Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar

TDK Dolgozat

Charaf Kamel

Anomália detektálás logfájlokban gépi tanulással

KONZULENS
Dr. Horváth Gábor

BUDAPEST, 2022

Absztrakt

Az adatok világában élünk. Sok adat keletkezik környezetünkben szenzorokból, közösségi médiából és számos egyéb forrásból. A komplex informatikai rendszerek kivétel nélkül rendelkeznek logfájlokkal, amelyek segítségével pontosabb képet kapunk a rendszerek működéséről, állapotáról és helyzetéről. Ezek az adatok számtalan információt hordoznak melyek nagy értéket képviselhetnek az operátorok számára. Előfordulnak olyan esetek, amikor anomáliákat fedezünk a logadatokban. Anomáliáról akkor beszélünk, ha eddig ismeretlen összetételű vagy sorrendű logsor érkezik. A logsorok egy nagyon fontos tulajdonsága, hogy rendszerint ismétlődő mintázattal rendelkeznek és megoldásomban ezt fogom felhasználni.
A feladatom az anomáliák észlelésével, hogy jelentsük a szokatlan logsor-szekvenciákat. Mivel a rendszerek folyamatosan logol-nak, így hatalmas adathalmaz keletkezik, ami nagyon megnehezíti a manuális feldolgozást, ezért ennek automatizálása elengedhetetlen.
Manapság előszeretettel használnak gépi tanulást problémák automatizált megoldására, ezért a megoldásomban is ezt fogom használni. Ez a módszer egyrészt lehetőséget biztosít az anomáliák detektálására, másrészt egy tanuló folyamatot is, amely képes folyamatosan fejlődni és egyre pontosabb eredményeket prezentálni.
Ezek alapján a TDK dolgozatomban arra vállalkozok, hogy kétfajta megoldást mutassak be az említett problémára:
Az első megoldásom a DeepLog alapú algoritmusból indul ki. Ez egy nagy irodalommal rendelkező és sokak által referenciaként kezelt algoritmus, amely függetlenül a számos gyengeségétől, hatékonynak bizonyult. Ennek az algoritmusnak az implementálására vállalkozom.
A második megközelítés egy új, KNN alapú eljárás, amely csoportokba rendezhető logfájlok esetén képes anomáliák azonosítására. Ennek az eljárásnak lényegesen szűkebb az irodalma, alig publikáltak róla, különösen a csoportokba rendezhető logfájlok esetén. A KNN alapú megközelítések egyik kulcseleme a megfelelő távolságmetrika megtalálása. A logsor-szekvenciák közötti távolság meghatározására többfajta lehetőséget megvizsgálok, amelyek teljesítményét hasonlítom össze egymással és a mélytanulás alapú megoldással.

Abstract

We live in a world of data. A lot of data is generated in our environment from sensors, social media and many other sources. Without exception, complex IT systems have log files that give us a more accurate picture of the operation, status and state of systems. These data carry a huge amount of information that can be of great value to the operators. There are cases where anomalies are detected in the log data. We speak of an anomaly when we receive a log series of unknown composition or sequence. A very important property of log series is that they usually have a repeating pattern upon which my solution is based on.

My task in detecting anomalies is to indicate unusual logline sequences. Since systems are constantly logging, a huge amount of data is being generated which makes manual processing very circumstantial causing an essential need to make automated solution.

Nowadays, machine learning is commonly used to solve problems in an automated way, which I will also use. This method provides both a way to detect anomalies and a learning process that is able to continuously improve and present increasingly accurate results.

In my TDK thesis I will present two solutions to the problem mentioned above:

My first solution starts from the DeepLog based algorithm. This is a popular and well known algorithm which has proven to effective regardless it's weaknessis.

The second approach is a new KNN (K-Nearest Neighbors)-based method that can identify anomalies in clustered log files. The literature on this method is considerably scarce, with hardly any published work, especially for grouped log files. One of the key elements of KNN-based approaches is to find a suitable distance metric. To determine the distance between the log-sequences, I consider several options, comparing their performance with each other and with the deep learning-based solution.

Tartalomjegyzék

Absztrakt 2

Abstract 3

1. Bevezetés 6

2. A munkám során felhasznált ismeretek és a meglévő megoldások 8

2.1. Logfájlok 8

2.2. Anomáliák, anomália detektálás 9

2.3. Gép tanulás 9

3. Jelenlegi eljárások és alkalmazott algoritmusok 11

3.1. Bevezetés 11

3.2. Gépi tanulás típusai 11

3.2.1. Felügyelt tanulás 11

3.2.2. Nem Felügyelt tanulás 12

3.2.3, Félig Felügyelt tanulás 13

3.3. Meglévő algoritmusok, publikált módszerek 14

3.3.1 Invariant Mining 14

3.3.2. PCA-based 14

3.4. Mélytanulás 15

3.4.1. Általánosan a neurális hálókról 15

3.4.2. LSTM-ről általánosan 16

4. DeepLog alapú megoldás log anomáliák detektálására 17

4.1. Log Parsing 19

4.2. Csoportosítás 20

4.2.1. Példa a modell működésére 21

4.3. Tanítás a tanító adathalmazzal 23

4.3.1. Embedding Layer 23

4.3.2. LSTM layer 24

4.3.3. Dense layer + compile + fit 25

4.4. Kimenet előrejelzése validációs adathalmazzal 27

4.5. Kimenet előrejelzése a tesztadathalmazzal 29

5. KNN algoritmus 29
5.1. Bevezető 29
5.2. KNN algoritmus általánosan 30
5.3. Általános KNN munkamenet 31
5.3.1. DeepLog által előállított adatok 31
5.3.2. Az algoritmusban használt paraméterek, metrikák, küszöbértékek 32
5.3.3. KNN tesztelése 38
6. HDFS, mint címkézett logfájl ismertetése 39
6.1. HDFS, mint címkézett logfájl 39
6.2. HDFS adatállomány felosztása 39
6.3. A HDFS-ben megtalálható template-k és azok gyakorisága 40
6.4. Szekvenciák hossza 41
7. Értékelés 43
7.1. Általánosan az értékelés folyamatáról 43
7.2. Összehasonlító metrika 44
7.3. DeepLog értékelés 45
7.3.1. Pontosság 45
7.4. KNN értékelés 46
7.4.1. Pontosság 46
8. Konklúzió: 48
9. Irodalomjegyzék 49

1. Bevezetés

A 21. század a digitalizáció évszázada. Gyorsan jelennek meg újabb és újabb megoldások különböző minket érintő problémákra. Ezen megoldások sikerességének alapja a megszerzett adat és annak helyes használata. Kétségtelen, hogy a minket körülvevő hatalmas adatmennyiséget el kell tárolni és fel kell használni a megoldásainkban. A komplex IT rendszerek kihívásai nem csak fejlesztésekből és tesztelésekből állnak, hanem fontos szerep jut az üzemeltetésre és a működtetésre egyaránt. A működés során a komplexitás kezelése szinte lehetetlen manuális eszközökkel, ezért erre gyakran különböző automatizmusok, algoritmusok segítségét vesszük igénybe. A rendszer működése során előforduló mindenfajta eseményről maga a rendszer információt szolgáltat nekünk logfájlok formájában. Ezek a logfájlok keletkezhetnek operációs rendszer, adatbázis, alkalmazás és hálózati réteg szintjén egyaránt. Minél komplexebb a rendszer annál pontosabb információra van szükségünk az előforduló hibákról, illetve azok előfordulásának helyéről, így ezek a fájlok hatalmas méretűvé válhatnak. Így beszélhetünk a logfájlok tartalmát tekintve különböző anomáliákról. Tehát ezek a fájlok ellentmondásokat is tartalmazhatnak, ami kellemetlen következményekkel járhatnak. Ezeknek az anomáliáknak a detektálására számos eszköz létezik. Kezdetben a humán erőforrás alkalmazása volt az egyetlen eszköz a sok különböző anomália detektálására, ami az utóbbi időben átalakult és a gépi tanulásnak köszönhetően lehetőség nyílt anomália detektálásra gépi tanulás segítségével. A nagyméretű logfájlok analíziséhez és anomáliák detektálására hatalmas humán erőforrás szükséges. Számos cégóriás dolgozik az algoritmusok optimalizálásán, hiszen felismerték annak a tényét, hogy mekkora potenciál lehet a loganalitikában. (pl. One Identity cég) Az algoritmusok helyessége és hatékonysága komoly kihívást jelent a kutatóknak és annak hatékonyabbá tétele nagy értékkel bír. A TDK dolgozatomban én is egy ilyen feladatra vállalkoztam nevezetesen, hogy hogyan lehet a mesterséges intelligencia lehetőségeit felhasználni a logfájlok-beli anomáliák detektálására. Ennek a feladatnak az elvégzéséhez számos meglévő algoritmust vizsgáltam és arra törekszem, hogy azoknak a hatékonyságát növeljem, valamint egy új, eddigiektől eltérő elven működő eljárást javasoljak.

A munkámat két részre bontottam: az első részben a loganomáliák detektálásánál a sokak által használt, mélytanulás alapú, DeepLog algoritmusnak egy saját implementációját alkottam meg néhány újítással. Ezt az implementációt referenciaként fogom használni a másik, általam kifejlesztett algoritmus bemutatásához. A második részben ismertetem ezt az új, általam kifejleszett K Nearest Neighbor (KNN) eljáráson alapuló algoritmust, ami egy jóval kisebb irodalommal rendelkező megközelítést használ loganalitika terén. A jelenleg létező KNN alapú eljárások csak egyes logsorok anomália voltát képesek kimutatni, a teljes szekvenciákét nem, ezzel ellentétben én egy szekvenciális-anomáliákat detektáló megoldást fogok bemutatni.

A dolgozatomat 9 fejezetre bontottam, a bevezetés után a 2. fejezetben ismertetem a munkámhoz szükséges elemeket. Itt ismertetem a logfájlok szerepét, az anomáliadetektálás lényegét, valamint a gépitanulás általam használt eszköztárát. A 3. fejezetben kifejtésre kerülnek az ismert algoritmusok és technológiák. Ennek kifejtése azért fontos, hogy értékelni

és pozícionálni lehessen a saját munkámat. A 4. és az 5. fejezet a saját munkámról szól, itt kifejtem az implementált algoritmusok működését és a saját megoldásaim lényegét. A 6. fejezetben bemutatom azt a logfájlt, amit a megoldásomhoz felhasználok, majd a 7. fejezetben az egyes algoritmusokat kiértékelem és összehasonlítom a kapott eredményeimet a meglévő eredményekkel. A dolgozatot egy konklúzióval foglalom össze és az Irodalomjegyzékkel zárom le.

2. A munkám során felhasznált ismeretek és a meglévő megoldások

2.1. Logfájlok

Az információs és az operációs rendszerek, illetve azok egyes részei hatalmas mennyiségű bejegyzést tárolnak el futásuk során logfájlokban. Ezek a fájlok jelentős információt hordoznak, melyek alapján bizonyos hibák felbukkanását előre meg tudjuk jósolni. Például, ha a rendszer összeomlik vagy váratlan viselkedést produkál, annak bizonyára nyoma akad a logfájlokban egyaránt.
Manapság annyi adat keletkezik, hogy ezek a logfájlok akár hatalmas méretűre is duzzadhatnak. Ha több adat keletkezik, több mintát fedezhetünk fel, több tudást szerezhetünk. Viszont miután az idősorelemző algoritmusoknak teljes egészében végig kell olvasniuk a fájlt, egyre több és több időt és erőforrást emészt fel az adatmennyiség növekedése.
A számítógépes rendszerek általánosan magukról szolgáltatnak adatokat logfájlok formájában. Ezek szöveges, ember számára könnyen értelmezhető formátumban, sorokban generálódnak. A logfájlok nagyon értékes információt hordoznak magukban, amelyek jellemzik a rendszer pillanatnyi állapotát, ezzel jelezve az üzemeltetőnek a lehetséges problémákat és veszélyforrásokat. Gyakori, hogy ezek a problémák csak a logfájlok analízisével mutathatók ki.

A logfájlok feldolgozásának legismertebb módja a fájl sorról sorra történő feldolgozás. A sorok elemzése közben megfigyelhető egy minta, ami jellemzi őket. Egy másik ismert és komplexebb technika, ami a sorok mellett az azt körülvevő kontextust is figyelembe veszi. A logelemzésre sok eszköz született. Annyira fontos ez a terület, hogy például Magyarországon is több külön cég jött létre többek között a logelemzési területre, ezek közül az egyik leghíresebb a Balabit Kft, melynek logelemzési algoritmusai a mai napig világvezetőnek tekinthető.
Az előbb említettek szerint a generált logfájlok alapos és pontos megfigyelése elengedhetetlen, hiszen bármely veszélyforrást előrejelző logsor figyelmen kívül hagyása kiszámíthatatlan károkat tud okozni a rendszerben. Ezen okokból a loganalitika manuálisan, ember számára közel lehetetlen, amely egy automatizált problémamegoldást indikál.

2.2. Anomáliák, anomália detektálás

Az anomália detektálás alatt azon elemek, bejegyzések, adatok, események, illetve megfigyelések azonosítását értjük, amelyek nem felelnek meg valamely, az adatsorra illeszthető mintának *[1]*.
Az anomáliák másik irányból megközelítve szokatlan vagy kiugró értékek. Azok lehetnek hibák, szándékos támadásból fakadó értékek vagy szimplán eddig még nem látott események következménye. Azoknak a szokatlan értékeknek a megtalálása, azonosítása, kritikus lehet a rendszerek helyes működése szempontjából. Anomália detektálással találkozunk a hálózatok üzemeltetésénél, IT rendszerek működése során, banki rendszerekben, egészségügyi adatokban és számos egyéb alkalmazásban. A feladat: minél pontosabban és gyorsabban azonosítsuk ezeket az anomáliákat.
Logfájlok esetében az anomáliákat 3 nagy csoportba tudjuk osztani:

1) Attribútum-anomáliák:
   Attribútum-anomáliáról akkor beszélünk, amikor egy normálisnak tűnő logsor olyan attribútumokat tartalmaz, amelyek hibás működést valószínűsítenek. Tehát maga a logsor helyénvaló, azonban a numerikus attribútumai nem azok.
2) Logsor-anomália:
   Olyan anomália, ami a logsorok számosságára vonatkozik. Ha egy logsor ritkán jelenik meg, akkor feltételezhetjük, hogy anomáliáról beszélünk.
3) Szekvenciális-anomáliák:
   Ezek alatt olyan anomáliát értünk, amikor nem egy adott logsor, hanem egy logszekvencia, azaz logsorok egymás utánisága mutat szokatlan viselkedést és ezeket próbáljuk detektálni. Ez kevésbé használt keresési alap, azonban a dolgozatomban én ennek a detektálására vállalkozom és mutatok be algoritmusokat.

Anomáliák előfordulása a logsorok előfordulásához képest kifejezetten alacsony, azonban általában ezek tartalmaznak a problémaelhárítás szempontjából releváns információt, így ezek pontos és gyors detektálása rendkívül fontos.

2.3. Gép tanulás

"A gépi tanulás fogalmához kapcsolódik az MI kialakulása. Onnantól kezdve, hogy egy gép képes az új, tetszőleges ismeretek elsajátítására és megfelelő felhasználására már beszélhetünk intelligens rendszerekről. Ennek a területnek a kutatása az 1957-ben megalkotott perceptronnal vette kezdetét. A perceptron segítségével képesek voltunk olyan gépi programokat megalkotni, melyek egy bizonyos feladathoz tartozó problémához kapcsolódó néhány megoldásból tudtak általánosításokat végrehajtani. Az általánosítás segítségével pedig következtetni az ismeretlen részekre. Azokat a rendszereket melyek ilyen feladatokat meg tudnak oldani neurális hálóknak nevezték, és több fajtájuk is létezik." [2]

A gépi tanulás a mesterséges intelligenciának egy olyan részhalmaza, ahol a számítógépek a kapott bemenetekben mintákat keresnek és ezeket a mintákat felhasználva predikcióra alkalmasak. Minél több a bemeneti adat, annál nagyobb átfedéssel és pontossággal lesz képes a modell előrejelezni. Célja, hogy az olyan problémákat, amelyeket az embereknek manuálisan kellett korábban megoldania, sikerüljön automatizáltan, humán erőforrás felhasználását minimalizálva megoldani. Ilyen probléma közé lehet sorolni az önvezetés problémáját is. Amikor vezetünk tulajdonképpen feltérképezzük az előttünk található teret és a rajta elhelyezkedő objektumokat (élőlényeket, tárgyakat, körülményeket stb..) és ennek az ismeretnek a birtokában hozunk meg bizonyos döntéseket. Az önvezetés mögötti ötlet hasolnó, különböző szenzorokkal feltérképezzük a környezetet és a korábbi ismereteink alapján (azok, amelyeket betanítottunk a számítógép számára) a lehető legoptimálisabb döntést hozzuk. *[3]*
Egy kissé eltérő, de nagyon is fontos áttöréseket jelentett a gépi tanulás a különböző osztályozási feladatok megoldásában. Ilyenek közé lehet sorolni az Email osztályozást és a spamszűrést. A korábbiakhoz hasonlóan itt is számos paraméter figyelembevételével lehet eldönteni, hogy az adott email milyen kategóriába tartozik és tartalmaz-e valamilyen kártékony vírust. [4]

3. Jelenlegi eljárások és alkalmazott algoritmusok

3.1. Bevezetés

Ahogy a bevezetőben is említettem a gépi tanulás loganalitika területén hatalmas sikert aratott és a mai napig napról napra fejlődik. A 3. fejezet elején bemutatom röviden az elterjedt megoldási opciókat log anomália detektálására. Majd bemutatom a gépi tanulás 3 nagy csoportját, amely legmagasabb szinten különbséget jelent a megoldás megközelítésének irányából. Majd bemutatom ezeknek az előnyeit és hátrányait egymással szemben.

3.2. Gépi tanulás típusai

Ebben a fejezetben sorra veszem a különböző tanítási módszereket a gépi tanulás terén, hogy egy helyen legyen minden olyan fogalom, amit felhasználok a dolgozatomban.

3.2.1. Felügyelt tanulás

Felügyelt tanulás esetén a modellt olyan training (tanító) adatokkal tanítjuk, amely megfelelően van címkézve, így ismert az elvárt kimenet. A tanítás folyamata addig történik, amíg a modell magától képes felismerni a kapcsolatot a beadott bemenet és a kimenet között és azt nagy hatásfokkal reprodukálni is tudja (Természetesen a nagy hatásfok szubjektív, alkalmazástól függ). Miután ez teljesült, a modell kap egy teszt bemenetet, ami csak az inputot tartalmazza és annak a korábban megtanult módszer szerint meg kell jósolnia a kimenetet (ami rendelkezésünkre áll). Ennek eredménye összehasonlítható a meglévő „helyes” kimenettel és így megállapítható a hatásfok.

Logfájlokban található anomáliák detektálására felügyelt tanulással a klasszifikációs technikák a leginkább alkalmazottak. Klasszifikáció alatt a loganalitika területén azt értjük, hogy előre definiált osztályok szerint csoportosítjuk a logsorokat. Mivel ismert a bemenet és a kimenet így a rendszer könnyen tesztelhető. Jelen esetben bináris klasszifikációról beszélünk, hiszen a kimenet bináris (vagy anomáliás vagy normális az adott szekvencia). Jelen dolgozatomban ezzel a megközelítéssel nem foglalkozom, mivel a gyakorlatban szinte sosem áll rendelkezésre címkézett logadat. [5]
Loganalitikában a felügyelt tanulásos algoritmusok előnyei és hátrányai a többivel szemben:

Előnyök	Hátrányok
• *Nagyobb pontosság a többi tanulási módhoz képest, hiszen definiált a bemenet és az elvárt kimenet*	• Fontos a tanító adathalmaz minősége, ami nagy és megterhelő manuális munkát vár
• *Meglévő és megbízható tanítóadathalmaz számos esetben rendelkezésünkre áll, így ez olcsóbbá válhat a tanítási erőforrás ráfordítás szempontjából a többi eljárással szemben.*	• Való világban történő alkalmazáshoz adaptálódni kell a körülményekhez, ami egy előre definiált osztályozás esetében nehezen kivitelezhető
	• Könnyen előáll a túltanulás (overfitting) jelensége, ami pontatlansághoz vezet
	• Előállhat az az eset is, hogy a modell a hibákat tanulja meg a normális viselkedés helyett és emiatt az ismeretlen anomáliát nem fogja felismerni

3.2.2. Nem Felügyelt tanulás

Nem Felügyelt tanulás esetén a gépet olyan training adatokkal tanítjuk, amely nincs (vagy nem megfelelően van) megcímkézve, így nem ismert az elvárt kimenet. A cél, hogy a megfelelő mintákat észrevegyük a bejövő adathalmazban, aminek a segítségével létre tudunk hozni valamilyen csoportosítást.

Loganalitkában a csoportosítás (clustering) irány a leghasználtabb. Ennek lényege, hogy az algoritmus maga keres hasonlóságokat a megadott (jelen esetben) sorok között és ezek alapján hozza létre a csoportokat. Ezeket a csoportokat megtanulja a rendszer és az újonnan bejövő logsort megpróbálja besorolni az egyik ilyen generált csoportba. Ha be tudja sorolni, akkor normális logsorról beszélünk, ha nem, akkor anomáliáról. Ezzel a megközelítéssel sem foglalkozom a dolgozatomban [6].

Előnyök	Hátrányok
• *Nem igényel manuális erőforrást a címkézéshez meg az előkészítéshez, mert nem szükséges*	• Nagy pontatlanság is kialakulhat, ha nem lesz manuális validáció a kapott eredményről
• *Való világban történő alkalmazáshoz kiváló, hiszen magas az adaptációs készsége: A gyakorlatban nincs előredefiniált osztályozás*	

3.2.3, Félig Felügyelt tanulás

„Semi-supervised learning (SSL) is halfway between supervised and unsupervised learning. In addition to unlabeled data, the algorithm is provided with some super- vision information – but not necessarily for all examples. Often, this information will be the targets associated with some of the examples.” [7]

Félig Felügyelt tanulás esetén a gépet a tanítás során tanítjuk címkézett és nemcímkézett training adathalmazokkal is. A cél, hogy megtanulja a gép, hogy hogyan lehet statikus címkézett vagy nemcímkézett tanulásról egy sokkal dinamikusabb tanulásra átállni, ami kombinálja a korábban említett két stílust ezzel egy új aspektust hozzáadva a tanuláshoz.

A loganalitikában semi-supervised-nak hívják azt az esetet is, amikor csak normális log szekvenciákkal tanítunk egy modellt. Ez tulajdonképpen felügyelt megközelítés (mivel kellenek hozzá címkék), de így a modell biztosan csak a normális viselkedést tanulja meg, és minden attól való eltérésre rá tud mutatni. Dolgozatomban ennek a megközelítésnek nyilvánítok nagy szerepet és jelentőséget.

3.3. Meglévő algoritmusok, publikált módszerek

Az alábbiaknak a log anomália detektálására született megoldásokat fogom röviden bemutatni, amelyek a mai napig jelentősek és nagy felhasználótáborral rendelkeznek:

3.3.1 Invariant Mining

Az *Invariant mining* alapvető célja, hogy a logsorok bizonyos logika mentén történő csoportosítása után ezekben a csoportokban keres olyan invariánsokat, amelyek a csoportot jellemzik. Ilyen invariáns keresés az alábbi cikkben olvasható: *[8]*

Olvasható, hogy a megoldásukban a csoportosítás után megvizsgálják az adott csoportban található összes logsor típusra, hogy mekkora a számossága. Ezekből alkotott vektor jellemzi az adott csoportot és ezt minden csoportra elvégezve egy olyan metrikát kapunk, amellyel képesek vagyunk összehasonlítani a csoportokat.

Miután megalkottuk a megfelelő vektorokat elvégezzük az ebből alkotott mátrixon az *SVD (singular value decomposition)* faktorizációt. Ennek segítségével elő tudunk állítani nemnégyzetes mátrixnak is egy olyan mérőszámot, mint a négyzetes mátrixoknál a sajátérték. Ezt jelen esetben *szinguláris értéknek* hívják. *[9]*. Ennek segítségével közvetlenül mérhető a korreláció a logsorok és az invariánsok között.

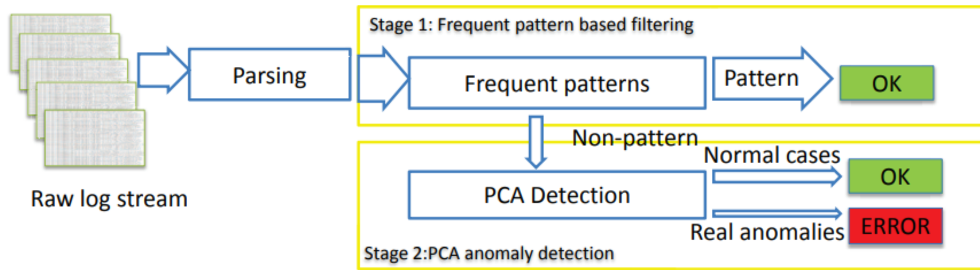
3.3.2. PCA-based

„Principal component analysis is primarily a dimensionality reduction technique. It works by identifying the principal components. Principal Components are independent feature vectors also called Eigenvectors of a given data which explain the maximum variance in the data. Each PC is a linear combination of existing correlated features and lies orthogonal to other eigenvectors. Using PCA we can reduce the number of feature vectors without losing information value. Let us see how we can find PCs for the IRIS dataset.” *[10]*

A PCA (Principal component analysis) technika a nevéből adódóan bizonyos részekre bontja az adatmátrixot (amelynek sorai a tanulómintákat, az oszlopai pedig az adott típusú logsor számosságát jelöli) és ezeket nevezzük „principal component” - nek. Ezek a component-ek (komponensek) tulajdonképpen vektorként kezelhetőek.

Anomália detektálásához a részekre bontott elemeket megpróbáljuk újra összerakni és ezzel rekonstruálni az eredeti mátrixot. Amennyiben ez egy bizonyos pontossággal teljesül, akkor normálisnak, ha nem, akkor anomáliásnak tekintjük az elemet. A rekonstruált adat nem fog tudni teljesen megegyezni az eredetivel, azonban a helyes küszöbérték megválasztásával az eredményesség növelhető.

Ezt az eljárást általában nemfelügyelt tanulásos algoritmusok használják és alapvetően képek feldolgozásánál elterjedt, de loganalitikába is kiválóan használható.



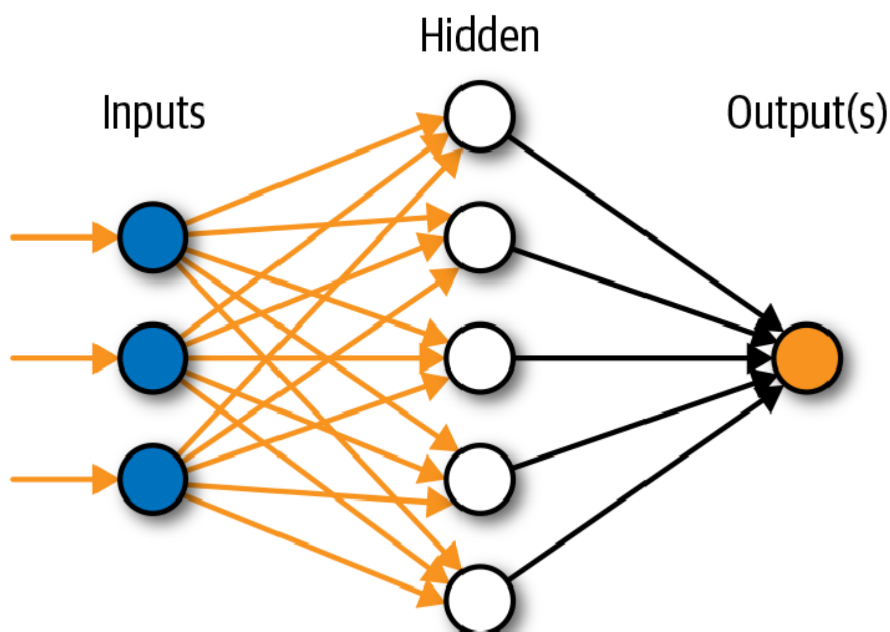
1. ábra

Az 1. ábra [10] a PCA használatát mutatja be. Ebben az esetben egy kétlépéses ellenőrzést végző modell látható, amely PCA-t használ az olyan komponenseknél, ahol nincs szokásos minta észlelve. Ennek értelmében egy még pontosabb eredményt kaphatunk és ki tudjuk szűrni vele a zaj által torzított adott mintával rendelkező sorokat, amelyek nem anomáliások.

3.4. Mélytanulás

3.4.1. Általánosan a neurális hálókról

Egy előszeretettel használt Mesterséges Intelligencia technika a mélytanulás. Ennek alapja az emberi tanulási folyamatok szimulációja és hasonló módon hagyjuk, hogy a gép saját maga tanuljon a kapott adatokból. Az új ismeretek esetén ahogy az agyunk neuronokat termel, amelyek között szinapszisok alakulnak ki, úgy a neurális hálók is csomópontokból és élekből állnak. A neurális háló egy gráfként képzelhető el, amelynek szintjeit (csomópontok halmazát) úgynevezett rétegnek hívnak. Ezek a rétegek csomópontokat tartalmaznak, amelyek össze vannak kapcsolva a velük szomszédos rétegek csomópontjaival. Minden csomópont tartalmaz egy súlyt, ami meghatározza, hogy mekkora hatása legyen az ott generált értéknek a következő rétegre. Az utolsó szint határozza meg a kimenetet a korábbi eredményekből.



2. ábra

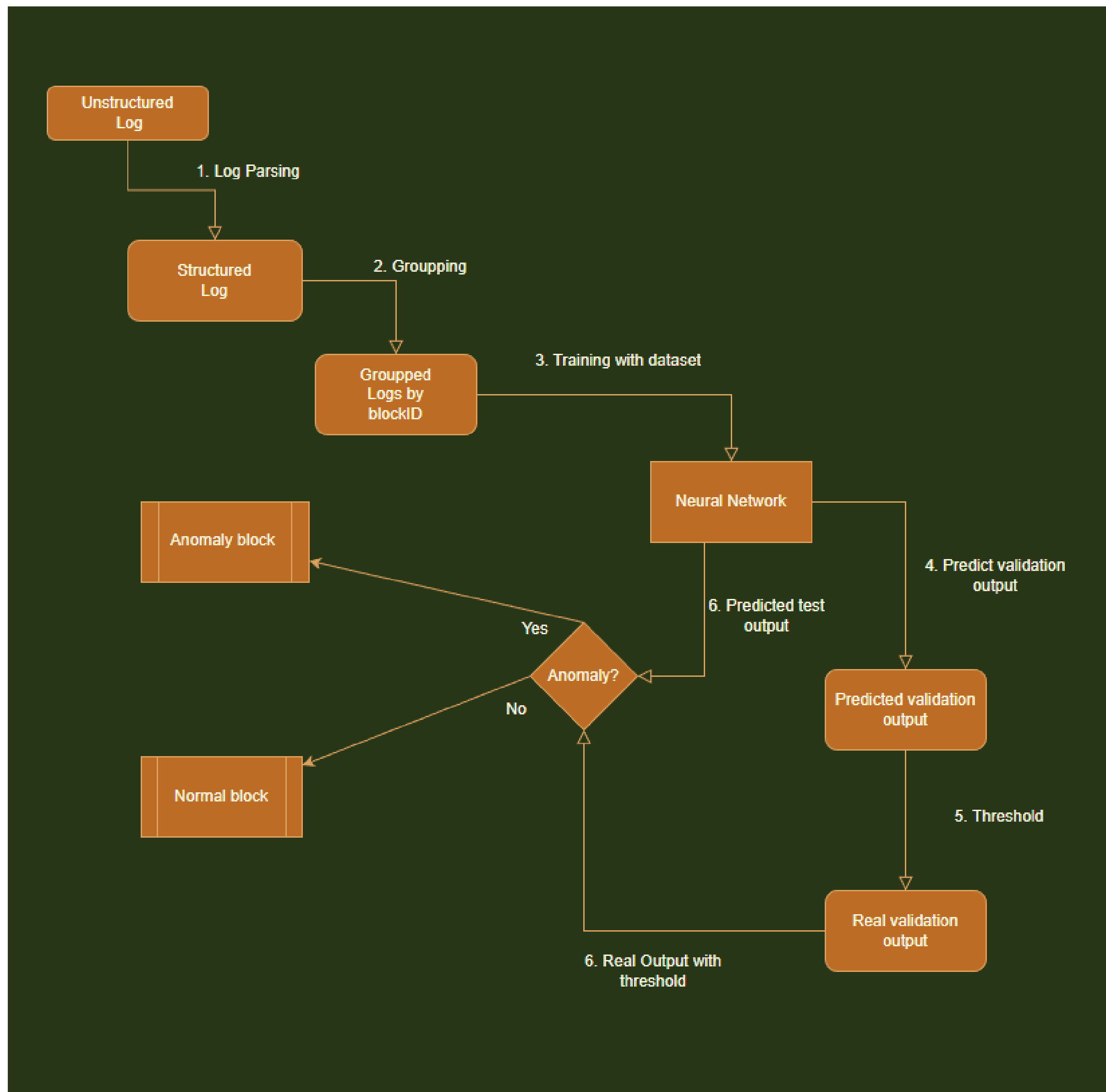
Ahogy a 2. ábrán is látszik, a bemeneti szinten lévő csomópontok száma megegyezik a bemenetek számával és a kimeneti réteg csomópontjai pedig a kimeneti réteg számával. Az összes köztes réteget rejtett rétegnek nevezünk, hiszen azok „csak a számolást” végzik, nem lépnek interakcióba a külvilággal.

3.4.2. LSTM-ről általánosan

Megoldásomban a Deep Learning szekvenciális problémákra alkalmazott egyik legismertebb módszerét az LSTM-et (Long short-term memory) [11] használom. Az LSTM egy olyan RNN változat, aminek alapja a visszacsatolás szerinti tanulás. Egy adott lefutás alatt folyamatos visszacsatolás történik a kapott eredményről ezzel effektívebbé téve a tanulás folyamatát. Amíg az RNN ebben nyújtja az újdonságot, addig az LSTM képes tárolni az adatokat:
Tartalmaz egy rövid meg egy hosszútávú memóriát (Long short-term memory), innen is kapta a nevét.
Loganomáliák detektáláshoz is kiválónak bizonyult ez az ötlet is, így ennek következményeképp megjelentek különböző megoldások, amelyek közül az egyik legismertebb az LSTM alapú **DeepLog** lett, melyet a 4. fejezetben mutatok be.

4. DeepLog alapú megoldás log anomáliák detektálására

A DeepLog implementációm jobb megértése érdekében létrehoztam a következő ábrán látható munkamenetet (workflow):



3. ábra: DeepLog workflow

Kezdetben kifejtem a munkamenet egyes részeit röviden, utána részletesen, olyan sorrendben, ahogy a munkamenet diagramon (3.ábra) szerepel.

Log Parsing

A logfájlok alapvetően strukturálatlanok, így azokon a műveletvégzés nagyon nehézkes. Ahhoz, hogy elhárítsuk ezt a nehézséget elvégzünk egy átalakítást. Egy rendezetlen, strukturáltalan szöveges fájlból, egy rendezett, strukturált, a számítógép számára feldolgozhatóan tagolt fájlt generálunk.

Csoportosítás

A csoportosítás során megkeressük az összetartozó logsorokat. A benchmark céllal gyakran használt HDFS logban (Hadoop Distributed File System, melynek részleteit 6. fejezetben ismertetem) az azonos blokkra vonatkozó logsorok képezik ezeket a csoportokat. Ezáltal megkapjuk azokat a szekvenciákat, amelyeket vizsgálni fogunk.

Tanítás a tanító adathalmazzal

Miután megtörtént a csoportosítás elkezdjük tanítani a meglévő bemenettel a neurális hálót. Ennek eredményeképpen előállítunk egy betanított neurális hálót, amelynek feladata a korábbi logsorok alapján a következő logsor helyes predikciója.

Kimenet előrejelzése a validációs adathalmazzal

Miután elvégeztük a tanítást, előkészítjük a validációs adathalmazt és azzal létrehozunk egy predikciót, mely a korábbi logsorok alapján előrejelzi a következőt. Ha az előrejelzés nem elég sikeres, akkor anomáliát jelzünk, hiszen ilyen mintát a modell nem sokat látott a tanulás során.

Küszöbérték

A predikciót összehasonlítva a validációs adathalmaz kimenetével egy kiértékelési metrika segítségével megállapítjuk, hogy mekkora küszöböt (threshold) válasszunk, ami maximalizálja az eltalált anomáliák számát.

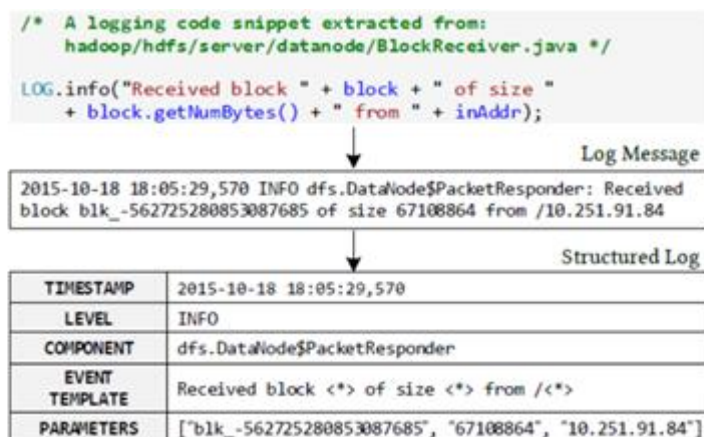
Kimenet előrejelzése a tesztadathalmazzal

Ha sikerült megállapítani a threshold-ot, megnézzük a tesztadathalmazra a predikciót majd összehasonlítjuk a tesztadathalmazhoz tartozó kimenettel. A threshold segítségével így megkapjuk, hogy milyen pontosan tudtunk előrejelezni.

A továbbiakban az eljárás főbb lépéseit írom le, egyesével, részletesebben:

4.1. Log Parsing

Ahogy azt a leírásban említettem, az első lépés a Log Parsing. Az eljárás viselkedését a később részletesen bemutatott HDFS adatsoron keresztül mutatom be.
A következő példában vizsgáljunk meg egy ilyen tipikus HDFS logsort:



4. ábra: LogParser működése

A 4. ábrán *[12]* jól látható, hogy hogyan is néz ki egy általános logsor: Az elején egy időbélyeg található, ami jelzi a kibocsátás időpontját. Ezt követi egy információs címke, ami jelzi, hogy az adott logsornak pontosan mi is a szerepe. Jelen esetben ez egy informatív logsor, ezt jelzi az INFO érték a „LEVEL” mezőben. Majd ezután látható egy „komponens” (COMPONENT) mező, ami a logsor funkcióját fejezi ki, hogy ez jelen esetben egy PacketResponder. Az „Event TEMPLATE” mező követi a COMPONENT-et, ami egy általános visszajelzés az adott sorról, ami megmutatja, hogy ez milyen sablont követ: A szöveg megtalálható a logsorban és ahol <*> van, ott valamilyen paraméter érkezik. A template-eket a template parser állítja elő, a logsorok gyakran változó részeinek <*>-ra cserélésével. A PARAMETERS-ben pedig fel van sorolva megfelelő helyiértékkel, hogy milyen értékek érkeztek a logsorba a <*> helyére. Ezek közül számunkra az első paraméter lesz a releváns, ami a blockazonosítót tartalmazza. A block meghatározza, hogy milyen template-tel rendelkező logsorról beszélünk (az azonos block-hoz tartozó logsorok azonos template-tel rendelkeznek)
Látható, hogy egy ömlesztett 2 soros logsorból milyen szépen lehet strukturált adatállományt létrehozni Parsing segítségével. Számos ilyen Parsing algoritmus létezik, ilyen például:

- *IPLoM: [13]*
- *LKE: [14]*
- *Spell: [15]*
- ***Drain*** [16]

Ezek közül én a *Drain*-t használom, amely az egyik legelterjedtebb algoritmus. Miután lefuttattam a HDFS-en ezt az algoritmust, két fájlt kaptam eredményül a kijelölt könyvtárban:

1) **HDFS_structured.csv**

Ez egy .csv fájl, ami tartalmazza strukturáltan oszlopos elrendezésben egymás alatt a logsorokat és a fejrészben a korábbiakban definiálthoz hasonló típusok szerepelnek. Ezekből számunkra 3 oszlop lesz releváns:

- **LineID**, ami egy autoinkrementális azonosító, hogy meg tudjuk különböztetni a sorokat
- **EventID**, ami definiálja, hogy az adott sor milyen Sablonnak felel meg
- **ParamterList,** ami több elemből is álló lista, azonban mindegyik tartalmaz egy blockazonosítót

Így ezt a 3 oszlopot megtartjuk a többit pedig eldobjuk és a jelenlegi algoritmusunkban ezekkel a paraméterekkel nem súlyozzuk a rendszert. Természetesen ezzel pontatlanabb modellt kapunk, azonban a többi olyan kis súllyal járulna hozzá az eredményhez és annyival lassítja a folyamatot, hogy jelen esetben én nem használtam fel őket, azonban valós rendszerek esetén célszerű lehet megfontolni a használatukat.

2) **HDFS_templates.cs**

Ez egy olyan .csv kiterjesztésű fájl, ami tartalmazza az EventID-t, tehát az azonosítót, ami azonosítja az adott template-tel rendelkező logsort. Emellett tartalmazza a hozzá tartozó template-t és ezeknek az előfordulását. Ez a fájl rendkívül hasznos lesz számomra, hiszen itt van felsorolva az összes lehetséges template és annak azonosítója. Ezalapján minden azonosítót le tudok cserélni egy saját, egyszerűbb azonosítóra (egy 1-től kezdődő egész számra), amit majd azonosításra tudok használni. (Lásd: Groupping fejezet)

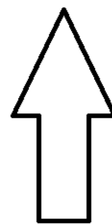
4.2. Csoportosítás

Miután megtörtént a logfájl strukturálása és megtartottuk azokat az oszlopokat, amelyeket figyelembe fogunk venni az analitika során, következhet a csoportosítás. Ennek első szakaszában meghatároztam, hogy melyek azok a logsorok, amelyek egy adott csoportba tartoznak. Ez a csoportosítás HDFS esetében a block-azonosító mentén történt, így kaptam egy olyan dictionary-t, ami tartalmazza az adott azonosítót kulcsként és a hozzá tartozó EventTemplate-k azonosítóit egy listában felsorolva értékként. Fontos, hogy az EventTemplate-k nem a generált azonosítójukkal szerepelnek itt a dictionary-ben, hanem generáltam nekik egy új egész számot, mint azonosító, hiszen a generált EventTemplate azonosítók strukturáltalanok, amelyek nem használhatóak a modellben.
Miután ez a szintű csoportosítás megtörtént és egységesen kezelem az összes azonos blokkazonosítóhoz tartozó Eventeket, megkezdődhet a training (tanító) adathalmaz megalkotása. A training adathalmaz létrehozásánál fontos szempont, hogy a korábban bemutatott LSTM modellt fogjuk használni (Lásd következő alfejezet)

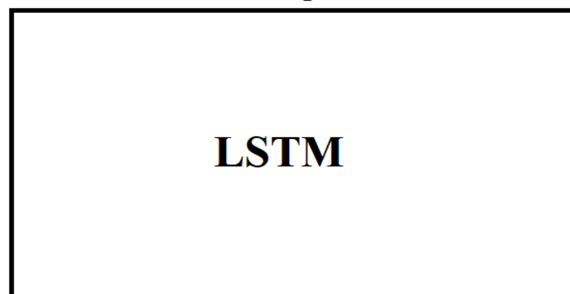
4.2.1. Példa a modell működésére

Mivel a csoportosítással az adott blockhoz tartozó saját azonosítókat létrehoztuk észrevehetjük, hogy ezeknek a hossza eltérhet: Lehet olyan block, amihez három Event tartozik, de egy másikhoz akár harminc. Ennek egységesítése érdekében (hogy az LSTM számára azonos bemeneti méretű listákat adjunk), kiegészítettem 50-re a hosszukat és feltöltöttem 0-val a kimaradt helyeket. (50 az egy „magic number”, az ennél hosszabb szekvenciákat anomáliának detektálom és nem foglalkozom vele lásd HDFS rész). A visszacsatolásos tanuláshoz fontos, hogy folyamatosan visszajelzést adjunk a prediktált eredmény értékéről. Ezért az eredményhalmaz (Y) meg fog egyezni a bemeneti halmazzal (X), annyi különbséggel, hogy egy értékkel el lesz tolva jobbra, hiszen ekkor leszünk képesek a korábbi bemenetek segítségével az azt követő elem megjósolására (Lásd 5. ábra).

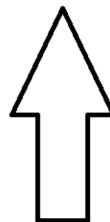
Y = 1, 1, 1, 2, 3, 4, 3, 4, 3, 4, 5, 5, 5, 49,0,...........,0



Output



X = 49, 1, 1, 1, 2, 3, 4, 3, 4, 3, 4, 5, 5, 5, 0,.........,0



Input

5. ábra

Az 5. ábra úgy értelmezendő, hogy az X bemeneti halmaz egyes elemeit megadjuk az LSTM-nek és elvárjuk, hogy megjósolja a következő elemet. Az első bemeneti elem a példánkban a 49-es (később látni fogjuk, hogy miért) és elvárjuk a modelltől, hogy minél nagyobb valószínűséggel jósolja meg a 1-es kimeneti értéket. Belátható, hogy a maximális tanulás elérése érdekében minden elemet meg kell próbálni megjósolni és azt a tudást felhasználva pontosabb eredményt kaphatunk a későbbi predikciók során. Alapvetően az X első elemének predikciója nem történik meg, hiszen azt kapjuk bemenetként, amelynek a függvényében a következő elemet jósoljuk meg.
Hasonló helyzet áll elő akkor, amikor már X utolsó elemén túljutottunk és 0-át olvasunk be, akkor nekünk még az Y halmazban egy értéket meg kellett volna jósolni, ami 0-s értékkel, mint maszkinputtal nem lehetséges. Ennek a két hiányosságnak a kiküszöbölésére vezettem be egy jelzőértéket, amely az X halmaznál az első elemet jelenti (hogy ne egyből értékes inputtal kezdjünk, hanem azt is megpróbáljuk megjósolni) és az Y halmaznál pedig a 0 maszkértékek előtti elem (hiszen így az bejövő utolsó nemnulla inputra is egy „valós” outputot a flag-et tudjuk jósolni és nem a maszkértéket.)

Ezek alapján az tapasztaltam, hogy nagy teljesítménynövelő hatással jár, ha megjelöljük a bemeneti szekvencia kezdetét és a kimeneti szekvencia maszk előtti elemét. Az ábrán ezt a flag-et 49-cel jelöltem. Ennek az az oka, hogy ehhez a jelöléshez 0-át nem használhatunk, hiszen az maga a maszk értéke, konkrét azonosítót sem rendelhetünk hozzá, mert akkor nem lenne megkülönböztethető a rendes azonosító az flag azonosítótól, ezért templateID-k (1-48) + 1 (tehát a 49) egy olyan szám, amelyet még nem használtunk sehol így nem lesz zavaró a modell számára. Ezzel a szekvencia hossza 50-ről 51-re módosult. Ez egy saját megoldási ötlet, a DeepLog hagyományos algoritmusa ezt nem tartalmazza és ez minden esetben teljesítménynövekedéssel jár tapasztalataim szerint, hiszen ezzel nyerünk +2 tanulási értéket minden szekvenciában. Természetesen ez megnöveli az adatfeldolgozás folyamatát, azonban a jelen dolgozat célja, hogy a hatékonyságot a lehető legjobban növelni tudjuk.
Miután kitaláltuk, hogy milyen modell lesz alkalmas számunkra és ahhoz elvégeztük a tanító adathalmaz előkészítését léphetünk tovább a következő fázisra.

4.3. Tanítás a tanító adathalmazzal

Az előzőek alapján meghatároztuk azt a formátumú bemeneti és kimeneti adathalmazt, ami számunkra megfelelő lesz a tanításhoz. Az alábbi lépésben megtervezzük, hogy pontosan milyen rétegekre (layers) van szükségünk és azokat milyen módon paraméterezzük fel. A következő kódrészlet (6.ábra) mutatja a saját implementációmat ***tensorflow*** felhasználásával, ami tartalmazza sorban a neurális háló rétegeit *[17]*.

```
In [11]: regression = Sequential()

         from tensorflow.keras.layers import TimeDistributed

         regression.add(tf.keras.layers.Input(shape = (51,)))
         regression.add(tf.keras.layers.Embedding(50,100, mask_zero = True))
         regression.add(LSTM(units = 50, return_sequences = True, dropout=0.1))
         regression.add(LSTM(units = 100, return_sequences = True, dropout=0.1))
         regression.add(TimeDistributed(Dense(units = 49, activation = 'softmax')))

In [12]: regression.compile(optimizer = 'adam',  loss='sparse_categorical_crossentropy')

In [13]: regression.fit(X_train, y_train, epochs = 10, batch_size = 128, shuffle = True)
```

6. ábra

Először is definiálunk egy modellt, ami jelen esetben egy Sequential lesz, amelyre tudunk majd ráépíteni különböző rétegeket a szükséges funkció függvényében. A korábban (neurális hálóknál) leírtak alapján a neurális háló első szintje a bemeneti szint, ami annyi csomópontból áll, amekkora a bemenetünk mérete. Az imént láttuk (előző, groupping fejezetben), hogy a bemenetünk 51 elemű (50 elem + 1 flag), így az Input layer 51 méretű kell legyen.
Ezt követően lehet építeni a hidden layereket, amelyek a számításért felelősek.

4.3.1. Embedding Layer

Az LSTM számos szempontot figyelembe vesz, a következő érték megjósolásához és ilyen paraméter a bementi értékek közötti távolság. Ez annyit takar, hogy koherenciát keres abban, hogy a bemenetek (amelyek jelen esetben az azonosítók) értéke milyen távol van egymástól. Számunkra ez egy hibafaktor, hiszen az azonosítók értéke nem függ a szekvenciák milyenségétől. Keresnünk kéne egy olyan metrikát, amely segítségével ezt a problémát át tudjuk hidalni, azaz olyan azonosítót adni az elemeknek, ami kifejezi, hogy azok azonos távolságra vannak egymástól. Vizsgáljuk meg egy példán keresztül, hogy mit is jelent ez a gyakorlatban:
Feltételezzük, hogy van egy bementi szekvenciánk, ami az 1,2,4 elemekből áll. Az LSTM megvizsgálja, hogy ezek az elemek milyen távol vannak egymástól:

- 1 -> 2: 1 távolságra van egymástól
- 1->4: 3 távolságra van egymástól
- 2->4: 2 távolságra van egymástól

Látható, hogy eltérőek a távolságok, így ez is egy szempont lesz a kiértékelés során. Ennek nem szabadna így lennie, hiszen ezek csak egyszerű azonosítók, az értéküknek nincs szerepe.

Ennek a problémának a kiküszöbölésére kétfajta megoldást találtam:

1) One-hot-encoding (OHE):

A One-hot-encoding egy olyan transzformáció, ami a bemeneti számot átalakítja egy megadott elemű vektorrá, amelynek minden helyiértékén a 0-s érték szerepel kivéve a szám értékének a helyiértékét, ahol 1-es érték áll. Tehát ha a bemeneti érték a 2, akkor a vektor egy csupa 0 vektor lesz, amelynek egyedül a második eleme lesz 1. Belátható, hogy így bármely 2 vektort nézem pontosan 2 transzformációval egymásba alakítható a két vektor, amivel sikerült egységesíteni a távolságot az értékek között. Ez azonban egy költséges és hosszú transzformációt igényelne a bemenetünkön, amelyet meg lehet spórolni a másik megoldás segítségével.

2) Embedding Layer alkalmazása

Az Embedding Layer tulajdonképpen ugyanazt a módszert használja, mint az OHE annyi különbséggel, hogy nincs szükség a bemeneti adathalmaz transzformációjára. Az Embeddig Layer úgy működik, hogy a template azonosítónak megfelelő sort választja ki az embedding mátrixból, és azt adja tovább a következő rétegnek. Ez ugyanaz, mintha a template azonosítónak megfelelő one-hot-encoded vektort megszoroznánk az embedding mátrixszal és annak az eredményét adnánk tovább.

Emedding Layer a kódban látható paramétereinek magyarázata:

Az ***50*** jelenti az input dimenzióját, ami azt jelenti, hogy a legnagyobb érték nem lehet 49-nél nagyobb

A ***100*** jelenti az output dimenzióját, ami a kimeneti vektor hosszát mutatja, amit most 100-nak választottam.

mask_zero: True jelentése pedig, hogy maszkolás történt, méghozzá 0-val történő maszkolás.

4.3.2. LSTM layer

Az LSTM Layerek határozzák meg a hidden layereket a modellben. Az LSTM szintek paraméterezése jelentősen befolyásolhatja a végeredményt ezért ezek precíz megválasztása elengedhetetlen. Nekem a képen látható értékek jelentették a legjobb eredményt. A paraméterek:

units: Meghatározza, hogy hány csomópont helyezkedjen el a megadott szinten

return_sequence: Meghatározza, hogy visszatérjen-e az utolsó kimenettel vagy a teljes szekvenciával

dropout: Meghatározza, hogy a bemeneti értékek lineáris kombinációjából hányad részét hagyjuk el, a regularizálás (a túltanulás megelőzése) céljából

Látható a képen, hogy 2 ilyen LSTM réteget hozunk létre 0.1-es dropout értékkel. Kevesebb layer elhelyezése teljesítménycsökkenéssel járt, több szint pedig csekény javulást eredményezett azonban jóval lassabb lefutást.

4.3.3. Dense layer + compile + fit

Az utolsó layer – az output layer – egy úgynevezett Dense Layer, ami meghatározza a kimenetünket és annak típusát. Fontos megemlíteni, hogy 49 (a példaként bemutatott HDFS logfájlnak 48 különböző template-je van +1 a bevezetett 49-es flag) unit az output, ezt a következő bekezdésben részletesen kifejtem, azonban most tekintsük át az összes többi elemet. Látható, hogy a Dense Layer egyik paramétere az activation, aminek az értéke „softmax”.

Az activation = „softmax” aktiváció eredményeként a modell a kimenetre valószínűségeket generál, minden bemeneti értékhez pontosan 49 darabot, hiszen ennyi különböző értéke lehet a bemenetnek. Ezek a valószínűségek meghatározzák, hogy a bementi érték függvényében milyen valószínűséggel milyen kimeneti értéket generál.
A hagyományos DeepLog implementációk általánosan úgy működnek, hogy megnézik a modell milyen valószínűségeket állított elő az egyes bemeneti értékekhez és kiválasztják a legnagyobb 9 valószínűséget a listából. Ha a bementi érték valószínűsége szerepel ebben a top 9 valószínűségben, akkor a szekvenciát normálnak, ha pedig ezen kívül esik, akkor a szekvenciát anomáliának jelzik.
A saját megoldásomban a kapott 49 valószínűség közül nem a legnagyobb 9 valószínűséget vizsgálom, hanem mindössze azt az egyet, amelyik a kimeneti értékhez tartozik. Tehát ha a helyes kimenet értéke 3, akkor a harmadik jósolt valószínűséget megnézem és eltárolom. Ha ez a valószínűség egy bizonyos küszöbérték alatti érték, akkor a szekvenciát anomálisnak, más esetben pedig normálisnak jelzem.

Miután összeraktuk a neurális hálónkat következhet a model.compile:
Először is kiválasztunk egy *optimizer*-t, amely jelen esetünkben „***adam***” lett. Határozottan jó eredményt kaptam ennek használatával, de nem tartom kizárhatónak más optimizer használatát. Ami viszont számunkra fontosabb az optimizer-nél az a *loss függvény* meghatározása. A *tensorflow.keras* dokumentáció külön kiemeli, hogy abban az esetben ha valaki one-hot reprezentációt használ, használjon CategoricalCrossentropy loss-t hozzá. Azonban mi csak formai one-hot reprenztációt használunk (Embedding Layer-t), amihez alkották meg a SparseCategoricalCrossentropy-t. Ennek okán mi a loss függvényt a ***sparse_categorical_crossentropy*** használatával számoljuk
A *compile* után a *fit* következik, ami a konkrét tanulási folyamatot jelenti a definiált halmazon a megadott iterációszámmal. Ezt reprezentálják a paraméterek is:
x: Ennek helyére kerül a bemeneti adathalmaz, amivel tanítani fogunk. Ez jelen esetben az X_train
y: Ennek helyére kerül a kimeneti adathalmaz, ami a visszacsatolást adja. Ez jelen esetben az y_train
batch_size: Meghatározza, hogy a tanult adatokkal történő frissítés hány „adatsor” olvasása után történjen. Ez az adatsor áll egy bemeneti sorból és egy kimeneti sorból. Ha a *batch_size*

értékét 1-nek választjuk (annál kevesebb nem lehet), akkor a modell belső paramétereit minden sor olvasása után frissíti. Ha a *batch_size* az adatsorok számával megegyezik (annál nagyobb nem lehet), akkor az összes adatsor beolvasása után frissíti a belső paramétereket. Ha nem adunk meg neki értéket akkor 32 az alapértelmezett. A vizsgálataim során a ***128***-as érték vállt be.

epoch: Meghatározza, hogy hányszor menjen végig a tanulóalgoritmus az egész adathalmazon. Miután lefutott a megfelelő epoch-számú iteráció készen vagyunk a tanítási fázissal és következhet a validáció és a teszteli fázis.

4.4. Kimenet előrejelzése validációs adathalmazzal

Az adatfeldolgozás első szakaszában 3 nagy csoportra osztottuk az adathalmazt:

- tanító adathalmaz: Amely segítségével betanítjuk a rendszert, megmutatjuk az adott bemenethez tartozó kimenetet és folyamatos visszacsatolással pontosítjuk a tudást
- validációs adathalmaz: Amelyet a küszöbértékek megállapítására használunk
- tesztadathalmaz: Ez egy új, a modell számára még nem látott adathalmaz, amivel a tesztelést és a kiértékelést fogjuk elvégezni

Eddig a tanító adathalmazzal foglalkoztunk, azonban az elkövetkezendőkben a második nagy csoporttal, a validációs adathalmazzal fogunk foglalkozni.

Miután létrehoztuk a modellt és elvégeztük a tanítást, következhet a validálás folyamata. A modelltől most már elvárjuk, hogy ha kap egy – a korábbihoz hasonló – bemeneti adathalmazt, akkor meg tudja állaptani, hogy azoknak milyen kimenete van. Ehhez fog elsősorban szolgálni a validációs adathalmaz. A neurális hálókra általában jellemző az erős interpoláció és nagyon gyenge az extrapoláció.

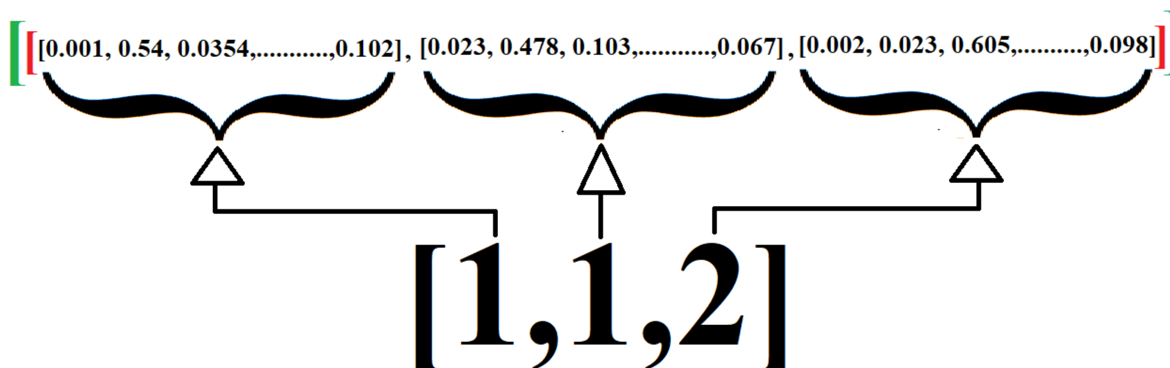
Alapvetően a *Sequential modell* biztosít nekünk egy beépített lehetőséget predikcióra. Meg kell mondani, hogy melyik adathalmazon végezzük a predikciót (jelen esetben a validációs adathalmazon). Konkrét példát erre a 7.ábrán láthatunk:

```
In [22]: y_pred = regressior.predict(X_val)

         4492/4492 [==============================] - 26s 5ms/step
```

7. ábra

A megoldásomban a softmax activation miatt ez a predict az X_val-hoz tartozó outputot „jósolta" meg úgy, hogy minden lehetséges értékhez valószínűséget rendelt. Ebből következik, hogy egy háromdimenziós tömb lesz a kimenete ennek a predikciónak. Az alábbi kép is ezt szolgáltatott bemutatni:



8. ábra

Tegyük fel, hogy az X_train egyik szekvenciája az 1,1,2-es azonosítójú Event-eket tartalmazza. Minden elemhez, tehát az 1,1,2-höz külön-külön visszatérít a prediction egy tömböt (lásd fekete

kistömbök), amely az egyes EventID-khoz tartozó valószínűség található. A példában látható, hogy az első 1-es bemenethez 0.54 valószínűséget jósolt a modell. A 2-es valószínűsége pedig 0.605-nek bizonyult, stb…
A piros szögletes zárojellel jelölt tömb az egy szekvenciához tartozó elemek határa, tehát a piros tömb jelenleg 3 kistömböt tartalmaz, hiszen 3 eleme van a vizsgált szekvenciának. Ebből látható, hogy egy szekvencia önmagában egy kétdimenziós tömböt ad vissza, mert a szekvencia több elemű és minden elemhez 50 elemű kistömb tartozik. És mivel nem 1, hanem sok szekvenciát vizsgálunk, így lesz az eredmény háromdimenziós.
Miután megjósolta nekünk a modell a **prediction** segítségével az elvárt kimenetet, kiszűrjük belőle a számunkra szükséges valószínűségeket. Ha a szekvenciában 1-es elem szerepel, akkor megnézzük, hogy mekkora valószínűséggel jósolta meg az 1-es értéket és azt az értéket eltároljuk. Ezek alapján sikerült minden szekvenciához annyi valószínűséget rendelni, ahány elemet tartalmaz, így ismételten 2 dimenzióssá változott vissza a tömbünk. Ahhoz, hogy ebből információt tudjunk kinyerni, meg kell vizsgálni, hogy van-e esetleg olyan valószínűség, ami feltűnően kicsi, hiszen az indikálja az anomália jelenlétét az adott szekvenciában. (Ha normális lenne, akkor a korábban megszerzett tapasztalatok alapján magas valószínűséget kapott volna a modelltől az adott érték). Azonban merülhet fel a kérdés, hogy pontosan mi az, hogy „feltűnően kicsi”?
Most bontakozik ki valójában a validációs adathalmaz szerepe. Ennek segítségével próbáljuk meg minél pontosabban definiálni, hogy mi az a valószínűségi érték, aminél kisebbet feltűnően kicsinek titulálunk és mi az, amit még elfogadhatónak. Ezt nevezzük küszöbértéknek (thresholdnak). **A threshold megválasztásához manuális módszert választottam, tehát egy bizonyos intervallumon bizonyos léptékkel végigmegyek és megvizsgálom, hogy ha ezt az adott értéket választottam volna thresholdnak, akkor milyen pontossággal teljesít a predikció**. (A kiértékelés pontos folyamatáról és a kiválasztott metrikáról később írok).
Az eddig felmerült információ birtokában kiválasztottam a megfelelő thresholdot, ami kritikus fontosságú lesz a következő részben.

4.5. Kimenet előrejelzése a tesztadathalmazzal

Miután elvégeztük a strukturálást, a csoportosítást, a megfelelő szűréseket, a tanítást és a threshold megállapítását jöhet a tesztelés fázisa. Erre hagytuk az adathalmaz 3. részét a teszthalmazt. Hasonló lesz itt is az eljárásunk, mint a validációs halmaznál, azonban itt már rendelkezésünkre áll a threshold ami az előbb a szűk keresztmetszet volt.
Első lépésként előkészítjük a korábbiakban bemutatott módon a teszthalmazt is majd ezt átadjuk a *predict* függvénynek, mint paraméter és az így kapott eredmény lesz a rendszer által jósolt háromdimenziós tömb. Minden elemhez szekvenciánként tartozik egy tömb, abból kiválasztjuk a „helyes” elemhez tartozó valószínűséget, majd ezeket egybevesszük. Mostmár ismert számunkra a threshold, így ezek alapján meg tudjuk állapítani, hogy ahol van olyan valószínűség, amely kisebb ennél a threshold-nál azt a szekvenciát anomáliásnak, ahol nincs azt a szekvenciát pedig normálisnak nyilvánítjuk.

Az általam készített implementációt egy fajta megoldása a DeepLog algoritmusnak, ami több előnyt jelent. A konklúzióban azokat az előnyöket felsorolom. Sok időt vett igénybe, de megérte hiszen sokat tanultam belőle.

5. KNN algoritmus

5.1. Bevezető

A DeepLog implementálása után egy olyan algoritmust sikerült létrehozni, ami stabilan képes logfájlokban anomáliát detektálni. Ebből fakad, hogy ennek az algoritmusnak számtalan megoldási módja és gyakorlati használata van. Mint mindennek ennek is megvannak a hátrányai:

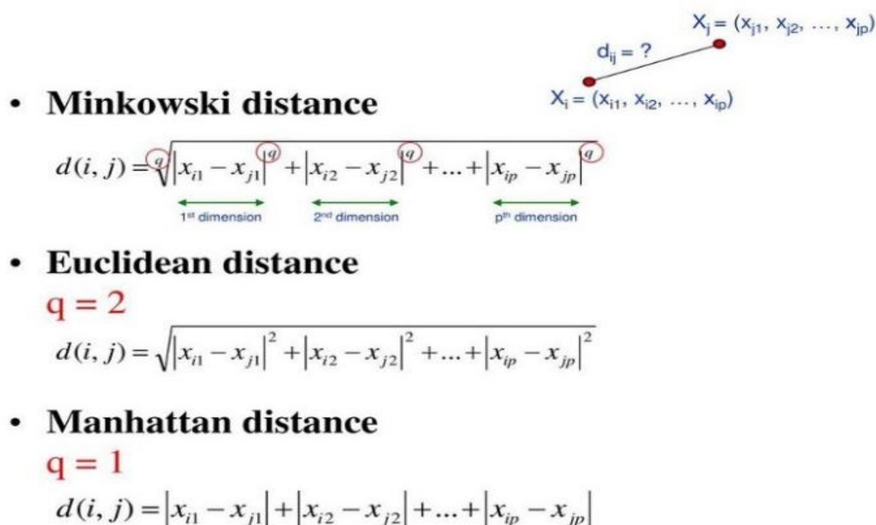
- Elég lassú, hiszen rengeteg adatfeldolgozást igényel. Láthattuk, hogy mennyi mindent műveltünk az adathalmazokkal azért, hogy tudjuk tanítani velük a neurális hálót és mennyi művelettel jár a kapott eredmény helyes értelmezése
- A DeepLog algoritmus 95-96%-os eredményt produkál HDFS logsorok esetében, míg vannak olyan megoldások, amelyek képesek ezt túlszárnyalni

Ez a két mérték határozza meg leginkább egy algoritmus jóságát és használhatóságát. A saját DeepLog implementációt referenciaként használom majd a KNN algoritmusnál. Ez annyiban fontos, hogy így a bemeneti adatok adottak lesznek, hiszen, ha össze akarom hasonlítani a két eredményt akkor ugyanazzal a bementi adattal kell számolnom. Így a következő részben található workflow-ból kivettem azokat a részeket, amiket meg kéne ismételni, ha nem lenne meglévő referenciánk.

5.2. KNN algoritmus általánosan

Ez az egyik legegyszerűbb gépi tanulásos, supervised algoritmus, azonban az egyszerűsége ellenére nem rendelkezik akkora irodalommal a loganalitika területén, mint az LSTM alapú DeepLog. A létező KNN megoldások log anomália detektálására mindegyike logsorról logsorra dolgozza fel az adathalmazt és ennek segítségével keres anomáliákat. [8] Az én megoldásom ettől eltérve nem egyes logsor anomáliák detektálásával foglalkozik, ***hanem szekvenciális anomáliákkal***.

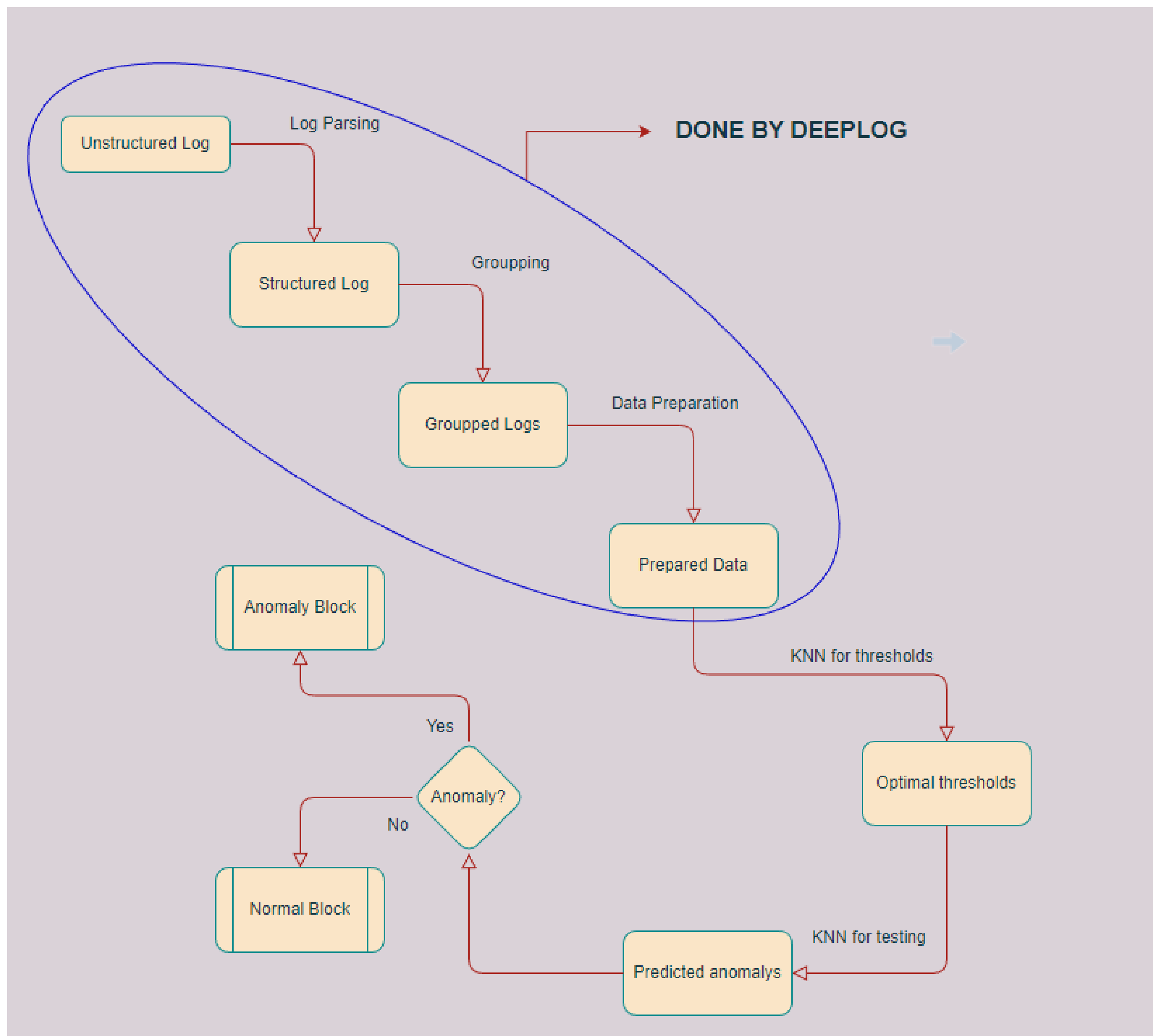
A KNN alapvető gondolata, hogy a „hasonló” elemek egymáshoz közel helyezkednek el, így a szomszédok ismeretéből lehet következtetni az újonnan érkező érték tulajdonságaira. Az angol nevéből is kiolvasható (K-Nearest Neighbors), hogy itt tulajdonképpen a k-dik legközelebb szomszéd a kérdéses, hogy az milyen távol helyezkedik el a bejövő értéktől. A távolság meghatározására számos távolságmetrikát alkalmaznak (9. ábra) [8], azonban számunkra ez nem lesz megfelelő, hiszen mi nem pontok közötti távolságot határozunk meg, hanem potenciálisan eltérő hosszúságú szekvenciák közötti távolságot.:



9. ábra

Szekvenciák távolságmérése sokkal nehezebb feladat a pontoknál. Vannak rá módszerek (lásd: távolságmetrikák szekvenciákhoz alfejezet), azonban ezek sem voltak elengedőek a probléma megoldására. ***Ennek érdekében egy saját súlyozási sémával testreszabott súlyozott távolságmetrikát használok, ami alkalmasnak bizonyult a probléma megoldására. A szekvenciák távolság mérése, szinte a legkritikusabb része a munkámnak a KNN algoritmus esetén. Látni fogjuk az eredményeimből, hogy szignifikáns javulást eredményez az én megoldásom, a klasszikus Levenshtein metrikákhoz képest.***

5.3. Általános KNN munkamenet



10. ábra

5.3.1. DeepLog által előállított adatok

Mivel a DeepLog algoritmus referenciaként használom a KNN algoritmushoz, így ahhoz, hogy reprezentatív eredményt kapjak, ugyanazzal az adathalmazzal kell dolgoznom. A DeepLog algoritmusnál a kapott eredményeket fájlokba kimentettem, hiszen:

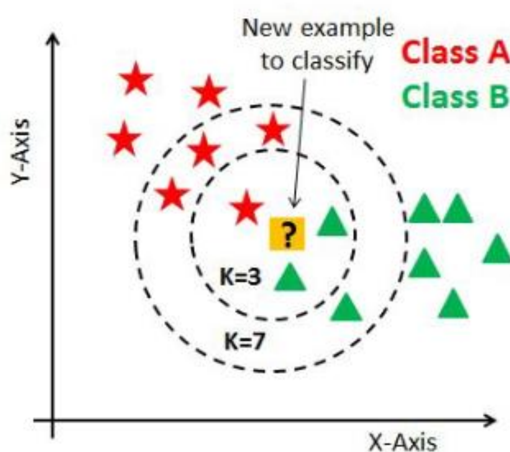
1) Ha újra fel akartam használni más paraméterekkel az algoritmust, akkor a hozzá szükséges adatot nem kell újra betölteni (ami hosszú folyamat), hanem egy gyors fájlművelettel be tudom olvasni
2) Mivel be van építve egy randomizált keverés az adathalmazok elemeire, így, ha azt újra felhasználnám a KNN algoritmusnál akkor nagy eséllyel más adathalmazt kapnék és ennek a probléma megoldására célszerű a fájlművelet.

Ezek alapján a korábban megismert három adathalmazra történő (train-validation-test) bontás már megvalósult, a különböző Eventek-hez saját azonosítót rendeltünk stb... Az alábbiakban megvizsgáljuk a KNN algoritmus részleteit és minden olyan elemet a munkamenet-ben, ami az adatelőkészítés után van.

5.3.2. Az algoritmusban használt paraméterek, metrikák, küszöbértékek

5.3.2.1. K paraméter

A KNN algoritmusban az adatok előkészítése után egy fontos paraméter a k értékének meghatározása. Ez definiálja, hogy hányadik legközelebbi elemet fogjuk vizsgálni a szomszédoktól (11. ábra). Ennek az értéknek a megválasztására nagyon feladatfüggő, általánosan működő, előre meghatározott módszert nem találtam az érték megállapítására. Azonban a k tulajdonságaiból fakadóan néhány szempontot figyelembe vehetünk.



11. ábra

Ha a K értékét túl kicsinek választjuk meg, akkor várhatóan csak a tényleg kiugróan távoli eredmények kerülnek bele a megoldáshalmazba, hiszen azoknak a nagyon közeli környezetében nincsen másik elem. Ez várhatóan megnöveli a helyesen előrejelzett anomáliás szekvenciák számát, azonban számos elemet nem talál meg, hiszen egy olyan érték, amelynek csak egy pár szomszédja van közel, a legtöbb elem nagyon távol helyezkedik el, az is várhatóan anomáliás lesz, de a kis k érték miatt normálisnak lesz ítélve.
Ha a K értékét túl nagyra választjuk, akkor előfordulhat, hogy egy normális block is anomálisnak lesz ítélve, hiszen akkor a K értéke, hogy a tőle messze álló normális block-ok is kiválasztásra kerülnek, amelyek helyezkedhetnek egymástól jó messze. Ha megnézzük a 12. ábrát és kiválasztjuk a bal felső sarokban található piros csillagot vizsgálandó elemnek és a K-t 7-re választjuk, akkor a 7. legközelebbi elem már zöld háromszög lesz, ami jelentősen messzebb van a pirosaktól. Ha ez az érték kiugrónak bizonyul, akkor anomálisnak lehet ítélve. Tehát túl nagy K esetén a legtöbb anomália megtalálásra kerül, azonban hibásan néhány normális is anomálisként lesz kezelve.
A fenti két eredményt összefoglalva, ha a cél azt követeli meg, hogy amit anomáliásnak titulálunk az anomáliás is legyen, de előfordulhat olyan elem is, amit nem találunk meg, akkor

érdemes a kis K értéket választani. Ha viszont az a célunk, hogy biztosra menjünk és minden anomáliát megtaláljunk olyan áron is akár, hogy normális blockot is anomáliásnak jelzünk, akkor válasszuk a nagyobb K értéket. (Számunkra ez lesz a jobb választás, hiszen minden elmulasztott anomália hatalmas veszteséggel járhat, míg, ha egy normális blockot anomáliának jelzünk, akkor az „csak” az erőforrás kárára megy). Az ideális megoldás, ha sikerül egy olyan köztes értéket találni, ami mindkét megoldás előnyével rendelkezik, de a szélsőséges hátrányait lehagyja. Mivel ahogy korábban írtam, erre nincsen egy általánosan használható metódus, így a sok próbálkozás és tesztelés jelentette nálam a megoldást. Reményeim szerint sikerült a megfelelő próbálkozással az optimális k értéket megtalálni.

5.3.2.2. Távolságmetrikák szekvenciákhoz

Mielőtt megvizsgáljuk az értékét nézzük meg, hogy pontosan mi ez és miért van rá szükség. A bevezetőben említettem, hogy a KNN-hez szükséges egy távolságmetrika definiálása, ami alapján meg tudjuk határozni, hogy mekkora a vizsgált elem távolsága a k-ik legközelebbi szomszédtól. Ennek a távolságmetrikának a helyes kiválasztása szekvencia esetén nem egyszerű feladat. Számos tényező figyelembevétele szükséges, ilyen például a változó szekvenciahosszúság. Ha rosszul választjuk meg a metrikát akkor az egész algoritmus hibásan fog működni. A szekvenciák közötti távolság meghatározására kezdetben Levenshtein modult használtam [19]. Ez a modul számos különböző metrikával rendelkezik, vizsgáljunk meg egy párat közülük:

Levenshtein distance [21]

A distance 2 kötelező paramétert vár: A vizsgált és a referencia szekvenciát. Müködési elve a következő:

Megvizsgálja, hogy a vizsgált szekvencia hány transzformáció segítségével alakítható át a referencia szekvenciává. Ilyen transzformációs művelet a beillesztés, a módosítás és a törlés. Fontos, hogy ezek a műveletek súlyozhatóak, azaz megadható, hogy mekkora súllyal vegye figyelembe a különböző műveleteket. Ennek a default értéke az 1, de átadható 3. paraméternek egy 3 elemű tuple, amiben a súlyok vannak definiálva. Nézzük meg a működését egy konkrét példán (12. ábra):

```
In [81]: from Levenshtein import distance, editops

In [82]: distance([1,2,3], [2,3,4,5])
Out[82]: 3

In [83]: editops([1,2,3], [2,3,4,5])
Out[83]: [('delete', 0, 0), ('insert', 3, 2), ('insert', 3, 3)]
```

12. ábra

A Levenshtein distance metrika szerint az [1,2,3] és [2,3,4,5] szekvenciák távolsága 3, hiszen 1 törlés és 2 beszúrás művelettel megoldható. (Az editops csak egy built-in eszköz, hogy kilistázzuk, hogy pontosan milyen műveletek lettek végrehajtva).

Lépések:

- (’delete’,0,0): Az első paraméter jelzi, hogy milyen művelet lett végrehajtva. A második paraméter jelzi, hogy a vizsgált szekvencia melyik elemével végeztük el a műveletet és a második paraméter jelzi, hogy a referencia szekvencia melyik értékét sikerült ezzel a művelettel a „helyére” tenni. Azzal, hogy kitöröljük a vizsgált szekvencia 0. elemét, azzal kapunk egy [2,3] elemeket tartalmazó szekvenciát, amiben a 2-es már a helyén van.

- ('insert,3,2): Itt be fogunk szúrni a 3. helyre (a kezdeti, még a törölt előtti állapotból kiindulva 3. helyre) egy értéket (4-est), méghozzá úgy, hogy ezzel a referencia szekvencia 2. eleme (a 4-es) is a helyére kerül: a vizsgált szekvencia így [2,3,4] lesz.
- ('insert',3,3): Itt is be fogunk szúrni a kezdeti szekvenciából kiindulva 3. helyre (szekvencia végére) egy elemet, méghozzá a referencia szekvencia 3. elemét (5-öst). Így megkaptunk valóban a [2,3,4,5] szekvenciát

Ha hozzáveszünk egy súlytuple-t akkor tudjuk közvetlenül módosítani a kapott távolságértéket. Nézzük meg a felső szekvenciát úgy, hogy a törlés súlyát 2-re állítjuk a default 1 érték helyett (13. ábra):

```
In [97]: distance([1,2,3], [2,3,4,5], weights = (1,2,1))

Out[97]: 4
```

13. ábra

Ezzel tehát láttuk, hogy a **distance** metrika pontosan hogyan működik és hogy miért a 3-mas értéket térítette vissza ennek a 2 szekvenciának a távolságára kezdetben, majd a törlés súlyát megemelve 2-re máris módusult az eredmény 4-re. Ez az metrika tehát egy pozitív egész számot térít vissza, ami a távolságot fogja reprezentálni.

Jaro distance [22]

Egy másik közismert szekvencia mérésére alkalmas mérték a *Jaro* távolság. Ennek az értéke egy 0-1 közötti szám, amely az alábbi képletből adódik (14. ábra) [20]:

$$Jaro\ similarity = \begin{cases} 0, \text{ if m=0} \\ \frac{1}{3}\left(\frac{m}{|s1|} + \frac{m}{|s2|} + \frac{m-t}{m}\right), \text{ for m!=0} \end{cases}$$

14. ábra

A képletben található paraméterek a következők:
|s1|: A vizsgált szekvencia hossza (elemszáma)
|s2|: A referencia szekvencia hossza (elemszáma)
m: Azonos értékek száma
t: Nem a helyes sorrendben lévő értékek számának a fele
Nézzük meg az előző példán keresztül a Jaro távolságot:

```
In [4]: jaro([1,2,3], [2,3,4,5])

Out[4]: 0.7222222222222222
```

15. ábra

|s1| = 3, mert három elemű az első szekvencia (1,2,3)
|s2| = 4, mert négy elemű a második szekvencia (2,3,4,5)

m = 2, hiszen kettő azonos érték van a két szekvenciában: a 2-es meg a 3-mas
t = 2, hiszen a célszekvencia 4 értékből az egyik sincs helyes sorrendben, így ez 4 lesz, aminek a fele 2
Ebből tehát a képlet:

$$\frac{1}{3} * \left(\frac{2}{3} + \frac{2}{4} + \frac{2}{2}\right) = 0.72222$$

Hamming distance

A Hamming metrika két szekvencia Hamming-távolságát határozza meg. Feltétele, hogy a két szekvencia azonos hosszúságú legyen. A Hamming-távolság lényegében annyit takar, hogy hány cserével lehet az egyik szekvenciát a másikba átalakítani. Hasonló, mint a distance, viszont sem beszúrni sem törölni nem lehet, csak módosítani elemet.

A fenti metrikák bemutatása után megállapíthatjuk, hogy a szekvenciák távolságát számos módszerrel tudjuk mérni. Megoldásomban kipróbáltam a **distance** és **Jaro** metrikát egyaránt (a **Hamming**-et nem, hiszen nem képes változó méretű szekvenciák közötti távolság meghatározására) és az F1 score-ra kapott eredmény 70%-on stagnált. Akárhogy módosítottam a k értékét egy bizonyos százaléknál nem teljesített jobban a rendszer. Számos utánanézés és vizsgálat után arra a következtetésre jutottam, hogy a probléma a szekvenciaelemek gyakoriságában keresendő. Észrevettem, hogy előfordulhat 2 olyan szekvencia, ami egy másik szekvenciától azonos távolságra van, így a KNN algoritmus közel sorolja be őket egymáshoz, azonban az egyik normális a másik pedig anomáliás szekvencia. Nézzük meg az alábbi szekvenciát a példa kedvéért és használjunk a távolság méréséhez Levenshtein-distance metrikát (16.ábra):

```
In [60]: X_anomaly_test[8]
Out[60]: [1, 2, 1, 1, 3, 4, 3, 4, 3, 4, 5, 5, 5, 32, 32, 32, 17, 17, 17, 33]

In [61]: X_train[14]
Out[61]: [1, 2, 1, 1, 3, 4, 3, 4, 3, 4, 5, 5, 5, 32, 32, 32, 17, 17, 17]

In [62]: X_train[102]
Out[62]: [1, 2, 1, 1, 3, 4, 3, 4, 3, 4, 5, 5, 5, 13, 32, 32, 32, 17, 17, 17]

In [63]: distance(X_anomaly_test[8], X_train[14])
Out[63]: 1

In [64]: distance(X_train[14], X_train[102])
Out[64]: 1
```

16. ábra

A 16. ábrán látható, hogy egy anomáliás szekvencia (X_anomaly_test[8]) és egy normális szekvencia (X_train[102]) azonos távolságra található egy normális szekvenciától (X_train[14]), amit mi távolságreferenciaként használni fogunk. Jelen esetben mindkét távolság értéke 1. Ami a jelenlegi példát illeti mindkét esetben egyetlen beszúrás történt, anomáliás esetben a 33-as elem, normális esetben a 17-es elem beszúrása. Ami feltűnt, hogy annak ellenére, hogy 1-1

beszúrt elemről beszélünk, az adott elem előfordulási gyakorisága a szekvenciákban nagy eltérést mutat:

A 33-as elem gyakorisága: 5545 (ez a vizsgált HDFS logsorok 0.04%-a)

A 17-es elem gyakorisága: 1.402.047 (ez a vizsgált HDFS logsorok 12.54%-a)

Nekem ebben az esetben meg kell különböztetnem a két szekvencia távolságát, hiszen máskülönben ezt a „közeli” anomáliát nem fogom tudni detektálni.

Erre egy megoldási módszer lehet, ha tudnánk súlyozni nem csak a beszúrás-módosítás-törlés tényét, hanem az adott elem függvényében tudnánk meghatározni a súlyt. Tehát az lenne a cél, hogy a 33-as elem transzformációja drágább lenne, mint a 17-es elemé, hiszen jóval ritkábban fordul elő. Mivel az eddig megismert metrikák erre nem alkalmasak, ezért keresnem kellett egy másik metrikát. Erre még ennél is kevesebb megoldást találtam, azonban egy jónak bizonyosult: a **weighted-Levenshtein** metrika.

weighted-Levenshtein distance

Ez a metrika alapvetően annyiban tér el a sima Levenshtein distance metrikától, hogy a súlyozás nem egy 3 egész számot tartalmazó súlytuple-ből áll, hanem a beszúrásra és törlésre egy-egy lista definiálása (ezek egyoperandusú műveletek), míg a módosításra egy mátrix definiálása lehetséges (ezek kétoperandusú műveletek). A listák működése az alábbi:

Alapvetően az alapértelmezett beállítás egy előre definiált 1-eseket tartalmazó lista, ami annyi elemből áll, amennyi a lehetséges legmagasabb elem értéke (jelen esetben 48 + 1 = 49, erre a +1-re az indexelés miatt van szükség). Ha a listában átírjuk a 1. elemet 2-re akkor a 1-es elem beszúrása vagy törlése (attól függ, hogy jelenleg a beszúrásnak vagy a törlésnek a listáját határoztuk meg) 2-es költséggel fog megjelenni. Tehát a mi esetünkben, ha átírom a beszúrási mátrix 33. elemét 2-re és hagyom a 17.elemét 1-nek, akkor máris távolabb lesz az anomáliás szekvencia a normálistól.

A módosítási mátrix elemei meghatározzák, hogy mely elem mely elemre történő cserélése mennyibe kerül. Tehát a mátrix sora meghatározza, hogy mely elemet akarjuk cserélni, az oszlopa pedig, hogy mely elemre. A kettő metszetén lévő elem pedig ennek a költségét jelenti. ((1,2) pozícióban megtalálható érték az 1-es elemet 2-esre cserélve történő műveletnek a költségét jelenti).

Ezen a ponton az alábbi kérdést kell megválaszolnunk:

- Mekkora előfordulást tekintünk kevésnek és mekkorát normálisnak?

  Az előző példában láthattuk, hogy az 5545-ös előfordulású 33-as elem ritkának kell, hogy számítson, viszont az 1.402.047 17-es elem már nem. Ezt az értéket nevezzük gyakorisági threshold-nak. ***Ennek a megválasztása manuálisan történt, megvizsgáltam, hogy milyen gyakorisági threshold mellett lehet maximalizálni az eredményt és ezalapján határoztam meg.*** Látható, hogy ennek az értéknek a megválasztása nem lehet statikus, hiszen amint bővül a logsorok száma, úgy dinamikusan kell változnia ennek az értéknek is. Ezért ezt az értékét a teljes logsor százalékos arányában határoztam meg és a legjobb eredményt 0.05%-os határnál jelöltem.

5.3.2.3. Anomália-súly threshold:

Az Anomália-súly threshold meghatározza, hogy mekkora súlyt rendeljünk a ritkán előforduló értékeknek (az adott listában, vagy mátrixban mekkora érték tartozzon hozzá). Erre a konstans 50-es érték számomra megfelelt, ezzel jó eredményeket kaptam.

5.3.2.4. Eredmény-threshold:

Az eredmény-threshold meghatározza azt az értéket, hogy mekkora legyen a távolságérték, ami felett jelzünk anomáliát. A korábban kiválasztott távolságmetrika meghatározásánál a kulcsszerepet játszott az a gondolat, hogy a normális szekvencia egy másik normálistól közel legyen, egy normális pedig egy anomálistól messze. Az eredmény-threshold ezt a kérdést feszegeti, hogy pontosan mit jelent a közel és a messzi kifejezés. Ezt a validációs adathalmaz segítségével fogjuk megállapítani. Fontos, hogy referenciaként csak normális szekvenciákat fogunk használni, hiszen az ezektől mért távolság releváns számunkra. A DeepLog algoritmus tanító adathalmazában szintén nem szerepelt anomáliás szekvencia, így ezt tökéletesen tudjuk itt is használni (ez fontos hiszen, ha össze akarjuk hasonlítani a két algoritmust akkor ugyanazzal az adathalmazzal kell tanítani/validálni/tesztelni). Az eredmény-threshold meghatározásához a következő a módszer:
Minden egyes validációs adathalmazbeli szekvenciát összehasonlítunk a tanítóadathalmaz minden szekvenciájával és megvizsgáljuk a közöttük lévő távolságot. Ezeket összegyűjtjük és amint megkaptuk az összes távolságot, kiválasztjuk a növekvő sorrendbe rakott lista k. elemét (így kapjuk meg a k. legközelebbi szomszédtól vett távolságot). Ezt az értéket elmentjük, az adott szekvenciához ezt az értéket rendeljük hozzá, mint távolságérték. Miután megkaptuk az összes szekvencia távolságértékét megvizsgáljuk, hogy milyen eredmény-threshold mellett kapunk pontos predikciót. A DeepLog-hoz hasonlóan itt is egy iteráció segítségével nézzük végig az egyes thresholdok mellett a kapott eredményt és annak függvényében állapítjuk meg azt.

5.3.3. KNN tesztelése

Miután meghatároztam a 3 „sima” thresholdot meg a validációs adathalamaz segítségével az eredmény-threshold-ot vizsgálataim szerint optimálisan, elkezdődhet a modell tesztelése. A tesztadathalmazt a validációs adathalmaz mintáját követve minden szekvenciáját összehasonlítjuk a tanító adathalmaz minden szekvenciájával és a k. legnagyobb elemet kinyerjük belőle. A korábban meghatározott eredmény-thresholdot figyelembe vesszük és megvizsgáljuk, hogy a kapott k. legnagyobb távolság milyen viszonyban áll az eredmény-thresholddal. Ha nagyobb az k. legnagyobb távolság, akkor anomáliát jelzünk, minden más esetben pedig normálisnak tituláljuk a szekvenciát.

6. HDFS, mint címkézett logfájl ismertetése

6.1. HDFS, mint címkézett logfájl

A HDFS (Hadoop Distributed File System) egy publikusan elérhető logfájl, amely egyike azon kevés elérhető logfájloknak, ami címkével van ellátva. Ezek a címkék manuálisan lettek hozzárendelve és megmutatják, hogy egy adott blokk normális vagy anomáliás.
A neurális háló tanításához szükségünk van címkézett logfájlra, hiszen folyamatos visszajelzést kell nyújtanunk, hogy az előrejelzések közül melyik helyes és melyik helytelen. A publikusan elérhető logfájlok száma limitált és ha találunk ilyen nyilvános adatforrást, akkor vagy kevés logsor címkézett benne, vagy nem megbízhatóak a címkék. Egyike a gyakran használt és bevállt logfájloknak a HDFS logfájl. Kettő változat elérhető jelenleg:

1) HDFS_2k
   Ez egy rövidített változat, 2000 logsort tartalmaz és ezzel lehet bemutatni/tesztelni az algoritmust.
2) HDFS
   Ez az eredeti változat, több mint 11 Millió logsorral. Én ezt a logfájlt használom fel az algoritmusomban.

Mindkét logfájl letöltésével kettő adatot kapunk meg:

1) HDFS.log
   Ez maga a strukturálatlan, nyers logfájl, amiből a LogParser segítségével strukturált adatot generáltunk. Ez a fájl
2) anomaly_label.csv
   Ez egy olyan fájl, ami tartalmazza számunkra a leghasznosabb adatot: A címkéket. Megtalálható benne, hogy az adott block normális vagy anomáliás-e. Ezt fogjuk használni a tanítás során visszacsatolásnak és természetesen ezzel fogjuk validálni és tesztelni a deep-learning algoritmusunk által generált predikció pontosságát. Fontos emellett, hogy ezeknek a címkéknek köszönhetően képesek vagyunk semi-supervised módon tanítani a rendszert, ami a korábban kifejtett előnyökkel jár.

Én ezeket az alábbi linkről töltöttem le: Loghub | Zenodo. Látható, hogy 2021. szeptember 1-je óta (amikor publikálták a honlapot) **82.351 letöltés történt**, ami egy bíztató referenciának tekinthető.

6.2. HDFS adatállomány felosztása

Ahogy azt korábban említettem, alapvetően 3 részre bontom a HDFS adatállományt:
training (tanító) adathalmazra, validation (validációs) adathalmazra és test (teszt) adathalmazra. Az alábbi táblázat megmutatja, hogy milyen értékeket tartalmaznak ezek az adathalmazok és milyen jellemzőik vannak:

	Blockszám	Randomizált-e	Tartalmaz-e anomáliás blockot
Tanító adathalmaz	*287.500*	*Igen*	*Nem*
Validációs adathalmaz	*143750*	*Igen*	*Igen*
Tesztadathalmaz	*143751*	*Igen*	*Igen*

Ahogy az a táblázatból is kiolvasható, a tanító-validációs-tesztadathalmazt egy: $\frac{1}{2}\ \frac{1}{4}\ \frac{1}{4}$ -es felosztásban osztottam fel. ***Tapasztalataim szerint szükség van legalább az adatállomány felére a megfelelő tanításhoz, de többet nem érdemes „rááldozni", hiszen az a validáció és a tesztelhetőség kárára megy***. Mindemellett fontos, hogy az adatállományok randomizáltak legyenek, máskülönben a modell a szekvenciák sorrendjéből is képes információt leszűrni, ami szintén rontja a teljesítményt. Mindemellett fontos, hogy a tanító adathalmaz nem tartalmazhat anomáliás blockot, hiszen a tanítással az a célunk, hogy a helyes mintákat tanítsuk meg a modell-nek, így fel tudja ismerni az anomáliás eseteket. Ha anomáliás block bekerülne a tanítási folyamatba, az szintén hatásfokcsökkenéssel járna.

6.3. A HDFS-ben megtalálható template-k és azok gyakorisága

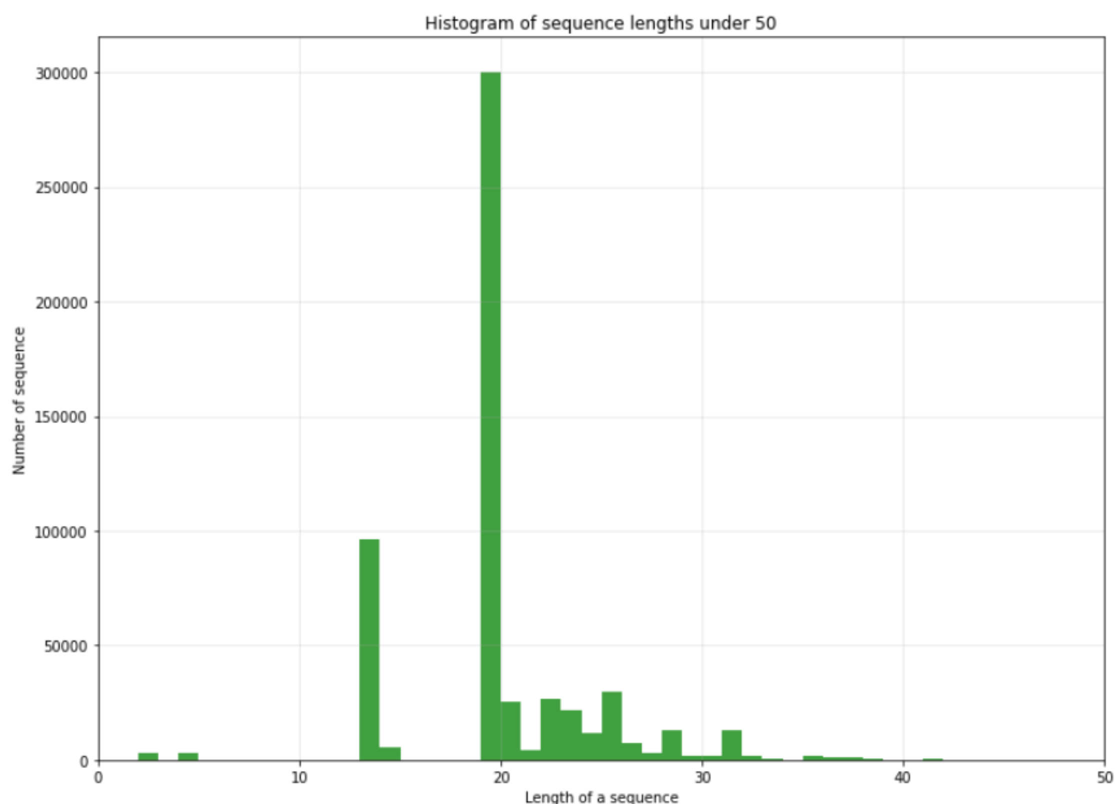
Az alábbiakban látható, hogy mely azonosítókból hány logsor található meg a HDFS adatállományban:

```
{1: 1723232,       13: 120036,      25: 65,          37: 5,
 2: 575061,        14: 67,          26: 34,          38: 15,
 3: 1706728,       15: 19,          27: 16,          39: 5,
 4: 1706514,       16: 3226,        28: 7,           40: 10,
 5: 1719741,       17: 1402047,     29: 6,           41: 4,
 6: 7097,          18: 1464,        30: 9,           42: 47,
 7: 6937,          19: 75,          31: 9,           43: 16,
 8: 165,           20: 75,          32: 1396174,     44: 5,
 9: 165,           21: 975,         33: 5545,        45: 1,
 10: 428726,       22: 22,          34: 1288,        46: 10,
 11: 6837,         23: 33,          35: 356207,      47: 3,
 12: 6837,         24: 56,          36: 11,          48: 2}
```

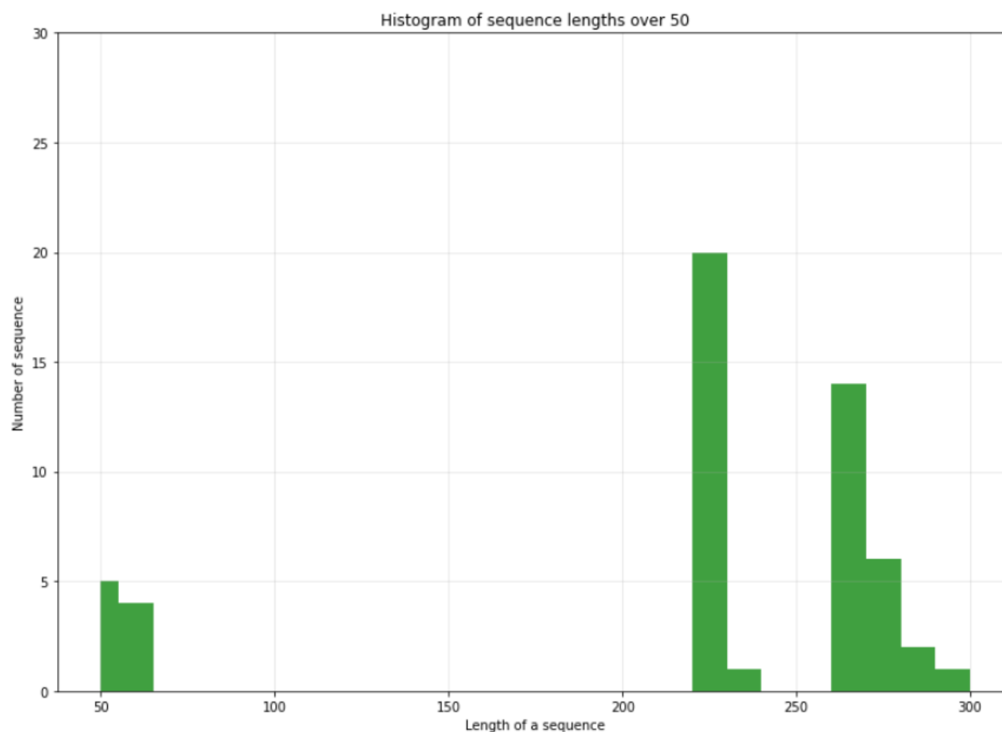
Az adatállomány összesen 11.175.629 logsorból áll, amelynek a felosztása feljebb található. Megfigyelhető, hogy a template-k elosztása eléggé változatos, egyáltalán nem függ az azonosító értékétől a template gyakorisága.

6.4. Szekvenciák hossza

A dolgozatomban számos helyen említettem, hogy a szekvenciáim változó hosszúságúak és ez számos helyen nehezítőtényezőként jelent meg. Az alábbiakban hisztogram segítségével bemutatom a HDFS egyes blockjaihoz tartozó szekvenciák hosszát. Két hisztogram lesz látható, az elsőben az 50 hosszúság alatt található szekvenciákat mutatom be, amely intervallumba a legtöbb szekvencia tartozik. Majd a többi szekvencia a nagy szórás miatt egy külön hisztogramban lesz bemutatva:



A képen diagrammon jól látszik, hogy a legtöbb szekvencia hossza 20 köré esik, egy-egy „kilógó” szekvencia van 0 és 10 között és egy pár van 30 felett is. Figyeljük meg, hogy a számosság 0 és 300.000 közötti értéket vesznek fel, tehát az alacsony hosszúságú szekvenciák több ezer darabot jelent. Most vizsgáljuk meg az 50 hosszúság felett található szekvenciákat:



A diagram indikálja, hogy ezeknek a száma jóval alacsonyabb az összes szekvencia számához. (Összesen 575001 szekvencia (tehát blokk is) található a HDFS adatállományban és ebből 57 ötven feletti hosszúságú, ez megközelítőleg 0,0099%). ***Így amikor a DeepLog-nál 50-re szabtuk meg a szekvencia hosszát és minden feletti értéket anomáliának értékeltünk, akkor az a diagrammok alapján egy helyes lépésnek bizonyult (Természetesen az eredmények is alátámasztják ennek a küszöbnek a helyességét).***

7. Értékelés

7.1. Általánosan az értékelés folyamatáról

Mind a kettő algoritmus esetén a tesztadathalmaz segítségével megtörtént a predikció. A megfelelő eredmény-threshold segítségével kiértékeljük, hogy az adott érték melyik kategóriába tartozik, és így létrehozzuk a tényleges eredményhalmazt: Ha a megfelelő threshold alapján az adott szekvencia anomáliás, akkor azt 1-gyel, ha normális akkor 0-val jelöljük. A helyes eredményt a HDFS címke által előállított anomaly_label.csv fájl tartalmazza. Ezt beolvasva megkapjuk, hogy melyik block anomáliás és melyik nem. Ezekután jöhet a predikált és a helyes eredmény összehasonlítása.

Az algoritmusokat pontosság alapján fogom összehasonlítani, ezért nézzük meg az ehhez szükséges metrikát.

Fontosnak tartom azt is, hogy célravezetőnek bizonyult a programozás során néhány részeredményt kimenteni egy fájlba, amelynek 2 nagyon fontos oka és szerepe van:

1) Sokkal gyorsabb egy fájlt beolvasni, mint újra lefuttatni a hatalmas logfájlban a keresést
2) Különböző adathalmazok esetén pedig így tudunk mindig ugyanazzal az adathalmazzal tanítani, validálni és tesztelni, ami a legjobb megoldás kifejlesztésének folyamatánál elengedhetetlen.

Ennek segítségével is szignifikáns gyorsítást tudtam elérni az algoritmus futtatásában.

7.2. Összehasonlító metrika

Az összehasonlítás elvégzését az F1 score segítségével tettem meg. Az F1-score alapvetően 2 érték kombinációjából tevődik össze:

1) **Precision:** A precision érték meghatározza, hogy hány helyesen pozitívnak ítélt érték van az összes pozitív predikció közül.
2) **Recall:** A recall megadja, hogy hány helyesen pozitívnak ítélt érték van az összes helyes érték közül.

helyes eredmeny \ előrejelzett eredmeny		Anomália = Igen	Anomália = Nem
helyes eredmeny	Anomália = Igen	True Positive	False Negative
	Anomália = Nem	False Positive	True Negative

17. ábra

Az 17. ábrán látható táblázat és képlet vizuálisan szemlélteti az F1-score alapfogalmait és kiszámítási módjukat:

True Positive (TP): Megadja, hogy hány olyan elem van, amelyet helyesen anomáliának jeleztünk

False Positive (FP): Megadja, hogy hány olyan elem van, amelyet helytelenül anomáliának jeleztünk

False Negative (FN): Megadja, hogy hány olyan elem van, amelyet helytelenül nem jeleztünk anomáliának

True Negative: (TN): Megadja, hogy hány olyan elem van, amelyet helyesen nem jeleztünk anomáliának.

Ezekután vizsgáljuk meg, hogy a Precision és a Recall és az ezekből meghatározható F1-score hogyan számolható ki (definiálunk még egy változót is, az Accuracy-t, ami, habár nem szerepel az F1-score képletében, viszont a helyesen megjósolt eredményekről ad egy jó és szemléletes képet):

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

$$Precision = \frac{TP}{TP + FP}$$

$$Recall = \frac{TP}{TP + FN}$$

$$F1 - score = \frac{2 * Precision * Recall}{Precision + Recall}$$

Miután bevezettük az F1-score-t vizsgáljuk meg, hogy milyen eredmények is születtek az egyes algoritmusok esetén.

7.3. DeepLog értékelés

7.3.1. Pontosság

Vizsgáljuk meg a DeepLog modell validációs adathalmazra történő predikciójának és a helyes kimenetnek az F1 score-ját és a hozzá tartozó értékeket (18. ábra):

Threshold	Accuracy	Precision	Recall	F1 Score
0.0000200	0.9965561	0.9899947	0.8916291	0.9382408
0.0000710	0.9969597	0.9823890	0.9127342	0.9462815
0.0001221	0.9969597	0.9725000	0.9224567	0.9468176
0.0001731	0.9970153	0.9671929	0.9298079	0.9481320
0.0002241	0.9972101	0.9660479	0.9378705	0.9517507
0.0002752	0.9974188	0.9622596	0.9492530	0.9557121
0.0003262	0.9975719	0.9557022	0.9618212	0.9587519
0.0003772	0.9973841	0.9452817	0.9668010	0.9559203
0.0004283	0.9970431	0.9324818	0.9694095	0.9505871
0.0004793	0.9969875	0.9226988	0.9793692	0.9501898
0.0005303	0.9968205	0.9177778	0.9793692	0.9475737
0.0005814	0.9967927	0.9169627	0.9793692	0.9471391
0.0006324	0.9968275	0.9161319	0.9817406	0.9478022
0.0006834	0.9967996	0.9153217	0.9817406	0.9473684
0.0007345	0.9967927	0.9138387	0.9834005	0.9473444
0.0007855	0.9967718	0.9130530	0.9836377	0.9470320
0.0008366	0.9967370	0.9120493	0.9836377	0.9464917
0.0008876	0.9967301	0.9116678	0.9838748	0.9463960
0.0009386	0.9963544	0.9009772	0.9838748	0.9406030
0.0009897	0.9963405	0.9004122	0.9841119	0.9404034
0.0010407	0.9963196	0.8998265	0.9841119	0.9400838
0.0010917	0.9963126	0.8994583	0.9843491	0.9399909
0.0011428	0.9960761	0.8928802	0.9843491	0.9363862
0.0011938	0.9960691	0.8926882	0.9843491	0.9362806
0.0012448	0.9951090	0.8669591	0.9843491	0.9219323
0.0012959	0.9950951	0.8665971	0.9843491	0.9217275
0.0013469	0.9950742	0.8660547	0.9843491	0.9214206
0.0013979	0.9950394	0.8651521	0.9843491	0.9209096
0.0014490	0.9949977	0.8640716	0.9843491	0.9202971
0.0015000	0.9949768	0.8635323	0.9843491	0.9199911

18. ábra

A 18. ábrán látható, hogy milyen F1-score értékek tartoznak az egyes thresholdokhoz. Jelen esetben a thresholdot 0-tól 0.0015-ig vizsgáltam, mivel ez az intervallum adta a legpontosabb értékeket. Pirossal jelöltem a maximális F1-score-hoz tartozó sort, amelyről leolvasható, hogy a maximális f1_score a **0.9587519, ami 95.8%-os DeepLog pontosságot jelent** és a hozzá tartozó threshold a **0.0003262**, amelyet a tesztelésnél fogunk használni. Ezek alapján nézzük meg, hogy az új, még korábban nem látott tesztadatra a modell milyen eredményeket ér el:

Threshold	Accuracy	Precision	Recall	F1 Score
0.0003262	0.9974816	0.9557416	0.9575743	0.9566571

19. ábra

Látható, hogy a 13. ábrán bemutatott maximális threshold-hoz az új, még nem látott adathalmazon történő lefutás esetén **95,66%**-os a pontosság. Az utahi egyetem a DeepLog eredménye HDFS logfájlokra 96% [17], azaz sikerült teljes mértékben reprodukálni a cikk eredményeit.

7.4. KNN értékelés

7.4.1. Pontosság

Kezdetben vizsgáljuk meg, hogy milyen eredményt tudunk elérni abban az esetben, ha az egyes klasszikus Levenshtein távolságmetrikákat alkalmazzuk, majd bemutatom a saját súlyozással ellátott metrika által elért eredménybeli javulást. Az ábrákon az adott metrikával kapott F1-score és a hozzá tartozó paraméterek eredményeit láthatjuk. Minden esetben piros téglalappal kiemeltem a legmagasabb F1-score értékhez tartozó sort. Fontos, hogy ezek az eredmények k = 12 paraméterválasztással adódtak.

Levenshtein-distance metrikával számolt távolság esetén az alábbi eredményeket kaptam:

Threshold	Accuracy	Precision	Recall	F1 Score
0.00	0.94	0.28	1.00	0.44000
1.00	0.96	0.31	0.70	0.42759
2.00	0.99	0.78	0.66	0.71605
3.00	0.99	0.93	0.64	0.75676
4.00	0.99	1.00	0.64	0.77778
5.00	0.99	1.00	0.64	0.77778
6.00	0.99	1.00	0.61	0.76056
7.00	0.99	1.00	0.50	0.66667
8.00	0.99	1.00	0.45	0.62500
9.00	0.99	1.00	0.45	0.62500
10.00	0.98	1.00	0.25	0.40000
11.00	0.98	1.00	0.07	0.12766
12.00	0.98	1.00	0.07	0.12766
13.00	0.98	1.00	0.07	0.12766
14.00	0.98	1.00	0.07	0.12766
15.00	0.98	1.00	0.07	0.12766

Az ábrán látható a validációs adathalmazon lefuttatott algoritmus eredménye. A maximális F1-score érték, amit 4-es küszöbérték mellett kaptam **77,77% lett**. A korábban látott DeepLog eredményét meg sem közelíti, így vizsgáljuk meg, hogy hátha más távolságmetrika bevezetésével javíthatunk az eredményen.

Levenshtein-Jaro metrikával számolt távolság esetén az alábbi eredményeket kaptam:

Threshold	Accuracy	Precision	Recall	F1 Score
0.00	0.98	0.00	0.00	0.00000
0.03	0.98	0.00	0.00	0.00000
0.07	0.98	0.00	0.00	0.00000
0.10	0.98	0.00	0.00	0.00000
0.14	0.98	0.00	0.00	0.00000
0.17	0.98	0.00	0.00	0.00000
0.21	0.98	0.00	0.00	0.00000
0.24	0.98	0.00	0.00	0.00000
0.28	0.98	0.00	0.00	0.00000
0.31	0.98	0.00	0.00	0.00000
0.34	0.98	0.00	0.00	0.00000
0.38	0.98	0.00	0.00	0.00000
0.41	0.98	0.00	0.00	0.00000
0.45	0.98	0.00	0.00	0.00000
0.48	0.98	1.00	0.02	0.04444
0.52	0.99	1.00	0.41	0.58065
0.55	0.92	0.11	0.41	0.17822
0.59	0.79	0.06	0.55	0.10458
0.62	0.31	0.02	0.73	0.04420
0.66	0.05	0.02	1.00	0.04407
0.69	0.02	0.02	1.00	0.04305
0.72	0.02	0.02	1.00	0.04305
0.76	0.02	0.02	1.00	0.04305
0.79	0.02	0.02	1.00	0.04305
0.83	0.02	0.02	1.00	0.04305
0.86	0.02	0.02	1.00	0.04305
0.90	0.02	0.02	1.00	0.04305
0.93	0.02	0.02	1.00	0.04305
0.97	0.02	0.02	1.00	0.04305
1.00	0.02	0.02	1.00	0.04305

A maximális1-score értéket 0.52-es küszöbértékkel kaptam, ami **58,065%** lett. Ez a metrika önmagában több mint 19%-os csökkenés produkált a hatásfokot tekintve csak a távolságmetrika megváltoztatása miatt. Ezek alapján belátható, hogy a távolságmetrika hihetetlen jelentőséggel bír. A *Levenshtein-Hamming* távolság a mi esetünkben nem használható, hiszen az csak azonos méretű szekvenciákra alkalmazható, ezért próbáljuk ki a saját súlyokkal ellátott metrikát:

weighted-Levenshtein metrikával számolt távolság esetén az alábbi eredményeket kaptam:

Threshold	Accuracy	Precision	Recall	F1 Score
0.00	0.94	0.27	1.00	0.43
1.00	0.95	0.34	1.00	0.51
2.00	0.99	0.79	1.00	0.88
3.00	1.00	0.96	0.96	0.96
4.00	1.00	1.00	0.96	0.98
5.00	1.00	1.00	0.96	0.98
6.00	1.00	1.00	0.91	0.95
7.00	1.00	1.00	0.83	0.90
8.00	0.99	1.00	0.78	0.88
9.00	0.99	1.00	0.78	0.88
10.00	0.99	1.00	0.78	0.88
11.00	0.99	1.00	0.74	0.85
12.00	0.99	1.00	0.70	0.82
13.00	0.99	1.00	0.70	0.82
14.00	0.99	1.00	0.70	0.82
15.00	0.99	1.00	0.70	0.82

Látható, hogy a weighted-Levenshtein az általam bevezetett súlyozással szignifikánsan jobb eredményeket produkál, ami jelen esetben **0.98-as F1-score értéket jelent, azaz 98%-os a pontosság, ha 4-es küszöbértéket veszünk figyelembe. Ez önmagában 21%-os javulást jelent, a súlyozatlan Levenshteinnel kapott eredményekhez képest!**

Threshold	Accuracy	Precision	Recall	F1 Score
4.00	1.00	0.98	0.98	0.98333

Az optimálisnak kapott 4-es küszöbérték felhasználásával vizsgáljuk meg, hogy milyen eredményt ad számunkra az algoritmus egy még korábban nem látott tesztadathalmaz esetén: Ahogy azt az ábra is jelzi, a tesztadathalmazon történő futtatás következtében és a korábban kapott küszöbértéket felhasználva **98.33%-os pontosság elérését tettem lehetővé, mindössze a távolságmetrika helyes megválasztásával!**
A bevezetőben említettem, hogy ezek az értékek k = 12 paraméter megválasztásával adódtak. Ennek az az oka, hogy tapasztalataim szerint ezzel a 12-es értékkel sikerült a legjobb eredményt elérni. (Más k paraméterértből kapott eredményt és különböző kiértékelési adatállományokat az alábbi git repository-ban lehet megtekinteni:

https://github.com/charafkamel/TDK_2022.git

8. Konklúzió:

Összefoglalva, a munkám alapvetően 2 nagy részből tevődött össze. Az első rész a DeepLog saját implementációját tartalmazza, ami eredményét tekintve megegyezik az irodalomban lévőkkel, annak ellenére, hogy egyszerűsítettem az algoritmust, mivel a 9 valószínűség helyett mindössze a dedikált kimenethez tartozó valószínűséget vettem figyelembe. Ezek alapján nyugodtan állíthatom, hogy egy egyszerűbb és gyorsabb implementációt adtam változatlan pontosság mellett. A második részben KNN alapú megközelítést követtem, de a meglévő megoldásokkal ellentétben nem az attribútum-anomáliákat, hanem a szekvenciális anomáliákat detektálom. Ez a megközelítés egy újszerű megoldás és legalább olyan pontosságot produkált, mint a meglévő KNN alapú algoritmusok. A KNN alapú megoldásommal kapcsolatban az egyik legkritikusabb feladat a megfelelő távolságmetrika azonosítása és használata volt. A dolgozatomban felsoroltam és megmutattam ezek használatával az algoritmus teljesítményét. Láthattuk, hogy ezen távolságmetrikákkal lényegesen gyengébb eredmények születtek, míg a saját súlyozással súlyozott metrikával több mint 21%-os javulást értem el, ami új tudományos eredménynek minősül.

Köszönetnyilvánítás:

Itt szeretném kifejezni köszönetem a konzulensemnek Dr. Horváth Gábornak, akinek a segítsége nélkül nem jöhetett volna létre ez a munka.

9. Irodalomjegyzék

[1] Chandola, Varun, Arindam Banerjee, and Vipin Kumar. "Anomaly detection: A survey." ACM computing surveys (CSUR) 41.3 (2009): 1-58.

[2] Wang, Bingming, Shi Ying, and Zhe Yang. "A log-based anomaly detection method with efficient neighbor searching and automatic K neighbor selection." Scientific Programming 2020 (2020).

[3] https://www.inf.u-szeged.hu/~rfarkas/ML20/MLintro.html

[4] Hullám Gábor, Salánki Ágnes, Kocsis Imre: Gépi tanulási módszerek alkalmazása, https://inf.mit.bme.hu/sites/default/files/materials/taxonomy/term/446/16/BD_ML_modszerek_HG_jav.pdf

[5] Jia, W., Shukla, R. M., & Sengupta, S. (2019, December). Anomaly detection using supervised learning and multiple statistical methods. In 2019 18th IEEE International Conference On Machine Learning and Applications (ICMLA) (pp. 1291-1297). IEEE.

[6] Intrusion detection using clusters. In Proceedings of the Twenty-eighth Australasian Conference on Computer Science - Volume 38, ACSC ’05, page 333–342, Darlinghurst and Australia, 2005. Australian Computer Society, Inc. ISBN 1-920-68220-1.

[7] Chapelle, Olivier, Bernhard Schölkopf, and Alexander Zien. "Semi-supervised learning. Adaptive computation and machine learning series." (2006).

[8] Lou, Jian-Guang, et al. "Mining invariants from console logs for system problem detection." 2010 USENIX Annual Technical Conference (USENIX ATC 10). 2010.

[9] https://de.wikipedia.org/wiki/Singul%C3%A4rwertzerlegung

[10] Xu, Wei, et al. "Detecting large-scale system problems by mining console logs." Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles. 2009.

[11] https://www.analyticsvidhya.com/blog/2021/03/introduction-to-long-short-term-memory-lstm/

[12] [GitHub - logpai/logparser: A toolkit for automated log parsing [ICSE'19, TDSC'18, ICWS'17, DSN'16]

[13] IPLoM — logparser 0.1 documentation

[14] LKE — logparser 0.1 documentation

[15] Spell — logparser 0.1 documentation

[16] Drain — logparser 0.1 documentation

[17] Du, Min, et al. "Deeplog: Anomaly detection and diagnosis from system logs through deep learning." Proceedings of the 2017 ACM SIGSAC conference on computer and communications security. 2017.

[18] datacomp.com

[19] https://maxbachmann.github.io/Levenshtein/

[20] https://www.geeksforgeeks.org/jaro-and-jaro-winkler-similarity/

[21] Levenshtein, Vladimir I. "Binary codes capable of correcting deletions, insertions, and reversals." Soviet physics doklady. Vol. 10. No. 8. 1966.

[22] Jaro, Matthew A. "Advances in record-linkage methodology as applied to matching the 1985 census of Tampa, Florida." Journal of the American Statistical Association 84.406 (1989): 414-420.