereket hozzájárulnak ahhoz, hogy a jövőben elterjedjenek az ilyen, és ehhez hasonló alkalmazások.

8. Köszönetnyilvánítás

Hivatkozások

Aberger R. C. (2014). *Recommender: An Analysis of Collaborative Filtering Techniques*. URL: http://cs229.stanford.edu/proj2014/Christopher%20Aberger,%20Recommender.pdf.

Akidau T. (2015). *The world beyond batch: Streaming 101*. URL: http://radar.oreilly.com/2015/08/the-world-beyond-batch-streaming-101.html (Letöltve: 12/17/2015).

Akidau T. Et al., (2013). *MillWheel: Fault-Tolerant Stream Processing at Internet Scale*, old. 734–746.

Akidau T. Et al., (2015). *The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-Scale, Unbounded, Out-of-Order Data Processing*. URL: http://www.vldb.org/pvldb/vol8/p1792-Akidau.pdf.

Balazinska M. Hwang J. Shah M. A. (2009). *Fault-tolerance and high availability in data stream management systems*. URL: https://homes.cs.washington.edu/~magda/encyclopedia-short.pdf.

Bottou L. (2010). *Large-Scale Machine Learning with Stochastic Gradient Descent*.

— (2012). *Stochastic Gradient Descent Tricks*. URL: http://research.microsoft.com/pubs/192769/tricks-2012.pdf.

Brewer E. (2012). *CAP Twelve Years Later: How the "Rules" Have Changed*. URL: http://www.infoq.com/articles/cap-twelve-years-later-how-the-rules-have-changed (Letöltve: 12/30/2015).

Buitinck L. Et al., (2013). *API design for machine learning software: experiences from the scikit-learn project*. URL: http://arxiv.org/abs/1309.0238.

C. Paris et al., (2015). *Lightweight Asynchronous Snapshots for Distributed Dataflows*. URL: http://arxiv.org/abs/1506.08603.

Chandy K. M. Lamport L. (1985). *Distributed Snapshots: Determining Global States of Distributed Systems*. URL: http://research.microsoft.com/en-us/um/people/lamport/pubs/chandy.pdf.

Cocke J. Slotnick D.S. (1958). *THE USE OF PARALLELISM IN NUMERICAL CALCULATIONS*. IBM RESEARCH CENTER.

CS229 Lecture notes (2012). URL: http://cs229.stanford.edu/notes/cs229-notes1.pdf (Letöltve: 12/21/2015).

DataBricks (2014). *Movie Recommendation with MLlib*. URL: https://databricks-training.s3.amazonaws.com/movie-recommendation-with-mllib.html (Letöltve: 01/28/2016).

Dean J. Ghemawat S. (2004). *MapReduce: Simplified Data Processing on Large Clusters*, old. 10–10. URL: http://dl.acm.org/citation.cfm?id=1251254.1251264.

Emmerich W. (2015). *Distributed System Principles.* URL: http://www0.cs.ucl.ac.uk/staff/ucacwxe/lectures/ds98-99/dsee3.pdf (Letöltve: 12/15/2015).

Ewen S. Tzoumas K. (2015). *How Apache Flink enables new streaming applications.* URL: http://data-artisans.com/how-apache-flink-enables-new-streaming-applications-part-1/ (Letöltve: 12/30/2015).

Felice F. R. Reid (2008). *Big data: Distilling meaning from data.*

Flink Apache (2015). *Streaming.* URL: http://flink.apache.org/features.html (Letöltve: 12/15/2015).

— (2016a). *Flink DataStream API Programming Guide.* URL: https://ci.apache.org/projects/flink/flink-docs-master/apis/streaming/index.html#task-chaining-and-resource-groups (Letöltve: 04/26/2016).

— (2016b). *Flink Programming Guide.* URL: https://ci.apache.org/projects/flink/flink-docs-master/apis/streaming/ (Letöltve: 04/28/2016).

Ghahramani Z. (2004). *Unsupervised Learning.* URL: http://mlg.eng.cam.ac.uk/zoubin/papers/ul.pdf (Letöltve: 12/16/2015).

Ghemawat S. Gobioff H. Leung S. (2003). *The Google File System.* Google.

Gilbert S. Lynch N. (2002). *Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-tolerant Web Services.* 2. New York, NY, USA, old. 51–59. DOI: 10.1145/564585.564601. URL: http://doi.acm.org/10.1145/564585.564601.

Hadoop Apache (2016). *What Is Apache Hadoop?* URL: https://hadoop.apache.org/ (Letöltve: 01/04/2016).

Hu Y. Koren Y. Volinsky C. (2008). *Collaborative Filtering for Implicit Feedback Datasets.* URL: http://yifanhu.net/PUB/cf.pdf.

IBM (2010). *Mainframes working after hours: Batch processing.* URL: http://www-01.ibm.com/support/knowledgecenter/#!/zosbasics/com.ibm.zos.zmainframe/zconc_batchproc.htm (Letöltve: 12/16/2015).

— (2016). *IBM InfoSphere Streams.* URL: http://www-03.ibm.com/software/products/en/ibm-streams (Letöltve: 01/08/2016).

Johnson R. Zhang T. (2013). *Accelerating Stochastic Gradient Descent using Predictive Variance Reduction.* URL: http://papers.nips.cc/paper/4937-accelerating-stochastic-gradient-descent-using-predictive-variance-reduction.pdf.

Koren Y. Bell R. Volinsky C. (2009). *MATRIX FACTORIZATION TECHNIQUES FOR RECOMMENDER SYSTEMS.*

KS B. (2011). *Word Count - Hadoop Map Reduce Example.* URL: http://kickstarthadoop.blogspot.hu/2011/04/word-count-hadoop-map-reduce-example.html (Letöltve: 12/21/2015).

Le Roux N. (2009). *Using Gradient Descent for Optimization and Learning*. URL: http://www.gatsby.ucl.ac.uk/teaching/courses/ml2-2008/graddescent.pdf (Letöltve: 01/06/2016).

Melville P. Sindhwani V. (2008). *Recommender Systems*. URL: http://vikas.sindhwani.org/recommender.pdf.

Odersky M. (2016). *A Scalable language*. URL: http://www.scala-lang.org/what-is-scala.html (Letöltve: 01/05/2016).

Pronin E. Kugler M. (2006). *Valuing thoughts, ignoring behavior: The introspection illusion as a source of the bias blind spot*. Princeton University.

Raschka S. (2015). *How Machine Learning Algorithms Work Part 1*. URL: http://sebastianraschka.com/Articles/2015_singlelayer_neurons.html (Letöltve: 01/08/2016).

Recht B. Wright S. (2013). *Optimization*. URL: https://simons.berkeley.edu/sites/default/files/docs/522/rechtslides.pdf (Letöltve: 01/08/2016).

Robbins H. Monroe S. (1951). *A Stochastic Approximation Method*.

SAS (2015). *Machine Learning: What it is and why it matters*. URL: http://www.sas.com/en_us/insights/analytics/machine-learning.html (Letöltve: 12/16/2015).

Sculley D. Et al., (2014). *Machine Learning: The High Interest Credit Card of Technical Debt*.

Shenoy P. (2013). *Failure Semantics*. URL: http://lass.cs.umass.edu/~shenoy/courses/spring13/lectures/Lec07.pdf (Letöltve: 01/06/2016).

Storm Apache (2016). *Storm*. URL: http://storm.apache.org (Letöltve: 01/08/2016).

TIOBE (2016). *TIOBE Index for April 2016*. URL: http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html (Letöltve: 04/25/2016).

Ábrák jegyzéke