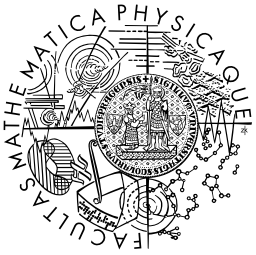


MATEMATICKO-FYZIKÁLNÍ
FAKULTA
Univerzita Karlova

DIPLOMOVÁ PRÁCE

Jan Babušík

Detekce anomalit v log datech

Katedra teoretické informatiky a matematické logiky

Vedoucí diplomové práce: Mgr. Marta Vomlelová, Ph.D.

Studijní program: Informatika

Studijní obor: Umělá inteligence

Praha 2021

Prohlašuji, že jsem tuto diplomovou práci vypracoval(a) samostatně a výhradně s použitím citovaných pramenů, literatury a dalších odborných zdrojů. Tato práce nebyla využita k získání jiného nebo stejného titulu.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona v platném znění, zejména skutečnost, že Univerzita Karlova má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle §60 odst. 1 autorského zákona.

V ..... dne ....

Podpis autora

Tímto bych chtěl poděkovat vedoucí práce, Mgr. Martě Vomlelové, Ph.D. za čas a ochotu řešit problémy. Také bych rád poděkoval své přítelkyni Kateřině Červinkové, rodičům a všem přátelům, kteří mi byli podporou na této cestě. Speciální poděkování patří Ing. Jiřímu Kandovi z firmy HAVIT, s.r.o. za konzulzace a poskytnutí reálně používaných dat pro zpracování.

Název práce: Detekce anomalit v log datech

Autor: Jan Babušík

Katedra: Katedra teoretické informatiky a matematické logiky

Vedoucí diplomové práce: Mgr. Marta Vomlelová, Ph.D., Katedra teoretické informatiky a matematické logiky

Abstrakt: Tato práce se zabývá detekováním anomalit v log datech. Velké sotfwarové systémy produkují velké množství logů, které nejsou dále zpracovávány. Logů je zpravidla tolik, že není možné ručně kontrolovat každý záznam. V této práci předkládáme modely, které minimalizují především počet falešně pozitivních hlášení s přihlédnutím k očekávané složitosti anotování dat. Porovnávané modely jsou založeny na PCA algoritmu, N-gramech a rekurentních neuronových sítích s použitím LSTM buňky. V práci prezentujeme výsledky modelů na standardně používaných datasetech i reálných datech poskytnutých firmou HAVIT, s.r.o.

Klíčová slova: Log data Detekce anomalit Strojové učení Neuronové sítě LSTM PCA

Title: Anomaly Detection on Log Data

Author: Jan Babušík

Department: Department of Theoretical Computer Science and Mathematical Logic

Supervisor: Mgr. Marta Vomlelová, Ph.D., Department of Theoretical Computer Science and Mathematical Logic

Abstract: This thesis deals with anomaly detection of log data. Big software systems produce a great amount of log data which are not further processed. There are usually so many logs that it becomes impossible to check every log entry manually. In this thesis we introduce models that minimize primarily count of false positive predictions with expected complexity of data annotation taken into account. The compared models are based on PCA algorithm, N-gram model and recurrent neural networks with LSTM cell. In the thesis we present results of the models on widely used datasets and also on a real dataset provided by HAVIT, s.r.o.

Keywords: Log Data Anomaly Detection Machine Learning Neural Networks LSTM PCA

Obsah

Úvod **3**

1 Související práce **5**
1.1 DeepLog . . . 5
1.1.1 Parsování . . . 5
1.1.2 Architektura . . . 6
1.1.3 Parametry modelu . . . 8
1.2 LogRobust . . . 9
1.2.1 Architektura . . . 9

2 Umělé neuronové sítě **12**
2.1 Perceptron . . . 12
2.2 Vrstevnaté neuronové sítě . . . 13
2.3 Učení neuronových sítí . . . 14
2.3.1 Učení parametrů . . . 15
2.3.2 Učení architektury . . . 16
2.4 Hluboké sítě . . . 16
2.5 Rekurentní a konvoluční sítě . . . 17
2.5.1 LSTM . . . 18

3 Datasety **20**
3.1 HDFS log dataset . . . 20
3.2 OpenStack log dataset . . . 21
3.3 HAVIT dataset . . . 22

4 Popis modelů **24**
4.1 Hladový model . . . 24
4.2 PCA model . . . 24
4.2.1 Popis PCA modelu . . . 25
4.3 N-gram model . . . 26
4.4 LSTM model . . . 27
4.4.1 Příprava dat . . . 27
4.4.2 Architektura . . . 27
4.4.3 Detekce . . . 28

5 Vyhodnocení **29**
5.1 HDFS dataset . . . 29
5.2 OpenStack dataset . . . 31
5.3 HAVIT dataset . . . 32
5.4 Výhody a nevýhody jednotlivých modelů . . . 33

Závěr **34**
Možná rozšíření . . . 36

Seznam použité literatury **37**

Seznam obrázků **40**

Seznam tabulek **41**

Seznam použitých zkratek **42**

A Přílohy **43**
A.1 HDFS log dataset 43
A.2 OpenStack log dataset 44

Úvod

Logování je jeden z nástrojů pro monitorování chování softwarových systémů, který je pro svou jednoduchost velice populární. Logy jsou většinou částečně strukturované textové řetězce obsahující různé informace o chování systému. Dokud jsou systémy malé, je programátor schopen tyto logy manuálně projít a prověřit stav sytému, ale už pro střední aplikace je tento úkol příliš časově náročný. Zprvu užitečný nástroj je tak často přehlížen pro obrovské množství dat a informace, které by mohly pomoci při řešení problému, zůstanou neobjevené.

Situace, kdy máme hodně dat, ale nemáme čas procházet data manuálně, vyžaduje nějaké automatizované řešení. Zde se nabízí využití některých nástrojů umělé inteligence. Oblíbeným přístupem, jak analyzovat logy, je hledání tzv. anomálií. Anomálii je možné definovat jako pozorování, které se natolik liší od ostatních pozorování, že vyvolává podezření, že bylo generováno jiným mechanismem (Hawkins, 1980).

Je zde ale několik problémů, které je potřeba mít na paměti. Zaprvé jsou logovací zprávy často nijak nestrukturované a v přirozeném jazyce a je třeba řešit, jakým způsobem logy připravit pro další zpracování. Dalším problémem je, že do jednoho souboru může zapisovat více procesů nebo vláken a samotné procesy mohou být implementovány paralelně. Záznamy tudíž nemusí být sekvenčně seřazené. Některé nástroje pro logování logy dokonce vzorkují, aby snížily jejich množství, tím však ještě více rozbíjejí jejich sekvenční charakter.

Jedním z možných řešení je tzv. rule-based přístup, kdy znalec konkrétního systému vytvoří nějaký program nebo protokol, podle kterého se logy třídí. Problémem tohoto přístupu je, že znalec musí detailně rozumět danému systému, jenž se navíc vyvíjí. S vývojem se mění i charakter a četnost logů a se změnami v systému je třeba měnit také monitorovací program. To časem skončí jeho zastaráním a takový program pak jen generuje nové logy, které budou znovu zapomenuty v množství neužitečných informací.

Druhou možností je použít strojového učení. Pokud použijeme učení s učitelem, je třeba získat dostatečné množství anotovaných dat. Výhodou tohoto přístupu je, že je možné model navrhnout tak, aby mohl být iterativně upravován a je jednoduší jej udržovat aktuální vůči systému, který je monitorován. Taková řešení lze rozdělit do tří tříd podle toho, jestli máme k dispozici anotovaná, částečně anotovaná nebo neanotovaná data (Ajith, 2019). Pokud je možné získat plně anotovaná data, lze vytvořit prediktivní model pomocí učení s učitelem, který bude přímo klasifikovat data do tříd. Dostatečné množství plně anotovaných dat je ovšem velice obtížné a časově náročné získat, proto podobné modely často nemají reálné využití.

Opačným přístupem je pokusit se využít učení bez učitele. Při použití této možnosti se obvykle předpokládá, že anomálií je málo, protože systém se chová většinou normálně. Odpadá tedy nutnost anotací dat. Takový systém se zdá být nejideálnější možností, protože je možné jej aplikovat na téměř jakoukoli množinu logů a měl by z ní dostat užitečné informace, ale takové modely bývají často příliš obecné (Du a kol., 2017).

Částečným kompromisem může být anotování pouze normálních dat, protože získat taková data je obvykle daleko snazší. Při detekci může být každý případ

klasifikován podle toho, jak moc se podobá viděným normálním příkladům. Je však třeba externě zvolit hodnotu, od které se příklady klasifikují jako anomální.

Jedena z klíčových vlastností systému, který detekuje anomálie v log datech, je nízký výskyt falešně pozitivních hlášení. Někomu se může zdát, že důležitější vlastností je zachytit všechny pozitivní případy. Tuto úlohu už ovšem dělají samotné logy a nastavení příliš jemného síta způsobí stejný problém, jenž se snažíme řešit.

Cílem této práce je prozkoumat existující přístupy zpracování log dat, na základě těchto informací navrhnout vlastní řešení a poté se pokusit toto řešení vyzkoušet na vlastních reálných datech.

1. Související práce

Problém hledání anomálií v log datech již byl v minulosti několikrát zpracováván, zde si uvedeme dva populární přístupy z posledních let, které daný problém řeší především pomocí neuronových sítí. Dostupné datasety nejčastěji spojují logy do posloupností, jež byly vygenerovány stejným procesem. Tyto posloupnosti se často označují jako bloky. Autoři zde uvedených prací se snaží právě tyto bloky klasifikovat.

1.1 DeepLog

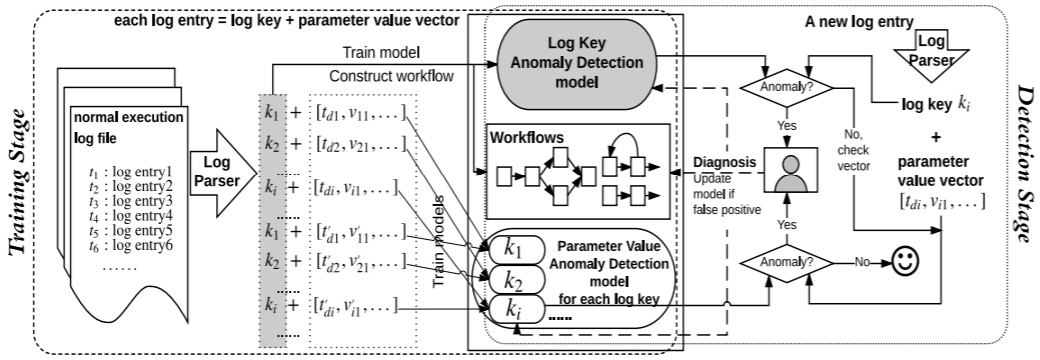
Jednou z nejzajímavějších prací na toto téma je *DeepLog: Anomaly Detection and Diagnosis from System Logs through Deep Learning* (Du a kol., 2017). Jako cíl si autoři zvolili vytvořit model, který bude schopen analyzovat anomálie v lozích obecně bez předchozí znalosti systému, nebude vyžadovat anotaci negativních případů a bude mít možnost online úpravy modelu. Protože je třeba systémové logy vnímat jako časovou řadu, rozhodli se autoři použít rekurentních neuronových sítí pro modelování vysoce dimenzionálních závislostí mezi jednotlivými log záznamy. Nakonec se ještě pokusí uživateli usnadnit diagnostiku pomocí rozdělení logů na jednotlivé úlohy, které pak analyzují pomocí stavového automatu.

1.1.1 Parsování

Nejprve je třeba převést nestrukturovaný text logů do strukturované reprezentace, kterou by model uměl zpracovat. Efektivní metodou, jak toho dosáhnout, je extrahovat tzv. *log key*, neboli typ zprávy, který přibližně odpovídá textovému řetězci bez hodnot proměnných parametrů, jež jsou do něj dosazeny. Skutečný log může vypadat takto *"Took* 10 *seconds to build instance"*, kde klíč záznamu je *"Took * seconds to build instance"* a vektor parametrů je *[10]*. Tento string je ještě zakódován do celého čísla. Hodnoty parametrů mohou být poté ještě užitečné, například pro identifikaci procesu nebo bloku, a proto jsou také zakódovány do vhodného formátu a přiloženy ke klíči jako vektor parametrů.

Cílem této části je tedy převést nestrukturovaná data do formátu *klíč* a *vektor jeho parametrů*. Pro tento účel lze využít metodu *Spell* (Du a Li, 2016), což je metoda parsování logů bez učitele navržena stejnými autory, která zpracovává přicházející logy pomocí nejdelšího společného podřetězce.

DeepLog se od svých předchůdců zabývajících se podobnými tématy liší především tím, že k analýze záznamů nepoužívá jen klíč, ale také vektor parametrů jednotlivých logů. Tímto způsobem dokáže zachytit i anomálie, které se projeví jen v těchto parametrech, například prodloužená doba mezi záznamy nebo zvláštní velikosti souboru.



Obrázek 1.1: DeepLog architektura. Zdroj: (Du a kol., 2017)

Tabulka 1.1: Parsování zprávy na klíč a vektor parametrů

log záznam (klíč podržený)	id klíče	vektor parametrů
t1 Deletion of file1 complete	k1	1
2	7	78
3	545	778

1.1.2 Architektura

DeepLog architektura se skládá ze tří hlavních komponent:

1. Model pro detekci anomalit v sekvenci klíčů
2. Model pro detekci anomalit ve vektoru parametrů
3. Workflow model

Detekce anomalit probíhá následovně: záznam je převeden do formátu *klíč* a *vektor parametrů*. Poté se analyzuje sekvence klíčů prvním modelem, a pokud neodpovídá normálu, prověří druhý model sekvence vektorů s parametry. Pokud některý z modelů určí, že záznam je anomální, DeepLog jej označí jako anomální. Nakonec se použije *workflow model* pro získání kontextu anomálie, aby měl uživatel jednodušší práci při řešení potenciální chyby.

Klíče logů v podstatě odpovídají generickým textům z logovacích příkazů, jejich sekvence tedy odráží exekuční cestu ve zkoumaném systému, která vedla k těmto záznamům. Z těchto skutečností autoři usuzují, že každý klíč je závislý na některých předchozích klíčích. Konkrétně závisí na klíčích, které se nachází před ním na exekuční cestě dané úlohy. Například *VM creation* bude předcházet *VM deletion*. Detekce anomalit je pak modelovaná jako klasifikační problém, kde každý klíč závisí na nedávné historii jeho h předchůdců. Výstupem modelu je distribuce klíčů k ze všech možných klíčů. Tato distribuce je potom seřazena od nejpravděpodobnější hodnoty a klíč je označen jako normální, pokud se nachází mezi g nejpravděpodobnějšími hodnotami.

K získání distribuce klíčů lze použít klasický n-gramový model, známý především ze zpracování přirozených jazyků. Zde se využívá předpokladu, že slovo je ovlivněno pouze několika předchozími klíči. Distribuce se pak spočítá přímo z dat jako $P[m_t = k_i|m_{t-n},...,m_{t-1}]$ pomocí markovských řetězců vyšších řádů.

Problémem této metody je, že pro větší h se model stává výpočetně velmi náročný. Autoři článku se tak rozhodli použít rekurentní neuronové sítě, konkrétně LSTM, protože dokáží zachytit daleko komplexnější vzory a umožňuje efektivní zvětšení uvažovaného počtu předchůdců.

LogKey model

Samotný model pro detekci anomalit v sekvenci klíčů potom funguje následovně. Protože je předem znám počet klíčů K, můžeme použít *one-hot* kódování. Id klíče je zakódované do nulového vektoru délky K, který má jedničku právě jen na indexu k. Data jsou rozdělena do sekvencí délky h. Výstupem modelu je už zmíněná distribuce pravděpodobností. Model ve výchozím nastavení se skládá ze dvou LSTM vrstev, kde každá buňka má 64 neuronů a používá *kategorickou křížovou entropii* jako ztrátovou funkci pro výpočet chyby mezi výslednou distribucí a výstupním klíčem v *one-hot* kódování.

Parameter value model

LogKey Model je velmi užitečný pro hledání anomalit exekučních cest. Existují ale také anomálie, které se projevují jen jako odchylky od hodnot ve vektoru parametrů. Různé logy mohou mít různé počty i typy parametrů a je důležité najít vhodný způsob kódování těchto vektorů. Autoři článku se rozhodli pro kódování do vektoru velikosti $c = \sum_k size(k)$ (součet velikostí všech vektorů), obsazeny budou jen pozice náležející k danému klíči. Když uvážíme tři různé klíče a jejich vektory parametrů $[a, b], [c, d], [e]$, první vektor bude zakódovaný jako $[a, b, null, null, null]$, druhý jako $[null, null, c, d, null]$ a třetí jako $[null, null, null, null, e]$. Matice takových vektorů je velice řídká a je možné ji udělat hustší, pokud bude jeden řádek reprezentovat více záznamů, ale tento přístup vede ke zpoždění detekčního procesu.

Po této úpravě lze aplikovat podobný model jako na sekvence klíčů, který také používá velice podobnou LSTM vrstvu. Je však nutné změnit ztrátovou funkci na *střední kvadratickou odchylku*, protože se porovnává předpovězená hodnota vektoru s jeho skutečnými hodnotami a není důležitá jen maximální hodnota jako u předchozího modelu.

Důležitým rozdílem mezi přístupem k předpovídání klíčů a k předpovídání parametrů je, že u parametrů se pro každou hodnotu klíče vytváří zvláštní *Parameter value model*, zatímco pro předpovídání klíčů se používá model jediný. Detekce pak probíhá spočítáním *střední kvadratické odchylky* modelem předpovězené hodnoty od skutečné hodnoty, která musí být menší než t. Hraniční hodnota hlášení chyby t je učena tak, že se trénovací data rozdělí na trénovací data pro model a validační set. Na každý vektor ve validačním setu se aplikuje model a spočítá se *střední kvadratická odchylka* predikce a očekávané hodnoty. Z těchto hodnot algoritmus namodeluje Gaussovo rozdělení a při aplikaci modelu na nová data jsou pak data označena za normální, pokud se střední odchylka mezi daty a předpovědí nachází v intervalu vysoké jistoty toho rozdělení.

Online aktualizace

Možných korektních posloupností klíčů je velmi mnoho a není možné, aby byly v trénovacích datech zastoupeny úplně všechny. Většina velkých systému navíc nikdy nedospěje do stavu, kdy by se úplně zastavil vývoj, tudíž se chování systému v čase proměňuje. Z tohoto plyne potřeba upravovat model podle aktuálního chování systému.

Tato úprava probíhá, pokud uživatel ohlásí falešně pozitivní nález, tj. model ohlásí anomálii, ale uživatel rozumí tomu, že toto chování je normální. Toto se stalo, protože model se ještě nesetkal s tímto vzorem chování a je nutné mu jej předložit.

Pro samotnou aktualizaci modelu není nutné vytvářet model od začátku, proces aktualizace probíhá stejně jako v trénovací fázi.

Někomu se může zdát zvláštní, že se nijak neřeší falešně negativní příklady. Toto chování má několik příčin, především je jedním z cílů eliminovat co nejvíce falešných pozitiv a falešné negativa nejsou tolik zajímavé. Navíc tím, že parametr g, tj. počet nejpravděpodobnějších klíčů, mezi kterými se následující klíč musí nacházet, je fixní. Tím pádem se model může naučit jen konečné množství různých vzorů a předpokládá se, že pozitivní se v něm po několika epochách už nebudou nacházet.

Workflow Model

Exekuční cesta programem, která je zapsána v lozích při plnění nějakého dílčího cíle, může být použita k postavení konečného stavového automatu, který by přijímal jen normální běhy programem. Tento přístup sice není efektivnější než model využívající LSTM, avšak je pro uživatele daleko snadněji čitelný a umožňuje mu snadněji nahlédnout anomálii cíle. Pro tento přístup je potřeba nejprve rozdělit sekvenci logů na jednotlivé úkoly nebo cíle, které program vykonával, a pro každý takový cíl je zvlášť vytvořen *Workflow model*.

1.1.3 Parametry modelu

Algoritmus má několik parametrů, které mají vliv na jeho chování. Některé z nich by pro co nejlepší chování programu měly mít co nejvyšší hodnoty, současně však zvětšují výpočetní náročnost programu. Jejich velikost je limitována především dostupnou výpočetní technikou. Jsou tedy obvykle nastaveny na nejvyšší mez, která byla ještě relativně přínosná pro efektivitu algoritmu a zároveň jej příliš výkonnostně nezatěžovala. Mezi takové parametry patří především počet neuronů v jedné LSTM buňce (výchozí nastavení je 64) a počet rekurentních vrstev (výchozí nastavení jsou *dvě* vrstvy).

Parametry h (historie, velikost okénka pro vstup) a g (počet nejpravděpodobnějších předpovězených klíčů, mezi kterými se musí nacházet zkoumaný klíč, aby byl prohlášen za normální) jsou parametry velice závislé na konkrétních datech. Pokud se zvyšuje počet všech možných klíčů K, může se také zvyšovat počet navazujících klíčů, které by se daly považovat za normální. Pokud zafixujeme hodnotu K, se zvyšujícím se parametrem g se bude snižovat počet falešně pozitivních případů, což může být žádoucí efekt. Vyšší h umožňuje pracovat s delší historií, zvlášť pokud podúloha (sekvence klíčů, která se často opakuje napříč daty, může

odpovídat například sekvenci logů při vytváření VM apod.), obsahuje nějaké paralelní části, závislý log může být ve vzdálenější historii. Na druhou stranu nemá smysl, aby byl parametr větší než délka jednotlivých nezávislých úloh, což je maximální vzdálenost, pro kterou lze nalézt závislosti mezi klíči. Jeho velikost také zvyšuje výpočetní náročnost modelu. Výchozí hodnota parametrů je $g = 9$ a $h = 10$.

1.2 LogRobust

Další zajímavou prací je *Robust Log-Based Anomaly Detection on Unstable Log Data*. (Zhang a kol., 2019) Autoři se soustředí zejména na jeden problém, který znemožňuje praktické použití současných modelů, a to je nestabilita logovacích zpráv.

Nestability lze rozdělit na dva typy:

1. Zašuměná data - chyby, které vznikají při zpracovávání a parsování logů, ztráta dat, paralelnost, vzorkování apod.

2. Vývoj systému - změna stávajících a přidávání nových typů logů.

Většina současných přístupů předpokládá uzavřenost světa, tzn. že existuje konečné množství typů logovacích zpráv, tzv. *klíčů*. Reálné systémy bývají pod neustálým vývojem, který se týká i samotných logovacích záznamů. Podle (Kabinna a kol., 2018) se 20%-45% všech logovacích zpráv za životnost systému změnila. Přístupy, které uvažují uzavřenou množinu klíčů, potom mají velký problém, když se nějaká změna vyskytne, a buď se sníží úspěšnost modelu, nebo je nutné celý model přetrénovat, což může být časově velmi náročné a tudíž nepraktické v reálném vývoji.

LogRobust oproti tomu nezávisí na výskytu *klíče*, ale každou zprávu transformuje do *sémantického vektoru* o fixní velikosti pomocí analýzy přirozeného jazyka.

1.2.1 Architektura

Parsování

První část parsování probíhá podobně jako v *DeepLogu*, nestrukturovaný text logu je rozdělen do generického textu logu a jeho parametrů. K tomuto účelu autoři používají metodu Drain (He a kol., 2017), která má vysokou rychlost i přesnost. Přesto se může stát, že do dat zanese další šum.

Následuje fáze, kdy se takto připravený klíč převádí do formy *sémantického vektoru*. Záznam se zpracovává jako věta v přirozeném jazyce, což je pro většinu systémů angličtina. Nejprve se odstraní všechny netextové prvky zprávy jako znaménka, oddělovače, čísla a podobně. Potom se odstraní všechna nevýznamová slova, mezi něž patří například členy. Nakonec se rozdělí proměnné skládající se z více slov.

Nyní následuje fáze, kdy se vektorizují jednotlivá slova. Transformace klíčů musí splňovat dvě kritéria:

1. Dvě různé události, které mají dva různé klíče, musí mít různé vektory.

2. Změněné logy, které reprezentují stejnou událost, by měly mít podobné vektory.

K tomuto *LogRobust* používá předtrénované vektory pro jednotlivá slova z *Common Crawl Corpus* datasetu, který extrahuje sémantickou informaci pro slova z anglického slovníku. Tento slovník dokáže převést slovo na sémantický vektor o velikosti 300.

Agregace vektorů slov

Nyní je třeba převést list vektorů slov do jediného vektoru pro záznam. K tomu se autoři rozhodli využít váženého součtu vektorů jednotlivých slov, kde váha slova by měla odpovídat jeho důležitosti ve větě. K tomuto účelu se používá metrika TF-IDF. TF značí četnost daného slova v celém textu, protože se předpokládá, že má význam zachytit slova, která se v textu často vyskytují. Na druhou stranu, pokud se nějaké slovo vyskytuje v každé zprávě, nemá dostatečnou rozlišující vlastnost. K tomuto právě slouží druhá složka IDF, která značí poměr logů, které dané slov obsahují.

$$TF(slovo) = \frac{\text{četnost slova}}{\text{počet všech slov}}$$

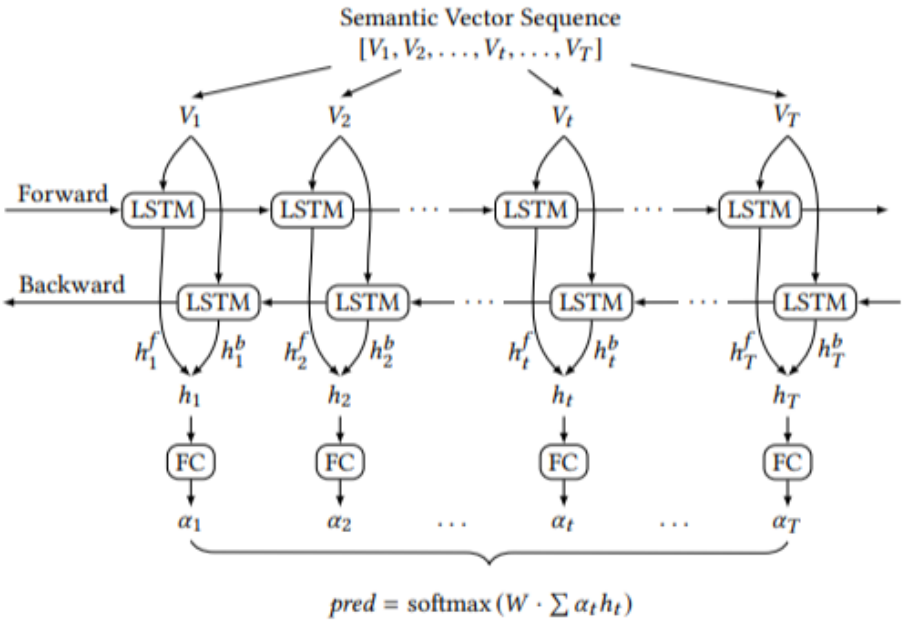
$$IDF(slovo) = log(\frac{\text{počet všech událostí}}{\text{počet událostí s tímto slovem}})$$

Celkový sémantický vektor se potom vypočte jako $V = \frac{1}{N}\sum_{i=1}^{N} w_i v_i$, kde w_i je TF*IDF slova a v_i je samotný sémantický vektor slova získaný z *Common Crawl Corpus* datasetu.

Tímto způsobem lze převést libovolný neznámý záznam do stejného 300 dimenzionálního prostoru, se kterým umí *LogRobust* model pracovat.

Klasifikace

Pro klasifikování sekvence sémantických vektorů se používá Bi-LSTM, obousměrná rekurentní neuronová síť. Protože každý log má odlišný efekt na klasifikaci, přidá se nakonec ještě jedna plně propojená vrstva, jež slouží jako vrstva důležitosti, která pro každý vektor spočítá koeficient důležitosti α. Ten se použije v vypočtení vážené sumy výstupů z Bi-LSTM vrstvy. Nakonec se přidá ještě jedna vrstva s aktivační funkcí *softmax*, jejíž výstup se použije pro klasifikaci. Architektura je znázorněna na obrázku 1.2.



Obrázek 1.2: LogRobust architektura. Zdroj: (Zhang a kol., 2019)

2. Umělé neuronové sítě

Neuronové sítě jsou technika strojového učení, která patří spolu s evolučními algoritmy do skupiny přírodou inspirovaných algoritmů. Inspirací pro umělé neuronové sítě bylo fungování lidského mozku, konkrétně jednotlivých neuronů. Typický neuron je složen z několika dendritů, které přijímají vstupní informaci, a jednoho axonu, jenž vytváří výstup. Člověk má asi 10^{11} neuronů (Azevedo a kol., 2009) a celkem se v mozku nachází až 10^{15} spojení neboli synapsí mezi neurony (Koch, 2004). Právě vznikem nových synapsí probíhá učení člověka.

Neuronová síť se snaží napodobit funkci lidského mozku pomocí umělých neuronů a spojení mezi nimi (Dongare a kol., 2012). Tímto způsobem může neuronová síť řešit složité klasifikační a regresní úlohy.

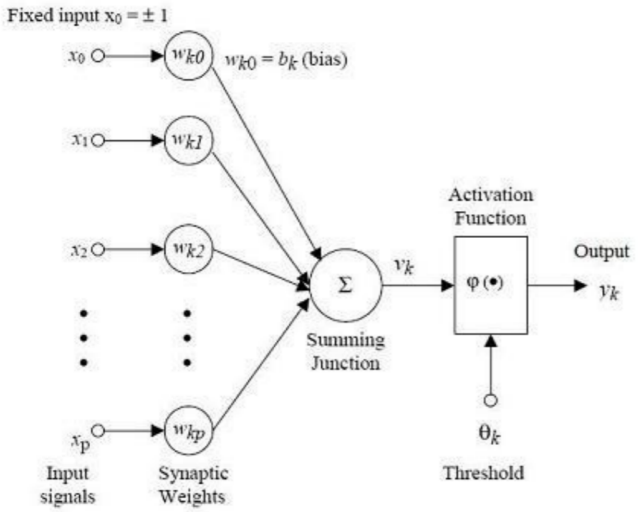
Metody strojového učení se dělí na učení s učitelem a bez učitele. Učení s učitelem znamená, že trénovací data se budou skládat ze vstupních objektů a jim odpovídajícím očekávaným výstupům. V některých případech však není jednoduché očekávané výstupy sehnat, proto se pak využívá alternativní způsob učení bez učitele, kde se trénovací data skládají pouze ze vstupních objektů a učení spočívá v hledání zákonitostí v datech. Neuronové sítě jsou příkladem metody učení s učitelem i bez učitele, ale v této práci jsme používali učení s učitelem.

Problém hledání anomálií v lozích nemusí mít jasně definované očekávané výstupy, v takovém případě je třeba data předzpracovat odvozením očekávaných výstupů ze samotných dat, aby se neuronová síť mohla učit i bez nich.

2.1 Perceptron

Základním stavebním prvkem neuronové sítě je umělý neuron, který se nejčastěji definuje jako $y = \varphi(\sum_i x_i w_i + b)$, kde φ značí aktivační funkci, x vektor vstupů, y výstup a w jsou synaptické váhy spojení se vstupními neurony. Aktivační funkce je funkce, kterou neuron aplikuje na váženou sumu vstupů. Nejjednodušší neuronovou sítí je *Perceptron*, jenž se skládá z právě jednoho umělého neuronu, takže má n vstupů a jeden výstup, a jehož úkolem je klasifikovat vstupní data do dvou skupin (Rosenblatt, 1957). Grafické znázornění perceptronu vidíme na obrázku 2.1. Perceptron data klasifikuje následujícím způsobem:

$$y = \begin{cases} 0 & \text{pro } \varphi(\sum_i x_i w_i + b) > 0, \\ 1 & \text{jinak.} \end{cases}$$



Obrázek 2.1: Perceptron. Zdroj: (Dongare a kol., 2012)

Základními kritérii, které klademe na aktivační funkci, jsou nelinearita, a rychlý výpočet funkce a její derivace. Mezi nejčastěji používané aktivační funkce patří *sigmoid*, *tanh* a *ReLU*.

$$\sigma(x) = \frac{1}{1+e^{-\lambda x}} \qquad \sigma'(x) = \sigma(x)(1-\sigma(x))$$

$$\tanh(x) = \frac{2}{1+e^{-\lambda x}} - 1 \qquad \tanh'(x) = 1 - \tanh(x)^2$$

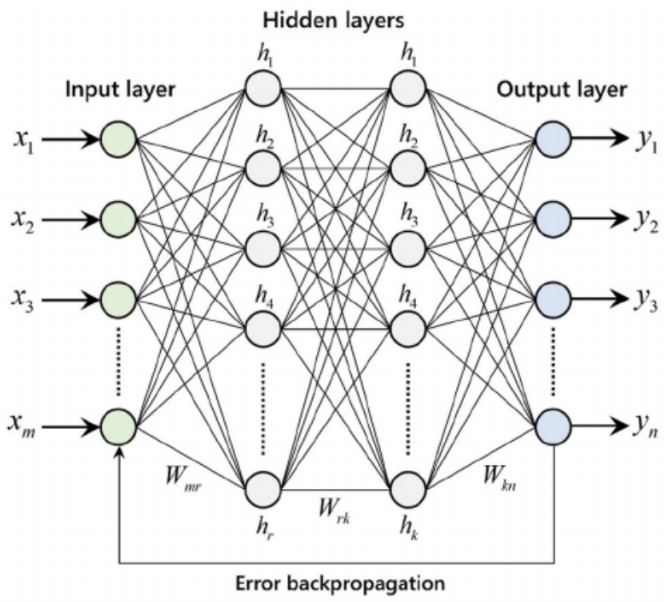
$$ReLU(x) = max(0,x)$$

$$ReLU'(x) = \begin{cases} 0 & | \text{ x} < 0, \\ 1 & | \text{ x} > 0 \end{cases}$$

Výhodou prvních dvou funkcí je, že jsou z obou stran omezené a mají spojitou derivaci. *ReLU* lze snadno spočítat a je lepší pro řešení problému mizejícího gradientu, který popíšeme později 2.3.

2.2 Vrstevnaté neuronové sítě

Problémem *Perceptronu* je, že umí klasifikovat jen lineárně separabilní množiny a většina problémů lineárně separabilní není (Elizondo, 2006). Tento problém lze vyřešit pomocí více umělých neuronů tím, že neurony uspořádáme do vrstev tak, aby zakódovaný vstup problému byl vstupem do neuronů v první vrstvě a výstupy těchto neuronů byly vstupy neuronů ve vrstvě následující. Neurony lze teoreticky uspořádat do libovolného orientovaného grafu, ale vrstevnatý způsob má především výpočetní výhody, protože lze uložit váhy synapsí vrstvy do matice a výpočet výstupu celé vrstvy pak spočítáme pomocí násobení vstupu maticí vah a následnou aplikací aktivační funkce. V jednoduchém modelu vrstevnaté sítě počítáme s tím, že neurony nemají žádná spojení v rámci jedné vrstvy ani mezi nesousedícími vrstvami. Pokud má vrstevnatá architektura neuronů alespoň jednu skrytou vrstvu s neomezeným počtem neuronů a používá vhodnou



Obrázek 2.2: Vrstevnatá síť. Zdroj: (Fernández-Cabán a kol., 2018)

aktivační funkci, je síť schopná aproximovat libovolnou spojitou funkci (Hornik a kol., 1990).

Celá vrstevnatá neuronová síť je jednoznačně definována spojením neuronů a maticemi vah vektorů jednotlivých vrstev. Schéma vrstevnaté sítě můžeme vidět v obrázku 2.2

2.3 Učení neuronových sítí

Cílem učení neuronových sítí je nalézt takovou síť, která bude mít co nejlepší výsledky na libovolných testovacích datech, která byla vygenerována stejným procesem jako data trénovací. Učení sítě lze v zásadě rozdělit na učení architektury a učení samotných vah neuronů (Abraham, 2004). Při učení je třeba si dát pozor na takzvané přeučení, to je stav, kdy se síť příliš dlouho trénuje na trénovacích datech a začne si pamatovat každý jednotlivý příklad. Datový šum, který se v učících datech nutně vyskytuje, pak převáží nad hledáním společných vzorců v datech a úspěšnost sítě na testovacích datech klesá. Existuje několik způsobů jak se této situaci bránit, jednou z možností je mít dostatečné množství dat nebo je rozšířit o podobná data s přidaným náhodným šumem. Problémem větších sítí však je, že mají příliš mnoho parametrů a ani opravdu velký dataset nemusí být dostatečný. Další možností je zastavit učení včas, než se stihne přeučit, nebo penalizovat složitost samotného modelu. Toho lze dosáhnout například tím, že jsou penalizovány příliš vysoké váhy parametrů. Nejlepších výsledků se obvykle dosáhne kombinací všech těchto metod (Brownlee, 2018). (Dongare a kol., 2012).

2.3.1 Učení parametrů

Návrh vhodné architektury je většinou úkolem programátora. Pokud však chceme učit jen parametry sítě, předpokládáme, že samotnou architekturu už máme. Výpočet výsledku pak lze převést na násobení vstupního vektoru maticemi parametrů jednotlivých vrstev.

Jednou z možností, jak tyto parametry učit, je definovat si nějakou ztrátovou (loss) funkci, například *střední kvadratickou odchylku* $L = \frac{1}{n}\sum_i (y_i - \hat{y}_i)^2$ nebo *kategorickou křížovou entropii* $L = -\sum_i y_i log(\hat{y}_i)$, kde y je požadovaný výstup a $\hat{y}$ je výstup modelu po aplikování na vstup x. Takovou funkci pak lze zderivovat podle každého jejího parametru a určit gradient funkce pomocí algoritmu zpětného šíření chyby, který je popsán v pseudokódu níže 2.3.1 (Rojas, 1996). Od parametrů funkce potom odečteme α násobek gradientu a řekneme, že α je učící konstanta.

Samotný gradient získáme jako derivaci složené funkce přes jednotlivé vrstvy sítě. K tomuto úkolu je potřeba, aby derivace funkce šla snadno vypočíst z původní hodnoty. Stochastická verze tohoto algoritmu upravuje váhy po každém záznamu. Kvůli efektivitě výpočtů se ale gradient nejčastěji počítá pro několik příkladů najednou (batch). Tento druhý způsob učení parametrů je mnohem rychlejší a díky počítání na grafické kartě lze také masivně paralelizovat, navíc náhodné volení příkladů, jež se budou počítat spolu, přináší do dat přirozený šum a pomáhá v boji s přeučením. Nevýhodou je určitá přímočarost, gradientní metody půjdou vždycky nejprudším spádem v daném místě a velikost kroku je také dána velikostí gradientu. Pro funkce, které mají lokální optima nebo větší plochy stejných hodnot, bude učení parametrů v těchto místech značně zpomalené nebo zastavené (Abraham, 2004).

Algorithm 1 Algoritmus backpropagace. Zdroj: (Guo a kol., 2019)

$X \leftarrow m$ trénovacích vstupních vektorů o velikosti n
$y \leftarrow$ očekávané výstupy pro trénovací data
$w \leftarrow$ vektory vah neuronů v jednotlivých vrstvách
$l \leftarrow$ počet vrstev neuronové sítě, $1...L$
$D_{ij}^{(l)} \leftarrow$ chyba, pro všechna i, j, l
$t_{ij}^{(l)} \leftarrow 0$, pro všechna i, j, l
for $i = 1$ to m **do**
    $a^l \leftarrow$ výsledek dopředného průchodu sítě s parametry $x^{(i)}$ a w
    $d^l \leftarrow a(L) - y(i)$
    $t_{ij}^{(l)} \leftarrow t_{ij}^{(l)} + a_j^{(l)} * t_i^{l+1}$
end for
if $j \neq 0$ **then**
    $D_{ij}^{(l)} \leftarrow \frac{1}{m} t_{ij}^{(l)} + \lambda w_{ij}^{(l)}$
else
    $D_{ij}^{(l)} \leftarrow \frac{1}{m} t_{ij}^{(l)}$
    kde $\frac{\partial}{\partial w_{ij}^{(l)}} J(w) = D_{ij}^{(l)}$
end if

Druhou možností přístupu je zakódovat síť do řetězce a použít dalšího zástupce biologických přístupů, evoluční algoritmus (Abraham, 2004). Jako fitness sítě si můžeme zvolit počet správně klasifikovaných případů z trénovacích dat, pokud řešíme klasifikační problém, nebo zápornou verzi *střední kvadratické odchylky*. V evolučních algoritmech totiž obvykle mívají lepší jedinci vyšší fitness. Tento způsob má několik výhod, například lze snadno paralelizovat a je možné se různými technikami bránit zaseknutí učení v lokálnímu optimu. Jeho hlavní nevýhodou je především to, že je velmi pomalý, zvlášť při velkém množství parametrů, a tudíž není moc využitelný pro učení větších sítí.

K přístupu založeném na evolučních algoritmech lze sáhnout i v případech, kdy nelze snadno definovat derivovatelnou ztrátovou funkci, která je potřeba pro následující postup.

2.3.2 Učení architektury

Pro učení architektury nelze snadno definovat gradient, používají se proto různé kombinace zpětnovazebního učení, evolučních algoritmů a dalších metod. Tyto postupy bývají časově náročné, protože trvá velmi dlouho vyhodnotit přesnost, časovou náročnost a celkovou kvalitu částečného řešení. Učení parametrů může v některých případech trvat dny nebo i týdny. Učení architektury trvá ještě násobně déle.

Existují i přístupy, které učí architekturu i váhy najednou (Stanley a Miikkulainen, 2002).

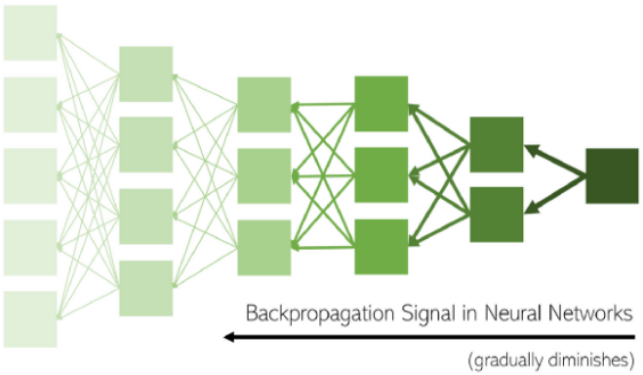
2.4 Hluboké sítě

Přestože algoritmus zpětného šíření není teoreticky omezen hloubkou sítě, a je tedy možné přidávat libovolné množství skrytých vrstev, dlouho nebyl důvod hledat hlubší sítě. Další vrstvy model nijak nezobecňují, protože už s jednou skrytou vrstvou (s neomezeným množstvím neuronů) můžeme aproximovat jakoukoli spojitou funkci. Navíc existovalo několik velkých překážek, které bránily naučení hlubších vrstev sítě.

Jedena z největších překážek bránících učení hlubokých sítí je mizení gradientu, jež nastává, pokud je derivace funkce ve velké části definičního oboru příliš blízko nule. Užitečná informace o změně vah se potom nezvládne propagovat z výstupní vrstvy až na začátek sítě. Tento problém lze vyřešit použitím aktivační funkce $ReLU$, jejíž derivace je vždy 0 nebo 1, a je nízká pravděpodobnost, že bude gradient ve všech parametrech vždy 0 a alespoň některé parametry se změní s plnou váhou gradientu (Krizhevsky a kol., 2012).

Opačný problém se nazývá exploze gradientu a lze jej řešit například maximální možnou hodnotou gradientu v daném směru. Oba tyto problémy lze také částečně řešit vhodnou instanciací vah parametrů na začátku učení nebo zavedením spojení mezi vrstvami, které spolu přímo nesousedí. U takovýchto sítí může gradient snadněji proplouvat skrz celou síť.

U hlubších vrstev bývá problém s adaptací na neustále se měnící vstup do vrstvy. Problém je v tom, že první vrstva se učí přímo z dat, která se nemění, ale každá další vrstva musí vlastně *čekat*, než se ustálí výstup předchozích vrstev, což značně prodlužuje učící fázi sítě. Tento problém lze řešit technikou zvanou



Obrázek 2.3: Mizející gradient. Zdroj: (Ye, 2020)

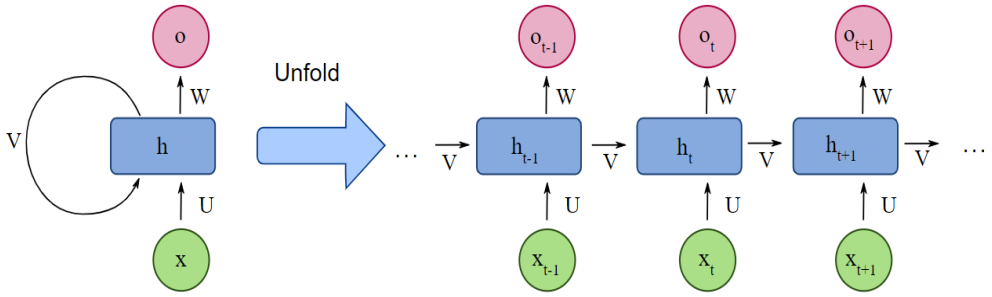
batch normalizace. Po vrstvě zde vyžadujeme, aby přes celou batch, neboli množinu příkladů, které se zpracovávají najednou, byla u výstupu z vrstvy stejná střední hodnota a odchylka. Střední hodnota a odchylka se obvykle volí tak, aby výstup odpovídal normálnímu rozdělení. Tím se docílí toho, že spodnější vrstvy si po částečném naučení horních vrstev nemusí zvykat na úplně nové hodnoty. Zároveň se tím řeší problémy při inicializaci matic, pomáhá proti mizení a explodování gradientu tím, že výstupy mají střední hodnotu 0, a nakonec také to, že funkce $ReLU$ není lichá (Ye, 2020).

Další klasická technika vylepšující výkonnost sítě je tzv. *dropout*, která řeší problém, kdy jsou neurony příliš naučené na trénovacích datech a jsou citlivé na šum, který se v nich přirozeně vyskytuje. V konkrétní vrstvě se to projevuje tím, že jsou na sobě neurony příliš závislé. Dropout tento problém řeší tak, že se vždy náhodně zvolí nějaké neurony z vrstvy, které se vůbec nepoužijí pro výpočet výstupu. Síť je potom celkově robustnější a neurony dokáží pružněji reagovat.

2.5 Rekurentní a konvoluční sítě

V klasické dopředné síti je každá část vstupu zpracovávána jiným neuronem. Pokud chceme v datech poznávat vzory, které se mohou vyskytovat kdekoli, je trénování klasické husté vrstvy velice náročné. Poznávání vzorů v datech je typická úloha při rozpoznávání obrázků a skvěle se na ni hodí konvoluční vrstva. Tato vrstva funguje tak, čte malá okna sousedních vstupů a aplikuje na ně stále stejnou funkci. Tímto způsobem se operace aplikuje na všechna možná okénka, která se mohou překrývat, a taková síť je schopna rozpoznat stejnou věc kdekoli v datech a má daleko méně učících parametrů (Krizhevsky a kol., 2012).

Podobný problém řeší také rekurentní sítě, až na to, že nás často zajímají data, která nemají stejnou délku. Nejčastěji se jedná o časové sekvence, kdy je potřeba rozpoznat události nezávisle na tom, kdy se vyskytnou. Některé události mohou mít mezi sebou dokonce proměnlivý počet událostí a síť musí tyto vzory zachytit. U klasických dopředných neuronových sítí tvoří neurony acyklický orientovaný graf a při detekci nejsou nové vstupy ovlivněny předchozími. U rekurentních sítí vede hrana zpět do stejné vrstvy a počítání předchozích výsledků tedy ovlivní počítání současného. V podstatě to funguje tak, že si síť předává vnitřní stav. Výpočet rekurentní sítě je zachycen na obrázku 2.4.

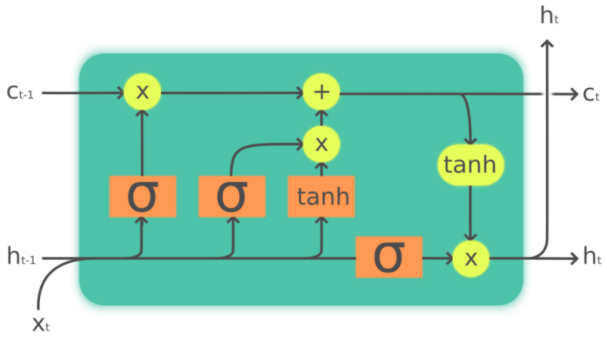


Obrázek 2.4: Rozvinutí rekurentní sítě. Zdroj: (Ixnay, 2017)

Výstup rekurentní vrstvy se určí na základě vah sítě, které jsou vždy stejné, a paměťového vektoru z předchozího výpočtu. Výstupem sítě po celé sekvenci může být sekvence vektorů nebo jen poslední vektor. Učení rekurentní sítě je realizováno pomocí rozvinutí algoritmu backpropagace v čase. Problémem tohoto algoritmu je, že se gradient vždy násobí stejnou vahou, čímž exponenciálně klesá nebo roste jeho hodnota. To znemožňuje obyčejné rekurentní síti pamatovat si události napříč dlouhými sekvencemi (Cuéllar a kol., 2006).

2.5.1 LSTM

LSTM neboli *Long-short term memory* je architektura rekurentní sítě, která úspěšně řeší problém dlouhodobé paměti. Hlavním principem je nahrazení jednoduché rekurentní vrstvy LSTM buňkou, jež je zobrazena na obrázku 2.5. Tato buňka má explicitní vnitřní paměť, ke které se chová jinak než ke klasickému vstupu. Má také navíc tři *brány*, což jsou klasické husté vrstvy s aktivační funkcí *sigmoid*. Zapomínací brána na základě vstupu určí, které části vnitřního stavu se mají zapomenout, vstupní brána na základě vstupu určí, které části potenciálního výstupu původní rekurentní vrstvy se mají zapamatovat a výstupní brána určí, které části vnitřního stavu půjdou na výstup. Právě schopnost zapomínat nedůležité informace, které zahlcují klasickou rekurentní buňku, se ukázala klíčová při řešení praktických úloh.



Obrázek 2.5: Buňka LSTM. Zdroj: (Chevalier, 2018)

$$f_t = \sigma(W_f x_t + U_f h_{t-1} + b_f)$$
$$i_t = \sigma(W_i x_t + U_i h_{t-1} + b_i)$$
$$o_t = \sigma(W_o x_t + U_o h_{t-1} + b_o)$$
$$\tilde{c}_t = tanh(W_c x_t + U_c h_{t-1} + b_c)$$
$$c_t = f_t \circ c_{t-1} + i_t \circ \tilde{c}_t$$
$$h_t = o_t \circ tanh(c_t)$$

Předchozí rovnice znázorňují samotný výpočet LSTM buňky, kde f_t, je aktivační vektor zapomínání, i_t je vstupní aktivační vektor a o_t je výstupní aktivační vektor. Matice W obsahují váhy vstupních parametrů a matice U obsahují váhy rekurentních spojení. Vektor $\tilde{c}_t$ je vlastně výsledek původní jednoduché rekurentní buňky. Váha stavového vektoru c_t pro další stav se počítá složkovým pronásobením vektoru $\tilde{c}_t$ se vstupním aktivačním vektorem, který se sečte se stavovým vektorem z minulého stavu, který byl napřed také složkově vynásoben aktivačním vektorem zapomínání. Výstup z buňky je pak spočten jako složkový násobek výstupního aktivačního vektoru a stavového vektoru, na který jsme předtím aplikovali funkci *tanh*.

3. Datasety

V této kapitole se budeme věnovat popisu datasetů, které budou v práci používány.

3.1 HDFS log dataset

Dataset poprvé zpracovali autoři práce (Xu a kol., 2009) a od té doby byl ještě mnohokrát zpracovaný jinými autory, především kvůli své velikosti a faktu, že data jsou anotovaná (Du a kol., 2017) (Zhang a kol., 2019) (Guo a kol., 2021). Data vznikla činností Hadoop clusteru skládajícím se z 203 uzlů na map-reduce jobech (Foundation, 2021), která trvala celkem 48 hodin.

Samotné záznamy jsou částečně strukturované, jeden záznam může vypadat například takto:
081109 203518 143 INFO dfs.DataNode$DataXceiver: Receiving block blk_-1608 999687919862906 src: /10.250.19.102:54106 dest: /10.250.19.102:50010,
kde první dvě položky kódují datum a čas. Důležitou částí logu je především samotná zpráva, která začíná za první dvojtečkou a končí na konci záznamu. Jako jeden z parametrů se vždy vyskytuje identifikační číslo bloku, podle něhož jsou zprávy sestavovány do sekvencí neboli bloků, které jsou anotovány Hadoop experty. Mezi dalšími parametry zpráv můžou být například velikosti souborů, IP adresy nebo časový limit. Některé modely používají časový limit nebo velikosti souborů pro učení, ale IP adresy jsou vždy zahozeny, protože není možné je zpracovat kvantitativně ani kvalitativně. Samotné parsování na *klíče* a *parametry*, které byly popsány v kapitole 1.1.1, je implementováno skriptem pomocí regulárních výrazů.

Dataset čítá celkem 11 197 954 záznamů a v záznamech se vyskytuje celkem 575 139 bloků, ale pouze 589 z nich má různé průchody. 2,9% logů je ohodnoceno jako anomální, ale ze všech 29 typů logů se 12 typů vyskytuje výhradně v anomálních blocích. Tabulky všech typů logů a počty jejích výskytů jsou uvedeny v příloze A.1.

Z tabulky můžeme vyčíst, že více než třetina všech klíčů se vůbec nevyskytuje v normálních datech a jsou tedy samy o sobě indikátorem abnormality sekvence. Průměrná délka u normálních sekvencí je lehce delší než u anomálních, podobně je tomu u maximální délky. Daleko výraznější rozdíl vidíme u minimální délky, která činí jen 3 klíče u anomálních sekvencí. Ze všech 16 838 anomálních sekvencí bylo dokonce 6 191 kratších než 10. Tento jev mohl být způsoben například tím, že v systému vyskytl problém a nemohl už běžet dál, takže se ukončilo také logování bloku.

Tabulka 3.1: HDFS log dataset - sekvence

Popis	Normální	Anomální	Celkem
Průměrná délka sekvence	19,5	17,1	19,4
Maximální délka sekvence	298	284	298
Minimální délka sekvence	13	2	2
Počet sekvencí	558 223	16 838	575 061
Počet sekvencí délky < 10	0	6 191	6 191

3.2 OpenStack log dataset

Tento dataset je mnohem menší než předchozí a vytvořili ho autoři modelu *DeepLog* (Du a kol., 2017). Obsahuje celkem 1 335 318 záznamů, z čehož asi 7% jsou anomální. Data byla vygenerována při plnění úkolů spojených s obsluhováním virtuálních strojů (VM) na cloudu CloudLab. Cyklus VM začíná *VM Create* a končí *VM Delete*, během toho se mohou objevovat páry událostí *VM Stop - VM Start, VM Pause - VM Unpause* a *VM Suspend - VM Resume*. Do logů vygenerovaných těmito událostmi byly potom vloženy tři typy chyb:

1. překročení časového limitu při vytváření virtuálního stroje
2. chyba při destrukci virtuálního stroje
3. chyba při čištění po destrukci virtuálního stroje

Záznamy v log souboru mohou vypadat například takto:
207641, nova-compute.log.2017-05-14_21:27:09, 2017-05-14, 19:39:03.166, 2931, INFO, nova.compute.manager, - , [instance: 2b590f10-49fd-4ec9-ae41-19596c2f4 b25] VM Stopped (Lifecycle Event), 7f86987c, [instance <> VM <*> (Lifecycle Event), "['2b590f10-49fd-4ec9-ae41-19596c2f4b25]', 'Stopped']"*

Opět se jedná o částečně strukturovaný text, obsahující datum a čas, bohužel velká část logů neobsahuje parametr *instance_id*, který spojuje záznamy do sekvencí. V datasetu jsou označeny instance, které měly vloženou chybu, a neoznačené logy tedy nelze nikam zařadit. Po odečtení těchto neidentifikovatelných záznamů jich zůstane celkem 55 694. Oproti předchozímu datasetu jsou abnormální data daleko podobnější normálním, sekvence mají totiž podobnou průměrnou, minimální i maximální délku. Autoři práce parsovali nezpracované logy pomocí metody *Spell* (Du a Li, 2016), která data rozdělila na 40 klíčů. Pro datovou analýzu jsme zpracovávali klíče manuálně, abychom měli co nejvyšší přesnost klasifikace, protože tyto automatické metody nejsou 100% spolehlivé a dostali jsme 42 neslučitelných klíčů. Protože ale 19 z těchto klíčů neobsahovaly parametr *instance_id*, konečná data mají jen 25 různých klíčů. Popis klíčů a jejich čestnost je uvedena v příloze A.2.

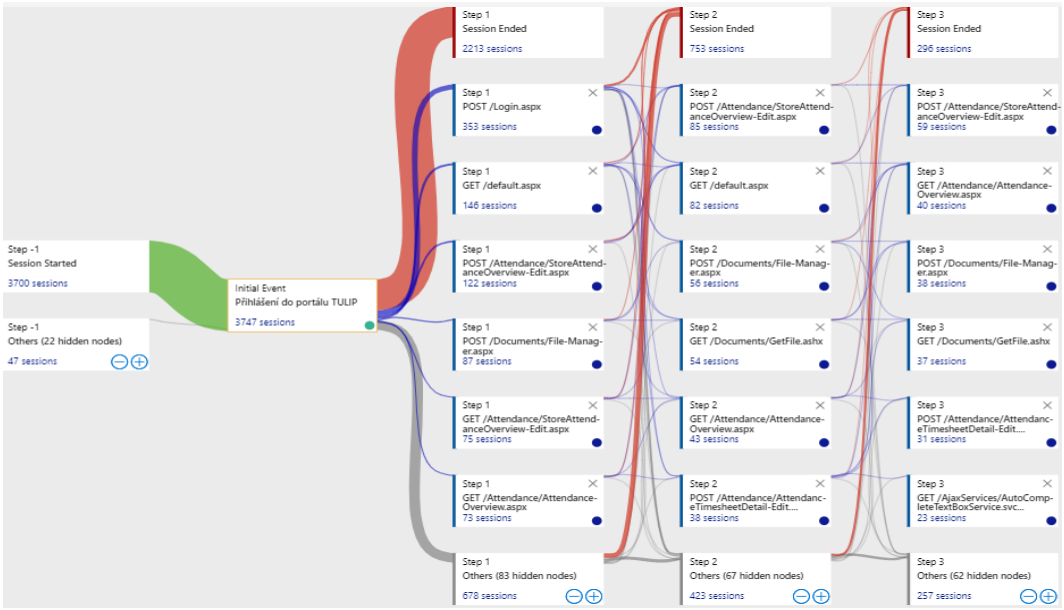
Celkem tedy dataset obsahuje 2090 různých sekvencí, ale když z každé sekvence vytvoříme vektor délky počtu různých událostí tak, že sečteme jednotlivé typy událostí v sekvenci, dostaneme pouze 11 různých vektorů. Těchto 15 typů sekvencí ani nemá mnoho permutací, protože celkový počet různých sekvencí je jen 24.

Tabulka 3.2: OpenStack log dataset - délka sekvencí

Popis	Normální	Anomální	Celkem
Průměrná délka sekvence	13,4	13,4	13,4
Maximální délka sekvence	28	27	28
Minimální délka sekvence	2	2	2
Počet sekvencí	2628	394	3741
Počet sekvencí délky < 10	2	1	3

3.3 HAVIT dataset

Ve spolupráci s firmou HAVIT, s.r.o. jsme vytvořili další dataset. Data byla vygenerována webovou aplikací napsanou v ASP.NET frameworku, která běží na Microsoft Azure cloudu, která spravuje docházky a výplatní pásky. Samotné zprávy byly zaznamenány pomocí technologie Application Insights v průběhu tří týdnů a jedná se o dotazy na backend aplikace. Sekvence logů jsou seskupeny podle *user_id*, takže jeden blok reprezentuje interakci uživatele s aplikací. Tuto interakci můžeme reprezentovat grafem těchto událostí. Obrázek 3.1 znázorňuje, jak se chovají uživatelé po přihlášení.



Obrázek 3.1: graf sekvencí, procházejících událostí přihlášení, Zdroj: Azure Portál

Problém takového sbírání logů je zejména ve velké vytíženosti aplikace a technologie Application Insights proto zprávy vzorkuje, čímž v normálních blocích mohou vznikat nevalidní sekvence a značně to zvyšuje datový šum.

Záznamy

Samotný záznam vypadá například takto: *7/20/2021, 8:27:40.085 PM,"POST /Documents/File-Manager.aspx","61.145300000000006",«250ms",AAYrN.* První dvě položky jsou datum a čas, třetí položka je typ události, další dvě položky se týkají doby trvání dotazu a poslední položka je *user_id*, podle které jsou události seskupovány. Celkově bylo sesbíráno 137 545 událostí, které patří do 14 583 bloků. Data obsahují celkem 274 různých událostí, ale pouze 152 se událo více než desetkrát. Neobsahují mnoho mnoho stejných posloupností, mají totiž celkem 10 394 různých vektorů s počtem událostí a 11 338 různých sekvencí.

Generování anomálií

Anotace tak velkého množství dat není možné dosáhnout v rozumném čase, ale pomůžeme si předpokladem, že většina dat by měla být normální, protože aplikace je reálně využívaná. Na základě tohoto předpokladu řekneme, že všechna nasbíraná data jsou normální. Anomálie budou generovány jako náhodné sekvence událostí vyskytujících se v datech. Délka sekvence je určena délkou náhodně vybrané sekvence v datech a události jsou vybírány podle četnosti jejích výskytů. Nakonec jsou z anomálních sekvencí odstraněny ty, které se už vyskytují v sesbíraných datech.

Tento způsob generování anomálií je validní, protože modely, které budeme používat, potřebují jen částečně anotovaná data. Učí se totiž jen na normálních příkladech a tyto náhodně vygenerovaná data budeme používat pouze pro vyhodnocování úspěšnosti modelů, nikoli pro samotné učení. Přesto je třeba myslet na to, že se ve vygenerovaných datech mohou vyskytovat normální sekvence. Při případné aplikaci je třeba zvážit, jestli daný pozitivní nález je skutečně anomální, a pokud není, model upravit s ohledem na daný případ, například přetrénováním. Následující tabulka ukazuje základní srovnání anomálních a neanomálních dat.

Tabulka 3.3: HAVIT log dataset - délka sekvencí

Popis	Normální	Anomální	Celkem
Průměrná délka sekvence	10.4	13,2	11.1
Maximální délka sekvence	2 101	1 231	2 101
Minimální délka sekvence	2	3	2
Počet sekvencí	14 583	4 373	18 956
Počet sekvencí délky < 10	11 978	3 382	15 360

4. Popis modelů

V této kapitole budou popsány modely, které jsme zvolili na základě informací o dostupných datasetech. Protože většina dostupných datasetů je buď neanotovaná, případně částečně anotovaná, bylo vhodné zvolit takové modely, které se budou učit pouze podle normálních dat, a anomální data budou využita pouze pro vyhodnocení daných modelů.

Modely jsou také navrženy pouze na detekování anomálií v posloupnostech klíčů událostí a parametry těchto událostí nejsou využity. Využití parametrů patří mezi možná rozšíření modelů v případné navazující práci.

Parsování konkrétních dat se liší podle konkrétního datasetu, proto už bylo popsáno v kapitolách týkajících se zpracování dat 3.

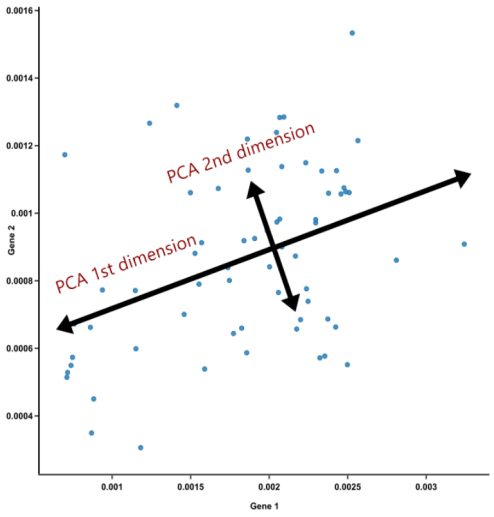
4.1 Hladový model

Pro některé datasety z předchozí kapitoly platilo, že některé záznamy se vyskytovaly pouze v anomálních sekvencích. Tyto anomální sekvence navíc často vybočovaly také tím, že byly velmi krátké. Hladový model tedy pro každý dataset definujeme tak, že anomální sekvence budou určeny svou délkou a svými charakteristickými záznamy.

4.2 PCA model

PCA neboli *Principal component analysis* je metoda lineární transformace do stejně velkého prostoru. Jedná se vlastně o regulární zobrazení rotace, které splňuje to, že vektory nové ortonormální báze jsou seřazeny tak, aby postupně zachycovaly maximální rozptyl v datech. Přesněji řečeno první vektor má takový směr, že při lineárním zobrazení dat na tento vektor získáme maximální rozptyl, jak můžeme vidět na obrázku 4.1. To samé potom platí pro následující vektory, přičemž se neberou v potaz dimenze předchozích vektorů. Vektorům této báze se říká hlavní komponenty (Jolliffe a Cadima, 2016) (Sekerka, 2017).

Dosud bylo zobrazení bezeztrátové, ale je možné se rozhodnout zachovat pouze nejdůležitější komponenty. Data totiž často lze vyjádřit jen pomocí několika málo dimenzí a nejméně důležité komponenty zachycují jen minimální část celkového rozptylu. Tímto způsobem lze opatrně redukovat dimenzi dat, aniž bychom ztratili příliš mnoho informace. Existují také velice efektivní algoritmy, zvlášť pokud chceme ponechat pouze hlavní komponenty, čehož se často využívá při předzpracování řídkých dat, než se zpracují složitější modelem, například neuronovými sítěmi.



Obrázek 4.1: Hlavní komponenty. Zdroj: (Ngo, 2018)

4.2.1 Popis PCA modelu

V práci (Xu a kol., 2009) autoři klasifikovali data pomocí PCA, kde hlavní myšlenka spočívá v tom, že se hledají hlavní komponenty pouze normálních dat. Při detekci se potom spočítá normálový vektor do prostoru hlavních komponent normálních dat. Pokud je norma tohoto vektoru větší než α, pak je vzorek určen jako anomální, protože se předpokládá, že normální vzorky se všechny nacházejí v prostoru s relativně nízkou dimenzí.

Předzpracování

Nejprve je třeba převést vstupní data, která jsou tvořená sekvencemi klíčů, do jediného vektoru. Toho lze dosáhnout jednoduchým sečtením one-hot reprezentace vektorů klíčů po složkách. Vektor nyní tedy obsahuje četnosti jednotlivých událostí, které se udály v dané sekvenci, již chceme reprezentovat. Potom jsou složky vektoru vydělené délkou sekvence. Nakonec je třeba sloučit všechna data do jediné matice a provést standardizaci, čili převedení jednotlivých proměnných na normální rozdělení, aby byly odstraněny efekty různých měřítek napříč proměnnými a PCA algoritmus mohl skutečně odhalit hlavní komponenty.

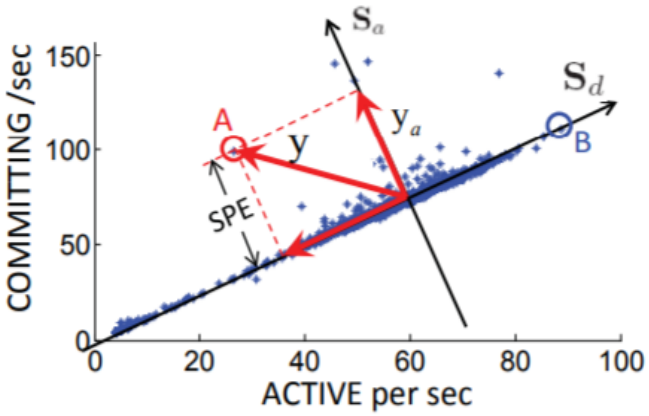
Učení

Učení probíhá tak, že se spočítají hlavní komponenty pro normální data. Zachová se tolik vektorů, aby celkový rozptyl byl alespoň 95%, což například u HDFS log datasetu bývá 3 vektory z původních 29. U HAVIT datasetu to bývá 170 vektorů.

Detekce

Když chceme vědět, jestli je záznam anomální, spočítáme normálový vektor do prostoru daného určenými hlavními komponentami. Pokud je normála tohoto vektoru větší než α, záznam je anomální. Tento parametr jsme volili podle konkrétních dat. Na obrázku 4.2 vidíme, že přestože vektor B není úplně ve stejném

clusteru jako ostatní normální body, bude ohodnocen jako normální, protože jeho normála na prostor daný hlavními komponentami je relativně nízká. Oproti tomu vektor A, který může mít některé parametry daleko podobnější některým jiným normálním vektorům, bude pravděpodobně ohodnocen jako anomální, neboť jeho vzdálenost do PCA prostoru je větší.



Obrázek 4.2: PCA detekce. Zdroj: (Xu a kol., 2009)

4.3 N-gram model

Další relativně přímočarou možností, jak modelovat anomální data, je použít *N-gramového* modelu (Du a kol., 2017). V tomto případě budeme předpokládat, že událost je ovlivněna jen několika bezprostředně předcházejícími událostmi. Pravděpodobnost události e v čase t pak lze modelovat jako počet n-tic předcházejících událostí včetně současné děleno počtem n-tic předchozích událostí bez současné události ve trénovací datech. Pokud by nás například zajímala pravděpodobnost události c a předcházející události byly a a b, výslednou pravděpodobnost $P[c|a,b]$ bychom spočítali jako počet n-tic (a,b,c) děleno počtem n-tic (a,b).

$$P[E_t = e|e_{t-n},...,e_{t-1}] = \#(e_{t-n},...,e_t)/\#(e_{t-n},...,e_{t-1})$$

Pro učení se tedy budou používat pouze normální logy, ze kterých se odvodí zmíněné pravděpodobnosti. V trénovací fázi se potom získá distribuce událostí za podmínky předešlých událostí. Sekvence bude normální, pokud pravděpodobnost každé její události za podmínky událostí bezprostředně předcházejících bude větší než nějaká konstanta α. Hodnota tohoto parametru závisí na konkrétních datech. Pokud při detekci narazíme na úplně nový příklad, posloupnost je automaticky prohlášena za anomální, tudíž není potřeba používat vyhlazování. Pokud bychom totiž vyhlazovali na hodnotu vyšší než α, bylo by to, jako bychom nové případy vždy povolovali. Kdybychom naopak zvolili vyhlazování nižší hodnotu než je α, bylo to stejné, jako kdyby žádné vyhlazování nebylo použito.

4.4 LSTM model

Náš LSTM model hodně vychází z předchozího N-gramového modelu, kdy se budeme snažit získat rozdělení pravděpodobnosti výskytu klíče za podmínky klíčů bezprostředně předcházejících. Stejně jako v předchozím modelu budeme pro učení používat pouze data, o nichž víme, že jsou normální, protože je snazší získat jen částečně anotovaná data. Výhodou LSTM sítí je, že dokáží efektivně využít daleko větší množiny předcházejících klíčů díky schopnosti zapomínat nedůležité informace. Nevýhodou tohoto přístupu je délka učení a také to, že je velmi snadné takový model přeučit.

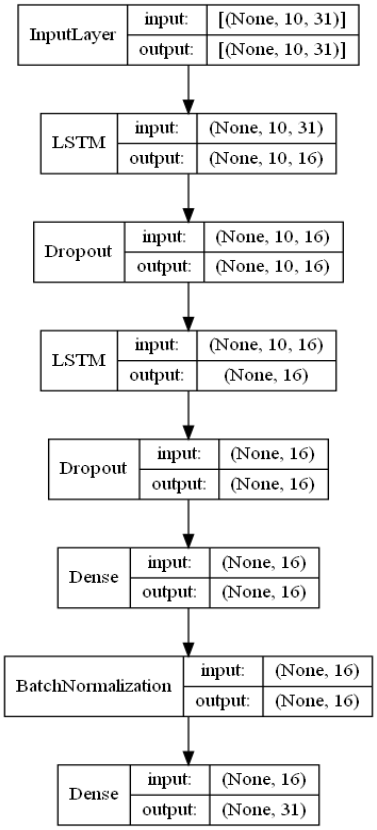
4.4.1 Příprava dat

Po úvodním parsování na klíče a vektory, které byly spojeny do množin podle identifikátoru sekvence, jsou pro učení modelu využity jen posloupnosti klíčů. Tyto posloupnosti jsou pak rozděleny do podposloupností podle délky okénka n, neboli předchozích klíčů, dle kterých se modeluje rozdělení dalšího klíče. Platí ovšem, že velikost okénka může být větší, než je velikost samotné posloupnosti. Také je důležité umět modelovat distribuce klíčů, které jsou před koncem okénka. Proto před každou posloupnost byla přidána skupina speciálních klíčů velikosti $n - 2$, které symbolizují události před začátkem sekvence, a jeden speciální klíč symbolizující samotný začátek. Pokud tedy budeme mít posloupnost $10, 11, 12, 13$ a velikost okénka bude $n = 3$, pak bude posloupnost rozdělena do následujících posloupností správné velikosti: $\{0, 0, 1, 10\}$, $\{0, 1, 10, 11\}$, $\{1, 10, 11, 12\}$, $\{10, 11, 12, 13\}$. První tři prvky budou vstupem do modelu a čtvrtý očekávaným výstupem. Výslednou trénovací množinu potom tvoří sjednocení těchto okének přes všechny normální posloupnosti v datech.

4.4.2 Architektura

Před učením je ještě kategorický vstup převeden pomocí *one-hot* kódování na vektor, který obsahuje jedničku na místě indexu kategorie, jinak nuly. Takový bude i očekávaný výstup modelu, proto je použita *kategorická křížová entropie* $L = -\sum_i y_i log(\hat{y}_i)$ jako ztrátová funkce.

Samotný model byl implementován v jazyce *Python* pomocí knihoven *TensorFlow a Keras*. První dvě vrstvy modelu jsou rekurentní s LSTM buňkou a používají rekurentní dropout i dropout jako vrstvu mezi nimi s parametrem 0,2. První z nich vrací sekvenci a druhá vrací pouze poslední vektor, který je potom vstupem do běžné plně propojené vrstvy, jež používá aktivační funkci *ReLU*. Nakonec je přidána ještě jedna plně propojená vrstva s aktivační funkcí *softmax*, která je vhodná pro kategorický výstup. Mezi posledními vrstvami je provedena *batch normalizace*. Všechny komponenty jsou teoreticky popsány v kapitole 2.



Obrázek 4.3: Graf LSTM modelu pro HDFS dataset

4.4.3 Detekce

Na výstup z modelu se můžeme dívat jako na pravděpodobnostní rozdělení následujícího klíče. Podobně jako v N-gramovém modelu můžeme u normálních sekvencí očekávat, že každý jejich klíč bude předpovězen s pravděpodobností větší než α, nebo že se bude nacházet mezi g nejpravděpodobnějšími klíči. Tento přístup byl využit v práci (Du a kol., 2017).

5. Vyhodnocení

V této kapitole budeme prezentovat porovnání modelů uvedených v předchozí kapitole na různých datech a porovnáme jejich výhody a nevýhody. Úspěšnost modelů se může na jednotlivých datasetech lišit, neboť každý dataset obsahuje trošku jiné typy anomálií. Samotné datasety jsou blíže popsány v kapitole 3.

Pro porovnávání modelů budeme používat metriky *accuracy, precision, recall a skóre F1. Accuracy (úspěšnost)* je počet správně předpovězených pozorování ku celkovému počtu pozorování. Jedná se o nejintuitivnější způsob vyhodnocení, který ale může být zavádějící, pokud nemáme symetrická data (počet pozitivních a negativních případů není stejný). *Precision (přesnost)* se počítá jako počet správně předpovězených pozitivních výsledků vůči všem výsledkům, které byly označeny za pozitivní. Vysoká přesnost značí nízké množství falešně pozitivních případů a konkrétně u problému hledání anomálních logů je důležitější vysoká přesnost než vysoká úspěšnost. *Recall (Citlivost)* značí poměr správně předpovězených pozitivních případů a všech skutečně pozitivních případů. *Skóre F1* je vážený průměr přesnosti a citlivosti, čímž se berou v úvahu falešně pozitivní i falešně negativní případy. Pozitivním výsledkem rozumíme, že je anomální, a negativní znamená, že výsledek je normální.

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

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

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

$$F1 = 2\frac{RecallPrecision}{Recall + Precision}$$

Nejprve popíšeme úspěšnosti modelů na jednotlivých datasetech. Protože každý model, který používáme, se učí jen na normálních příkladech, je možné pro vyhodnocení použít všechna anomální data a z normálních bude použit stejný počet, aby metriky byly co nejpřesnější.

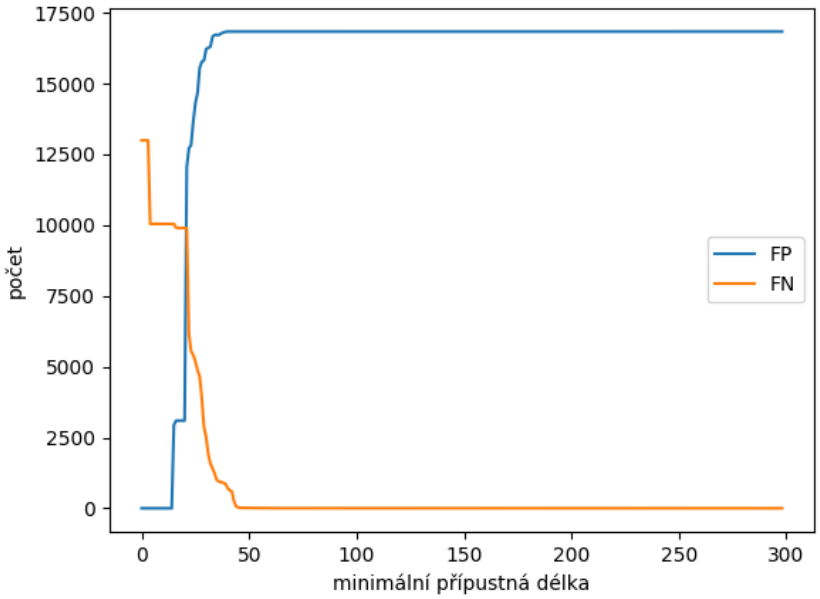
5.1 HDFS dataset

V následující tabulce, která porovnává různé modely na *HDFS log datasetu*, můžeme vidět, že hladový model má 100% přesnost, ale velkou část pozitivních případů není schopen identifikovat. Vidíme také, že LSTM je ve všech metrikách trochu lepší než N-gram, ale za cenu mnohonásobně vyššího času na učení. PCA přístup má ale jednoznačně nejlepší výsledky, včetně relativně rychlého učení. Tento výsledek je pravděpodobně dán tím, že agregované vektory posloupnosti v tomto datasetu tuto posloupnost identifikují jednoznačně. Jinými slovy v datech nejsou dvě posloupnosti, které by měly stejné počty událostí a zároveň jiné pořadí. Tento fakt umožňuje PCA algoritmu využít informací z celé posloupnosti a nikoli jen z několika předchozích událostí.

Tabulka 5.1: Porovnání modelů na HDFS datasetu

HDFS log dataset	Accuracy	Precision	Recall	F1 score
Hungry(n=13)	0.70178	1.0	0.40355	0.57504
PCA($\alpha = 0.99, t = 2$)	0.91979	0.99958	0.83995	0.91283
N-gram(n=1, $\alpha = 0.0005$)	0.90349	0.99218	0.81340	0.89393
LSTM(n=10, g=10, $\alpha = 16$)	0.90795	0.98931	0.82480	0.89960

V grafu 5.1, který srovnává počet falešně pozitivních a falešně negativních případů u hladového algoritmu v závislosti na minimální přípustné délce, vidíme, že hladový algoritmus může rozpoznat několik pozitivních případů jen na základě délky bloku. Jakmile je ale délka bloku větší než 13, pravděpodobnost pozitivního nálezu se stává nezávislou na délce.

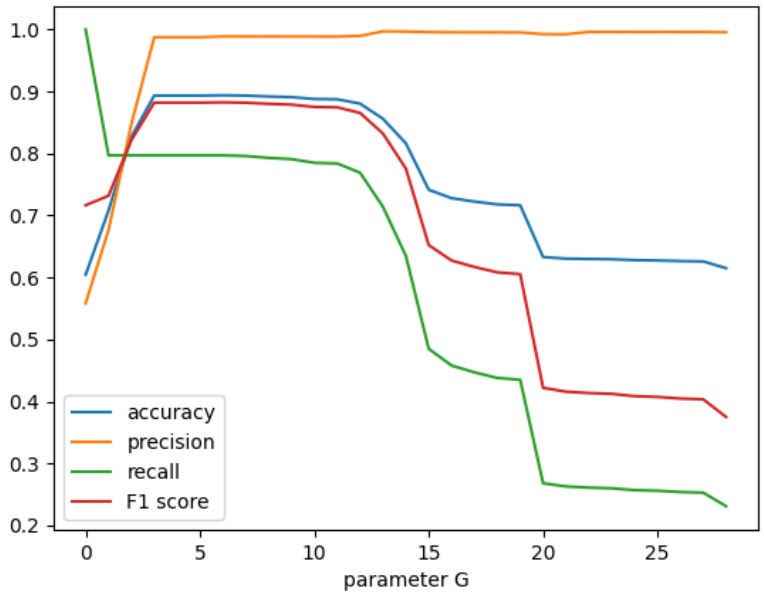


Obrázek 5.1: Porovnání FP a FN případů v závislosti na minimální délce přípustného bloku pro HDFS dataset.

Přestože se zdá, že je tento dataset dostatečně velký, jeho velmi nízká rozmanitost způsobuje, že je velice snadné přeučit LSTM model. Kombinace malého datasetu a vlastností modelu způsobuje velký nárůst falešně negativních případů při přeučení modelu. Proti tomuto přeučení nepomohlo snižování parametru g, snižování počtu neuronů ani snížení počtu učících iterací. Jediným efektivním způsobem bylo snížení velikosti trénovacích dat, protože se v nich příklady příliš opakují a jeden cyklus učení přes všechna data má stejný efekt, jako by cyklů bylo několik.

Graf 5.2 ukazuje, jak se mění metriky úspěšnosti modelu v závislosti na změnách parametru g. Tento parametr říká, jak velká je předpovězená množina nejpravděpodobnějších klíčů, ve které se musí nacházet očekávaný výsledek, aby byla sekvence označena za normální. Podle očekávání s vyšším g stoupá přesnost, protože je méně falešně pozitivních případů, a klesá citlivost, jelikož je více falešně

falešně negativních případů. Ideální volba parametru g je pro tento dataset mezi 4 a 9. U hodnot menších než 4 je vidět znatelné zhoršení při klesajícím g, především v přesnosti. Od hodnoty 11 a dále už začínají výrazně klesat ostatní metriky, přičemž přesnost se už od hodnoty 4 příliš nezlepšuje.



Obrázek 5.2: LSTM model, parametr g, HDFS dataset

5.2 OpenStack dataset

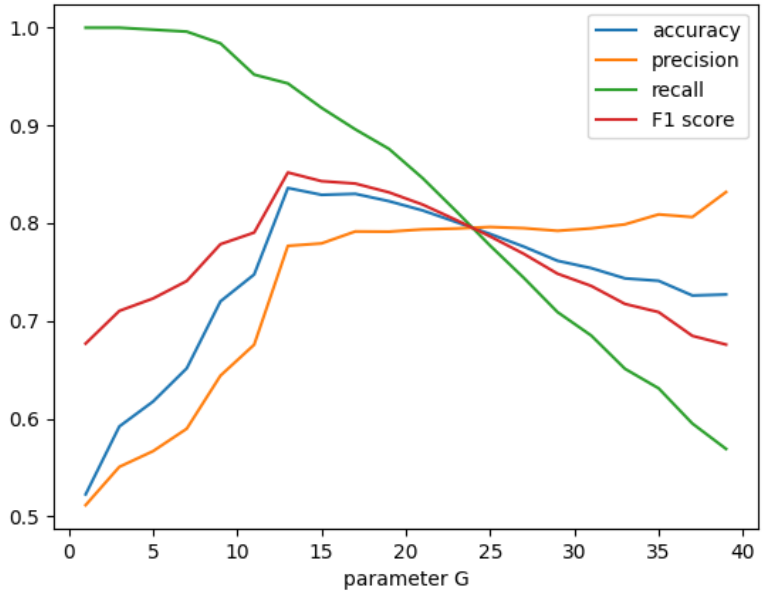
Chyby vložené do tohoto datasetu jsou především časového charakteru, tudíž lze předpokládat, že normální a anomální sekvence budou velmi podobné. Anomálie mohou být zachyceny ve vektoru parametrů a rozšíření modelů o možnost zpracování i této informace by mohla pomoci s jejich učením. Přesto bylo však zjištění, že se modely jen pomocí sledování typů událostí bez zpracovávání vektoru parametrů nenaučí téměř nic, velmi překvapivé. Je možné, že se nějaká přidaná informace nacházela v zahozených záznamech, které neobsahovaly položku *instance_id*, ale z popisu dat v práci (Du a kol., 2017) jsme nebyli schopni zbývající logy zařadit do existujících sekvencí. Možným řešením by mohlo být zařazení takových logů do každé sekvence. Takové řešení by ale způsobilo, že by se sekvence skládaly téměř výhradně z těchto společných záznamů a málo by se od sebe odlišovaly. Navíc by se tím narušila návaznost mezi logy, které by spolu jinak sousedily.

Tabulka 5.2: Porovnání modelů na OpenStack datasetu

HDFS log dataset	Accuracy	Precision	Recall	F1 score
Hungry	0.5	NAN	0	0
PCA($\alpha = 0.99, t = 2$)	0.5	0.75	0.007	0.01
N-gram(n=1, $\alpha = 0.0005$)	0.5	1	0.01	0.02
LSTM(n=10, g=10, $\alpha = 16$)	0.48	0.38	0.06	0.11

5.3 HAVIT dataset

Porovnávat hladový model na tomto datasetu nemá smysl, protože anomální sekvence byly generovány z normálních a neobsahují žádné speciální klíče ani nevybočují délkou. Graf 5.3 zobrazuje závislosti pozorovaných metrik na parametru g. Přestože jsme si pro tento problém definovali jako hlavní metriku přesnost, není vhodné tuto metriku maximalizovat za každou cenu. Nejvyšší přesnost má model s maximálním parametrem g. Takový model by každý případ vyhodnotil jako normální a to také není žádoucí chování. Nejvýhodnější hodnota parametru g s přihlédnutím k ostatním metrikám je 13, protože při dalším zvýšení začne velmi rychle klesat citlivost. V hodnotě 13 je také nejvyšší úspěšnost i skóre F1.



Obrázek 5.3: LSTM model, parametr g, HAVIT dataset

Tabulka 5.3: Porovnání modelů na HAVIT datasetu

HDFS log dataset	Accuracy	Precision	Recall	F1 score
PCA($\alpha = 0.95, t = 1$)	0.85193	0.88340	0.81088	0.84559
N-gram(n=1, $\alpha = 0.001$)	0.80848	0.94200	0.65744	0.77441
LSTM(n=10, g=13, $\alpha = 16$)	0.836	0.77677	0.943	0.85185

V tabulce 5.3 můžeme vidět, že PCA model má vysokou celkovou úspěšnost, podobně jako u HDFS datasetu. N-gramový model má nejvyšší přesnost. To znamená, že se nejlépe naučil předložená data a jen velmi malou část normálních označil jako pozitivní, na druhou stranu má nejhorší citlivost, takže z náhodných dat určil nejméně jako anomální. Toto ale nemusí být špatné chování, protože není jisté, že náhodná data musí být nutně anomálie, proto je třeba brát tento údaj na tomto datasetu s rezervou. LSTM model má výrazně nejhorší přesnost, což je pro nás nejdůležitější metrika. Sice má velmi vysokou citlivost, ale jak už bylo zmíněno výše, tento údaj není v tomto datasetu příliš vypovídající. LSTM

model má navíc velmi vysokou výpočetní náročnost. Tyto výsledky jsou odlišné od výsledků práce autorů modelu *DeepLog* (Du a kol., 2017), kde měl jejich LSTM přístup výrazně lepší výsledky než PCA. Tento rozdíl mohl být způsoben tím, že se *DeepLog* model skládá jak z *LogKey* modelu, tak z *Parameter value* modelu, který mohl zachytit další anomálie a tím zlepšit úspěšnost jejich modelu.

5.4 Výhody a nevýhody jednotlivých modelů

Hladový model byl úspěšný pouze na HDFS datech, která obsahují události příznačné pro anomální posloupnosti. Jakmile však data neobsahují tyto jednoduše odhalitelné anomálie, model přestává být jakkoli užitečný.

PCA model si na všech datasetech vede nejlépe, jeho největší slabina je předpoklad uzavřených dat. Reálné systémy jsou pod neustálým vývojem, který přináší stále nová data. PCA model je v tomto ohledu velmi statický a při vývoji systému je nutné jej neustále znovu počítat.

LSTM model je velmi náročný na výpočet, zvláště ve srovnání s podobně úspěšným N-gramem. Učení LSTM sítě trvá na dostupném počítači v řádu minut, zatímco počítání N-gramů několik sekund. Na druhou stranu poskytuje možnost po ukončení fáze učení upravovat model podle nově příchozích dat. Tato možnost je však omezená pouze na nové posloupnosti existujících událostí, nikoli úplně nové události. Pokud bychom chtěli podporovat také přidávání nových událostí, museli bychom změnit metodu parsování. Jeden z možných přístupu nabízí autoři práce (Zhang a kol., 2019), kteří text zpráv zpracovávali jako slova v přirozeném jazyce.

Nejvhodnější pro pozdější rozšiřování se tedy zdá *N-gramový* model, který má srovnatelné metriky úspěšnosti s LSTM modelem. Tento model podporuje přidání nových normálních nálezů tím, že se do datové struktury, která si pamatuje počty již zaznamenaných n-tic událostí, přidají n-tice vyskytující se v nové normální sekvenci. Poté je třeba ještě aktualizovat pravděpodobnosti n-gramů a z nich plynoucí rozdělení pravděpodobností. Toto však není časově náročný proces, zvlášť pokud je zvolena malá velikost okénka.

Závěr

V této práci jsme prozkoumali některé stávající možnosti zpracovávání anomálií v log datech. Na základě těchto informací jsme navrhli vlastní modely, které jsme vyzkoušeli na již existujících datasetech používaných v jiných pracích.

Anomálie je v nich obvykle definovaná na posloupnosti událostí, které spolu nějakým způsobem souvisí. Nejčastěji tím, že byly vygenerovány stejným procesem. Tuto posloupnost je někdy označována jako blok.

Analýza nejzajímavějších postupů je uvedena především v kapitole 1, kde jsou detailněji popsány dvě práce zabývající se daným problémem.

Naše vlastní vygenerovaná data jsme zpracovávali způsobem, který eliminuje největší nedostatky existujících datasetů. Bohužel největším problémem se ukázala data samotná a nemožnost přesných a úplných anotací dat, zejména z důvodu časové a vědomostní náročnosti tohoto úkolu. Pro správné anotace je nutná velmi dobrá znalost systému produkujícího logy a každý záznam nebo posloupnost záznamu musí být detailně prozkoumány, což vyžaduje hodně času.

Problém nedostatku kvalitních dat se projevil také při průzkumu existujících přístupů. Anotovaný HDFS dataset obsahuje přes 11 milionů logů, které jsou rozděleny do bloků. Tyto bloky jsou potom anotovány jako normální nebo anomální. Pouze 589 bloků má ale různé průběhy, všechny ostatní bloky, kterých je celkem přes půl milionu, byly autory datasetu (Xu a kol., 2009) anotovány podlě těchto bloků. Anomální bloky se navíc odlišují od normálních událostmi, jež se v normálních blocích nikdy nevyskytují.

Výslednými modely, které budeme porovnávat, jsou PCA model, N-gramový model, LSTM model a hladový přístup. Hladový přístup za anomální označí takové bloky, jež obsahují události, které jsou význačné pro anomální posloupnosti. PCA hledá nejmenší prostor, který dostatečně zachycuje normální příklady. Anomálie potom určuje podle vzdálenosti do tohoto prostoru. N-gramový model a LSTM model modelují distribuci budoucích log záznamů za podmínky bezprostředně předcházející množiny záznamů. Očekává se zde, že normální logy budou mít vyšší pravděpodobnost výskytu než anomální.

Na HDFS datasetu je z důvodu existence mnoha událostí význačných pro anomální bloky velmi úspěšný hladový přístup, jehož úspěšnost byla dokonce 70%. Navíc jako jediný model neměl žádný falešně pozitivní nález. Přesto byly komplikovanější modely schopné tuto úspěšnost překonat. PCA model měl nejvyšší úspěšnost ze všech a to 92%, jeho hlavní nevýhoda je v omezeném reálném využití pro systémy, které se ještě vyvíjejí. Tento nedostatek částečně odstraňují N-gramový model a LSTM model, které mají oba velmi podobnou úspěšnost jako PCA model, přičemž N-gram má asi 60-krát kratší dobu učení než LSTM model. Parametry všech modelů byly voleny takovým způsobem, aby počet falešně pozitivních nálezů bylo co nejméně, aniž by výrazněji utrpělo i množství falešně negativních případů.

Podobným problémem jako HDFS dataset trpí také dataset OpenStack, který je navíc zatížen nedostatečným popisem dat. Velká část záznamů nemá slíbené anotace a tyto záznamy jsme museli z analýzy vyloučit. Bohužel tímto vyloučením bylo způsobeno, že se úplně smazaly rozdíly mezi anotovanými a neanotovanými logy, a žádný model potom mezi nimi nedokázal rozpoznat rozdíly.

Dataset, který jsme vytvořili pro tuto práci ve spolupráci s firmou HAVIT, s.r.o., je založený na myšlence, že získaná data jsou pravděpodobně normální. Anomální data byla vytvořena náhodně, aby se co nejvíce podobala původním datům statisticky, ale samotné posloupnosti událostí jsou odlišné. Toto náhodné generování není na závadu, protože námi navržené modely se učí pouze z částečně anotovaných normálních dat a anomální data slouží pouze pro vyhodnocování.

Na tomto datasetu měl PCA model nejvyšší úspěšnost, ale nejméně falešně pozitivních výsledků měl N-gramový model. LSTM model měl velmi málo falešně negativních výsledků, ale zaprvé je pro tento problém důležitější minimalizovat falešná pozitiva a za druhé není řečeno, že anomální data jsou skutečně anomální, protože v tomto datasetu byla anomální data vygenerovaná náhodně.

Výsledné hodnocení modelů je takové, že LSTM model má nejvyšší časovou náročnost, nejvyšší programátorskou náročnost a nejnižší celkovou úspěšnost, což se odlišuje od výsledků práce (Du a kol., 2017). Je však možné že jsme někde přehlédli nějakou zásadní chybu, která by výsledky otočila. Dalším možným vysvětlením této situace může být to, že jsme neimplementovali také model identifikující anomální parametry logů, jako je například delší trvání dotazu. Tato skutečnost je z velké pravděpodobnosti příčinou toho, že se modely nebyly schopny učit na OpenStack datasetu. Rozšíření práce o tento model je tedy hlavním možným rozšířením této práce.

Cíle práce se nám z velké části splnit podařilo, neboť jsme vytvořili reálný dataset, jenž má daleko větší vnitřní komplexitu než datasety nejčastěji používané v pracích, z nichž jsme vycházeli. Modely, které jsme navrhli, přesto dokázaly s velkou úspěšností modelovat normální data a správně klasifikovat anomální příklady.

Možná rozšíření

Logovací záznamy lze zpracovat na klíč události a vektor parametrů. Jak už bylo zmíněno výše, nejvíce se nabízí rozšíření o identifikaci anomálií nejen pomocí klíče událostí, ale i s využitím informací, které se mohou nacházet v tomto vektoru parametrů. Model, jenž by úspěšně predikoval anomálie parametrů, se nám bohužel nepodařilo podle informací dostupných z práce (Du a kol., 2017) úspěšně implementovat. Jedním z důvodů by mohla být nízká pravděpodobnost nenulových hodnot parametrů. Model totiž ve většině případů předpovídal nulové hodnoty.

Dalším možným rozšířením je analýza přirozeného jazyka původní zprávy logu, zpracovaná v práci (Zhang a kol., 2019). Takový způsob umožňuje daleko vyšší rozšířitelnost modelů pro dosud neviděné události, což je jedna z klíčových vlastností v praktickém použití.

Výstup, že blok je anomální, může sám o sobě poskytovat příliš málo užitečné informace, aby byl programátor schopen jednoduše určit příčinu chyby. Možným rozšířením modelu může být poskytnutí dodatečných informací o výskytu anomálie.

Největší problém, který jsme v této práci řešili, byl nedostatek kvalitních velkých anotovaných datasetů. Vytvoření takového datasetu, jenž by byl veřejně přístupný, by mohlo velmi pomoci při hledání ideálního řešení pro hledání anomálií v log datech.

Seznam použité literatury

Abraham, A. (2004). Meta learning evolutionary artificial neural networks. *Neurocomputing*, **56**, 1–38.

Ajith, D. (2019). A survey on anomaly detection methods for system log data. *Int. J. Sci. Res.(IJSR)*, **8**, 23.

Azevedo, F. A., Carvalho, L. R., Grinberg, L. T., Farfel, J. M., Ferretti, R. E., Leite, R. E., Filho, W. J., Lent, R. a Herculano-Houzel, S. (2009). Equal numbers of neuronal and nonneuronal cells make the human brain an isometrically scaled-up primate brain. *Journal of Comparative Neurology*, **513**(5), 532–541.

Brownlee, J. (2018). *Better deep learning: train faster, reduce overfitting, and make better predictions*. Machine Learning Mastery.

Chevalier, G. (2018). Larnn: linear attention recurrent neural network. *arXiv preprint arXiv:1808.05578*.

Cuéllar, M., Delgado, M. a Pegalajar, M. (2006). An application of non-linear programming to train recurrent neural networks in time series prediction problems. In Chen, C.-S., Filipe, J., Seruca, I. a Cordeiro, J., editors, *Enterprise Information Systems VII*, pages 95–102, Dordrecht, 2006. Springer Netherlands. ISBN 978-1-4020-5347-4.

Dongare, A., Kharde, R., Kachare, A. D. a kol. (2012). Introduction to artificial neural network. *International Journal of Engineering and Innovative Technology (IJEIT)*, **2**(1), 189–194.

Du, M. a Li, F. (2016). Spell: Streaming parsing of system event logs. In *2016 IEEE 16th International Conference on Data Mining (ICDM)*, pages 859–864. IEEE.

Du, M., Li, F., Zheng, G. a Srikumar, V. (2017). Deeplog: Anomaly detection and diagnosis from system logs through deep learning. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1285–1298.

Elizondo, D. (2006). The linear separability problem: Some testing methods. *IEEE Transactions on neural networks*, **17**(2), 330–344.

Fernández-Cabán, P., Masters, F. a Phillips, B. (2018). Predicting roof pressures on a low-rise structure from freestream turbulence using artificial neural networks. *Frontiers in Built Environment*, **4**. doi: 10.3389/fbuil.2018.00068.

Foundation, A. S. (2021). Mapreduce tutorial. URL `https://hadoop.apache.org/docs/stable/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html`.

Guo, H., Yuan, S. a Wu, X. (2021). Logbert: Log anomaly detection via bert. *arXiv preprint arXiv:2103.04475*.

Guo, H., Nguyen, H., Vu, D.-A. a Bui, X.-N. (2019). Forecasting mining capital cost for open-pit mining projects based on artificial neural network approach. *Resources Policy*. doi: 10.1016/j.resourpol.2019.101474.

Hawkins, D. M. (1980). *Identification of outliers*, volume 11. Springer.

He, P., Zhu, J., Zheng, Z. a Lyu, M. R. (2017). Drain: An online log parsing approach with fixed depth tree. In *2017 IEEE international conference on web services (ICWS)*, pages 33–40. IEEE.

Hornik, K., Stinchcombe, M. a White, H. (1990). Universal approximation of an unknown mapping and its derivatives using multilayer feedforward networks. *Neural networks*, **3**(5), 551–560.

Ixnay (2017). Recurrent neural network. URL `https://en.wikipedia.org/wiki/Recurrent_neural_network#/media/File:Recurrent_neural_network_unfold.svg`.

Jolliffe, I. T. a Cadima, J. (2016). Principal component analysis: a review and recent developments. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, **374**(2065), 20150202.

Kabinna, S., Bezemer, C.-P., Shang, W., Syer, M. D. a Hassan, A. E. (2018). Examining the stability of logging statements. *Empirical Software Engineering*, **23**(1), 290–333.

Koch, C. (2004). *Biophysics of computation: information processing in single neurons*. Oxford university press.

Krizhevsky, A., Sutskever, I. a Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, **25**, 1097–1105.

Ngo, L. (2018). Principal component analysis explained simply. URL `https://blog.bioturing.com/2018/06/14/principal-component-analysis-explained-simply/`.

Rojas, R. (1996). The backpropagation algorithm. In *Neural networks*, pages 149–182. Springer.

Rosenblatt, F. (1957). *The perceptron, a perceiving and recognizing automaton Project Para*. Cornell Aeronautical Laboratory.

Sekerka, M. (2017). Algoritmus na rozpoznávání tváří pomocí metody pca, neboli analýzy havních komponentů. *Matematická sekce, Matematicko-fyzikální fakulta, Univerzita Karlova*.

Stanley, K. O. a Miikkulainen, R. (2002). Evolving neural networks through augmenting topologies. *Evolutionary Computation*, **10**(2), 99–127. doi: 10.1162/106365602320169811.

Xu, W., Huang, L., Fox, A., Patterson, D. a Jordan, M. I. (2009). Detecting large-scale system problems by mining console logs. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 117–132.

Ye, A. (2020). Batch normalization: The greatest breakthrough in deep learning. URL `https://towardsdatascience.com/batch-normalization-the-greatest-breakthrough-in-deep-learning-77e64909d81d`.

Zhang, X., Xu, Y., Lin, Q., Qiao, B., Zhang, H., Dang, Y., Xie, C., Yang, X., Cheng, Q., Li, Z. a kol. (2019). Robust log-based anomaly detection on unstable log data. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 807–817.

Seznam obrázků

1.1 DeepLog architektura. Zdroj: (Du a kol., 2017) 6
1.2 LogRobust architektura. Zdroj: (Zhang a kol., 2019) 11

2.1 Perceptron. Zdroj: (Dongare a kol., 2012) 13
2.2 Vrstevnatá síť. Zdroj: (Fernández-Cabán a kol., 2018) 14
2.3 Mizející gradient. Zdroj: (Ye, 2020) 17
2.4 Rozvinutí rekurentní sítě. Zdroj: (Ixnay, 2017) 18
2.5 Buňka LSTM. Zdroj: (Chevalier, 2018) 18

3.1 graf sekvencí, procházejících událostí přihlášení, Zdroj: Azure Portál 22

4.1 Hlavní komponenty. Zdroj: (Ngo, 2018) 25
4.2 PCA detekce. Zdroj: (Xu a kol., 2009) 26
4.3 Graf LSTM modelu pro HDFS dataset 28

5.1 Porovnání FP a FN případů v závislosti na minimální délce přípustného bloku pro HDFS dataset. 30
5.2 LSTM model, parametr g, HDFS dataset 31
5.3 LSTM model, parametr g, HAVIT dataset 32

Seznam tabulek

1.1 Parsování zprávy na klíč a vektor parametrů 6

3.1 HDFS log dataset - sekvence 21
3.2 OpenStack log dataset - délka sekvencí 22
3.3 HAVIT log dataset - délka sekvencí 23

5.1 Porovnání modelů na HDFS datasetu 30
5.2 Porovnání modelů na OpenStack datasetu 31
5.3 Porovnání modelů na HAVIT datasetu 32

Seznam použitých zkratek

VM: Virtual Machine,
TP: True positive (skutečně pozitivní),
TN: True negative (skutečně negativní),
FP: False positive (falešně pozitivní),
FN: False negative (falešně negativní),
LSTM: Long Short-Term Memory
PCA: Principal Component Analysis
HDFS: Hadoop Distributed File System
TF: Term Frequency
IDF: Inverse Document Frequency

A. Přílohy

A.1 HDFS log dataset

HDFS log keys, počet výskytů			
Celkem	Normální	Anomální	Log key text
11 175 629	10 887 379	288 250	Celkem
1 723 232	1 677 287	45 945	Receiving block * src: * dest: *
1 719 741	1 679 485	40 256	BLOCK* NameSystem.addS ...
1 706 679	1 674 669	32 010	PacketResponder * for block ...
1 706 514	1 674 669	31 845	Received block * of size * ...
1 402 047	1 366 574	35 473	Deleting block * file ...
1 396 174	1 364 251	31 923	BLOCK* NameSystem.delete: ...
575 061	558 223	16 838	BLOCK* NameSystem ...
428 726	416 884	11 842	* Served block * to *
356 207	347 948	8 259	*:Got exception while ...
120 036	116 907	3 129	Verification succeeded for *
7 097	2 617	4 480	Received block * src: * ...
7 002	2 543	4 459	* Starting thread to ...
7 002	2 543	4 459	BLOCK* ask * to replicate ...
6 937	2 542	4 395	*:Transmitted block * to *
5 545	222	5 323	Unexpected error trying to ...
3 416	0	3 416	writeBlock * received exception *
1 464	0	1 464	Receiving empty packet for ...
1 288	12	1 276	BLOCK* NameSystem.addSt ...
975	3	972	BLOCK* NameSystem ...
155	0	155	Exception in receiveBlock for ...
108	0	108	PacketResponder * * Exception *
65	0	65	Changing block file offset of ...
49	0	49	PacketResponder * for block * ...
47	0	47	PendingReplicationMonitor ...
34	0	34	*:Exception writing block * to ...
10	0	10	Adding an already existing ...
9	0	9	*:Failed to transfer * to * got *
5	0	5	Reopen Block *
4	0	4	BLOCK* Removing block * ...

A.2 OpenStack log dataset

OpenStack log keys, počet výskytů			
Celkem	Normální	Anomální	Log key text
55 694	50 397	5 297	Celkem
4 132	3 738	394	VM Resumed (Lifecycle Event)
4 130	3 738	392	* limit not specified, defaulting ...
4 053	3 666	387	During sync_power_state the ...
2 067	1 870	197	VM Stopped (Lifecycle Event)
2 066	1 869	197	VM Started (Lifecycle Event)
2 066	1 869	197	VM Paused (Lifecycle Event)
2 066	1 869	197	Instance spawned successfully.
2 066	1 869	197	Took * seconds to spawn the ...
2 066	1 869	197	Took * seconds to build instance.
2 066	1 869	197	Creating image
2 065	1 869	196	Took * seconds to ...
2 065	1 869	196	Creating event network-vif-
2 065	1 869	196	Attempting claim: memory * ...
2 065	1 869	196	Total memory: * MB, used: * MB
2 065	1 869	196	memory limit: * MB, free: * MB
2 065	1 869	196	Total disk: * GB, used: * GB
2 065	1 869	196	Total vcpu: * VCPU, used: ...
2 065	1 869	196	Claim successful
2 064	1 868	196	* "DELETE *"status: * len: *
2 064	1 868	196	Terminating instance
2 064	1 868	196	Instance destroyed successfully.
2 064	1 868	196	Deleting instance files *
2 064	1 868	196	Deletion of * complete
2 064	1 868	196	Took * seconds to destroy the ...
12	11	1	couldn’t obtain the vcpu count