FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

ZADÁNÍ DIPLOMOVÉ PRÁCE

Název:	Využití strojového učení pro extrakci kontextu síťových incidentů
Student:	Bc. Marek Jílek
Vedoucí:	Ing. Martin Kopp
Studijní program:	Informatika
Studijní obor:	Počítačová bezpečnost
Katedra:	Katedra počítačových systémů
Platnost zadání:	Do konce letního semestru 2018/19

Pokyny pro vypracování

- Nastudujte si detekční framework Cognitive threat analytics (CTA).
- Analyzujte kontext a chování jednotlivých skupin malwaru detekovaných CTA.
- Porovnejte kvalitu kontextových a bezkontextových klasifikátorů na reálných síťových datech.
- Navrhněte efektivní způsob extrakce a evaluace klasifikačních pravidel a srovnejte s konvenčními kontextovými klasifikátory.
- Na základě předchozích experimentů navrhněte metodu klasifikace, která kromě třídy nabídne i vysvětlení (kontext).

Seznam odborné literatury

Dodá vedoucí práce.

prof. Ing. Róbert Lórencz, CSc.
vedoucí katedry

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
děkan

V Praze dne 1. února 2018

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Diplomová práce

Využití strojového učení pro extrakci kontextu síťových incidentů

Bc. Marek Jílek

Katedra počítačové bezpečnosti
Vedoucí práce: Ing. Martin Kopp

6. května 2018

Poděkování

Děkuji všem, kteří mě během psaní této práce doprovázeli a podporovali. Jmenovitě bych chtěl poděkovat vedoucímu Ing. Martinovi Koppovi za trpělivost a vzorné vedení práce. Také děkuji své snoubence Lucii za psychickou podporu a jazykovou korekturu. V neposlední řadě děkuji také rodině, že mě dovedla až k tomuto momentu mého studia.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval(a) samostatně a že jsem uvedl(a) veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 46 odst. 6 tohoto zákona tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou, a veškeré jejich dokumentace (dále souhrnně jen „Dílo“), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla, a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu), licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 6. května 2018 .....

Odkaz na tuto práci

Jílek, Marek. *Využití strojového učení pro extrakci kontextu síťových incidentů.* Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2018.

Abstrakt

Tato práce pojednává o možnostech využití znalostního inženýrství pro odhalování síťových incidentů a extrakci jejich kontextu. V první části se zaměříme na současné systémy automatizovaného odhalování incidentů, především si představíme detekční framework CTA, jehož data poslouží jako zdroj pro experimenty.

V další části se zaměříme na metody extrakce a evaluace pravidel. Představíme si algoritmy FISM (více jeho verzi FP-Growth) a LISM. Navrhneme efektivní způsob evaluace pravidel pomocí prefixového stromu.

Dále porovnáme existující kontextové a bezkontextové klasifikátory, využívané v této oblasti (kNN, SVC, Naivní Bayes, neuronové sítě, náhodné lesy). Nejprve optimalizujeme parametry těchto klasifikátorů na omezené množině dat. Dále potom porovnáme všechny klasifikátory a ukážeme si, jak kvalitní mají výsledky na reálných datech. Nakonec analyzujeme schopnosti klasifikátorů dodávat kontext incidentů.

Klasifikátory s nejlepšími výsledky budeme dále zkoumat. Nejprve porovnáme jejich schopnost udržet precision a recall v čase. Následně se zaměříme na důležitosti features a ukážeme si, že stačí poměrně malá množina features pro plnohodnotnou klasifikaci. Na základě experimentů nakonec navrhneme koncept detekčního systému poskytujícího kontext detekovaných incidentů.

Klíčová slova síťová bezpečnost, kontext incidentů, klasifikace, FISM, LISM, FP-Growth, CTA

Abstract

This paper discusses the possibilities of using the knowledge engineering for detecting network incidents and extracting its context. In the first part we will focus on the current systems of automated incident detection, especially the CTA detection framework, whose data will serve as a source for experiments.

In the next part we will focus on extraction and evaluation of rules. Introducing the FISM (Frequent Itemset Mining) algorithms (specifically its version FP-Growth) and LISM (Logical Itemset Mining). We suggest an effective way of evaluating these rules using the prefix tree.

We also compare existing contextual and context-free classifiers used in this area (kNN, SVC (Suppor Vector Classifier), Naive Bayes, neural networks, random forests). First, we optimize the parameters of these classifiers on a limited set of data. Next, we compare all classifiers and show how good the results are on real data. Finally, we analyze the capabilities of classifiers to provide the context of incidents.

We will continue to investigate the best-performing classifiers. First we compare their ability to maintain precision and recall over time. Subsequently, we will focus on the importance of features and show that a relatively small set of features for a full-fledged classification is sufficient. On the basis of experiments, we will finally propose a concept of a detection system that provides the context of detected incidents.

Keywords network security, context of incidents, classification, FISM, LISM, FP-Growth, CTA

Obsah

Seznam obrázků

Seznam tabulek

Úvod

V roce 1955 byl na Massachusetts Institute of Technology (MIT) poprvé definován pojem `hack`, popisující „rozčilování se přístroji“ (v originále „fussing with machines“) [1]. Od té doby bylo toto slovo skloňováno v ohromném množství případů, kdy `hack` způsobil větší či menší škody různým cílům. Uveďme si zde 3 největší kyberútoky historie (dle počtu uniklých účtů):

- Společnost Yahoo v roce 2016 oznámila, že se stala obětí největšího úniku osobních dat v historii. Incidenty proběhly v letech 2013 a 2014 v rámci několika útoků. Celkem byly ukradeny informace o 3 miliardách účtů. Tento útok způsobil pokles prodejní ceny podniku o 350 milionů dolarů. [2]
- 427 milionů účtů bylo v roce 2016 odcizeno během kybernetického útoku na společnost MySpace. Odcizeny byly účtu spolu s hesly zabezpečenými SHA-1 algoritmem. [3]
- V roce 2016 byl úspěšně proveden útok na firmu FriendFinder Networks Inc. V rámci tohoto útoku bylo odcizeno 412 milionů uživatelských účtů včetně hesel. O rok dříve tato stejná firma zažila útok menšího rozsahu, kdy byly odcizeny informace o 3 milionech účtů. [4]

Podle zprávy organizace Accenture za rok 2017 (viz [5]) se během let zvyšují náklady, které organizace musí platit v návaznosti na utrpěné kybernetické útoky. V roce 2017 je to stálo o necelých 63% více, než v roce 2013.

Výše uvedená data svědčí o tom, že kyberkriminalita představuje pro organizace vážný problém, který nelze jednoduše přehlížet. Ty si to pomalu začínají uvědomovat, a tak meziročně stále více investují do oblasti informační bezpečnosti. V roce 2016 organizace celosvětově investovaly přes 80 miliard dolarů, v roce 2018 se očekává navýšení až na 93 miliard dolarů [6].

Jelikož organizace stále více zvyšují rozpočet na počítačovou bezpečnost, logicky očekávají, že systémy informační bezpečnosti budou výrazně snižovat dopady kyberútoků. Kromě vhodné detekce nebezpečí je velice důležitá

rychlost reakce. K zajištění okamžité reakce na případný útok slouží monitoring vnitřních sítí organizace. Monitoring produkuje enormní množství dat, která musí být zpracována pro bezpečnostního experta. Toto zpracování obnáší předzpracování dat a klasifikaci.

Klasifikace určí, zda jde o normální provoz, nebo potenciální hrozbu. V případě hrozby informuje systém bezpečnostního experta a poskytne mu informace o útoku, který byl právě odhalen. Standardní bezkontextová klasifikace pouze označí, že nějaký počítač se chová jako infikovaný, vzácně navíc určí i druh infekce. Kontextová klasifikace navíc přidává důležité informace obsahující důvod, proč byl nějaký počítač označen jako infikovaný. Uveďme si příklad kontextu: chování počítače bylo vyhodnoceno jako infikované protože:

- počítač si stáhl binární soubor z DGA (Domain Generating Algorithm) domény;
- začal používat jiného webového klienta, než používal dosud;
- začal napřímo komunikovat s velkým množstvím IP adres po celém světě.

Bezpečností expert tak dostává komplexnější informace o incidentu a může provést podrobnou analýzu a z ní vyplývající opatření. Právě tématem vhodné extrakce kontextu síťového útoku se zabývá tato práce. Hodnotíme jednotlivé klasifikátory nejen z pohledu přesnosti (precision) a úplnosti (recall) klasifikace, ale také z pohledu schopnosti extrahovat kontext daného útoku.

Na začátku práce si představíme problémem automatizovaného odhalování síťových útoků a ukážeme si dva moderní systémy řešící tento problém, konkrétně systémy MINDS (Minnesota Intrusion Detection System) a CTA (Cognitive Threat Analytics). Framework CTA nám poslouží jako zdroj dat pro experimenty. Tato data si představíme podrobněji. Především jejich strukturu a důležitost jednotlivých features pro rodiny útoků.

V další části se zaměříme na pravidla, jejich extrakci a evaluaci. Představíme si algoritmy pro vytěžování asociačních pravidel z dat FISM (Frequent Itemset Mining), jeho variantu FP-Growth a algoritmus LISM (Logical Itemset Mining). Pro efektivní vyhodnocování pravidel navrhneme prefixový evaluační strom.

V další části optimalizujeme parametry klasifikátorů a zjistíme, jak dobře si vedou na reálných datech. Porovnávat budeme několik známých klasifikátorů, jako jsou kNN, neuronové sítě, náhodné lesy či SVM z hlediska precision, recall a doby běhu jednotlivých klasifikátorů. Důležitým aspektem pro nás bude schopnost klasifikátorů extrahovat kontext detekovaného incidentu.

Vybrané klasifikátory s dobrými vlastnostmi podrobíme zevrubnějšímu testování. Změříme jejich schopnost držet své vlastnosti v čase a v redukovaném prostoru features. Na základě výše uvedených experimentů poté navrhneme vlastní systém schopný detekce síťových incidentů a extrakce jejich kontextu.

Kapitola 1

Úvod do problematiky

V následující kapitole si nadefinujeme základní pojmy a ukážeme si reprezentanty systémů automatizované detekce útoků. Představíme také problém automatizovaného odhalování bezpečnostních incidentů. Rozbor systémů MINDS (Minnesota Intrusion Detection System) a CTA (Cognitive Threat Analytics) ukáže, jak lze tento problém řešit v praxi. Framework CTA hraje v této práci významnou roli, protože při experimentech budeme vycházet z dat právě tohoto frameworku.

1.1 Problém automatizovaného odhalování bezpečnostních incidentů

1.1.1 Definice bezpečnostního incidentu

Pojmem bezpečnostní incident a kybernetická bezpečnostní událost si v této práci nadefinujeme ve shodě se zákonem o kybernetické bezpečnosti §7 [7]. Kybernetickým bezpečnostním incidentem nazveme narušení bezpečnosti informací v informačních systémech nebo narušení bezpečnosti služeb anebo bezpečnosti a integrity sítí elektronických komunikací v důsledku kybernetické bezpečnostní události. Za kybernetickou bezpečnostní událost označíme takovou událost, která může způsobit narušení bezpečnosti informací v informačních systémech nebo narušení bezpečnosti služeb anebo integrity sítí elektronických komunikací.

1.1.2 Definice systému automatizované detekce bezpečnostních incidentů

Systém pro automatizované detekce definujeme jako systém, který dokáže bez vnějšího zásahu člověka identifikovat bezpečnostní incidenty a vhodným způsobem na ně reagovat (například vyvoláním poplachu a vytvořením reportu).

Tyto systémy můžeme rozdělit podle metod jakými jsou incidenty odhalovány:

- systémy založené na pravidlech (rule-based) používají množinu pravidel, která definuje potenciálně nebezpečné události;
- systémy založené na detekci anomálií (anomally-based) sledují odchylky od normálního stavu/provozu.

Dále mohou být systémy rozděleny podle místa působení:

- systémy sledující události na cílovém zařízení (host-based) vyhodnocující nebezpečnosti událostí (nebo jejich posloupností) pouze na daném zařízení;
- systémy sledující provoz na síti (network-based) vyhodnocující nebezpečnost síťového provozu.

1.1.3 Historie automatizované detekce bezpečnostních incidentů

Myšlenku automatizovaného odhalování počítačových hrozeb připisuje historie Jamesovi P. Andersonovi díky jeho článku „Monitorování a sledování počítačových bezpečnostních hrozeb“ (v originále „Computer Security Threat Monitoring and Surveillance“), který vydal v roce 1980 [8]. Anderson vycházel z potřeby zajistit bezpečnost počítačových sítí letectva Spojených států amerických.

Mezi lety 1984 až 1986 byl vytvořen první model ISD (Intrusion Detection System) pracující v reálném čase. Šlo o systém vytvořený Dorothy Denningovou a Peterem Neumannem [9]. Na počátku šlo o systém založený na pravidlech (tzv. rule-based systém), který vyhodnocoval auditní data z počítače (host-based).

Todd Heberlein spolu se svým týmem představili jako první detekční systém, který detekoval bezpečnostní incidenty z provozu na síti, tzv. „Network Security Monitor“ [10]. Článek vyšel v roce 1990 a jeho hlavním přínosem byl důkaz, že techniky pro odhalování bezpečnostních incidentů v počítači (host-based) lze s mírnými změnami aplikovat i na síťový provoz. Heberlein tak odstartoval novou éru automatizovaných detekčních systémů monitorujících počítačové sítě (network-based).

V druhé polovině devadesátých let se objevuje tendence opustit detekční systémy založené na pravidlech. Jejich problémem je, že nedokáží reagovat na nové hrozby. Proto přichází myšlena detekce síťových anomálií, tedy odchylek od normálního provozu na síti. Tento systém na jednu stranu může detekovat spoustu falešně pozitivních nálezů (provoz označen za hrozbu, přestože o ni ve skutečnosti nejde), na druhou stranu ale dokáže nacházet nové a doposud

neznámé hrozby. Příkladem může být systém MINDS, který si v následující sekci rozebereme podrobněji.

Kvůli rozmachu nových technologií na začátku nového tisíciletí trpěly staré systémy vysokou mírou falešně pozitivních nálezů. Opětovný rozmach přinesla až éra cloudových počítačů [11]. Ty jsou tak komplexní, že manuální bezpečnost není možná. Navíc ohromné množství dat přítomných v cloudech (tzv. „big data“) slouží detekčním systémům jako dobrý podklad k efektivnímu učení.

1.2 MINDS

V následující sekci si popíšeme funkci systému MINDS (Minnesota Intrusion Detection System). Následně budeme mít možnost porovnat jeho způsob řešení detekce incidentů s modernějším systémem CTA (Cognitive Threats Analytics).

1.2.1 Základní popis

MINDS byl vyvíjen na Minnesotské univerzitě a finální článek popisující jeho funkci vyšel v roce 2004 [12]. Jde o systém pro odhalování bezpečnostních incidentů na síti (network-based). K odhalování těchto incidentů využívá technik hledání odchylek od standardního provozu (anomally-based). Tyto odchylky získává pomocí zpracovávání naměřených síťových dat. Pro získání smysluplných informací ze získaných dat jsou využívány techniky data miningu.

Mezi hlavní přínosy MINDS patří (kromě samotné detekce síťových anomálií) také výpočet skóre popisující, jak anomální dané připojení je. Veškeré učení probíhá tzv. bez učitele, tedy bez vnějšího dodávání informace o správnosti detekce (unsupervised learning). Na obrázku 1.1 můžeme vidět základní elementy systému. V následujících podsekcích si rozebereme jejich význam.

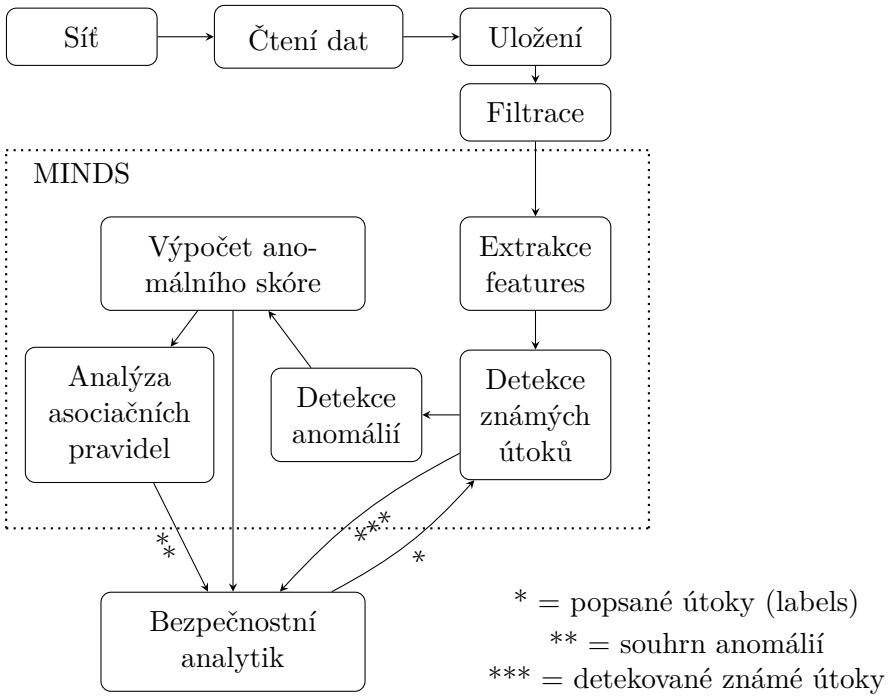
1.2.2 Vstupní data

Vstupem do MINDS systému jsou Netflows ve verzi 5. Netflow označuje protokol společnosti Cisco, který slouží k monitorování sítě pomocí IP toků [13]. IP tok označuje sekvenci paketů, které mají shodné následující parametry:

- zdrojová a cílová IP,
- zdrojový a cílový port,
- číslo protokolu.

Netflow tedy nepřihlíží k obsahu paketu, ale zabývá se pouze hlavičkou. Netflow záznam potom obsahuje důležité informace o IP toku, jako například:

- čas začátku a konce toku,



Obrázek 1.1: Základní elementy systému MINDS.

- zdrojová a cílová IP adresa,
- zdrojový a cílový port,
- agregace příznaků,
- počet paketů v toku.

Netflows mohou potom sloužit k měření statistik, využití sítí, nebo právě k odhalování síťových útoků.

Jak můžeme vidět na schématu 1.1, do systému vstupují filtrovaná data. Filtrace vymaže provoz, který není třeba dále zpracovávat. Typickým příkladem mohou být toky, které byly v minulosti označeny za neškodné.

Vyfiltrováné toky jsou ukládány a systémem zpracovávány v dávkách, v časovém okně dlouhém 10 minut.

1.2.3 Extrakce features

První část MINDS obstarává extrakci features. Ty jsou dále využívány v dataminingové analýze. Jako features označují autoři následující položky:

- zdrojovou a cílovou IP adresu,

- zdrojový a cílový port,
- protokol,
- pole příznaků,
- počet bajtů,
- počet paketů v session.

Z těchto základních se odvozují features založené na časovém okně. Tyto agregují připojení s podobnými charakteristikami v posledních `T` sekundách. Sleduje se počet toků, které jsou tvořeny nebo přijímány jednou adresou nebo portem. Můžou být použity například k detekci zdrojů vytvářejících velké množství spojení v krátkém čase.

Časové features nedokáží popsat dlouhodobější aktivity. Tvůrci proto přidali features založené na okně připojení. Agregují `N` posledních připojení ze specifického zdroje na specifický cíl. Využití mohou najít například pro odhalování pomalého skenování sítě.

1.2.4 Detekce známých útoků

V této fázi systém porovnává síťové připojení se signaturami existujících incidentů, které má systém k dispozici. V případě nalezení shody se vyvolá poplach, jenž se předává ke zpracování bezpečnostním analytikem. Tyto připojení dále systém nezpracovává.

1.2.5 Detekce anomálií

Tento modul lze označit za jeden z nejdůležitějších částí celého MINDS, protože zvládá hledat anomální (a tím pádem i podezřelé) toky. Jde o takové případy, které se výrazně odlišují od ostatních toků (tzv. outlieři).

Možností, jak hledat outliery v datech, existuje veliké množství. Systém MINDS používá model založený na vzdálenosti (proximity-based model), přesněji model založený na hustotě (density-based model). Pro každý element počítá *LOF* (Local Outlier Factor). Ten se podle [14] vypočítá:

$$LOF_k(p) = \frac{\sum_{o \in N_k(p))} \frac{lrd_k(o))}{lrd_k(p)}}{|N_k(p)|},$$

kde $lrd_k(p)$ vypočítáme jako:

$$lrd_k(p) = \frac{1}{\frac{\sum_{o \in N_k(p))} reachDistance_k(p,o)}{|N_k(p)|}},$$

a kde $N_k(p)$ označuje množinu k nejbližších sousedů bodu p. Zmíněnou funkci $reachDistance_k(p, o)$ dále vypočítáme jako

$$reachDistance_k(p, o) = max\left\{kDistance(o), d(p, o)\right\}.$$

Funkce $kDistance(o)$ označuje vzdálenost mezi k-tým nejbližším sousedem o a bodem o samotným. Funkce $d(p, o)$ znamená vzdálenost mezi body p a o.

Výpočety $LOF_k(p)$ pro všechny body jsou výpočetně náročné, protože musíme zjistit okolí každého bodu. K tomu musí být provedeno $\mathcal{O}(n^2)$ výpočtů. Proto tvůrci vzorkují pomocí m bodů, kde $m \ll n$. Všechny body si spočítají vzdálenost k těmto m prvkům a na základě těchto vzdáleností se poté určí okolí jednotlivých bodů. Tímto přístupem se zlepší složitost na $\mathcal{O}(n*m)$. Tento přístup také výrazně sníží možnost, že anomální provoz vygeneruje dostatek bodů datasetu tak, že ho algoritmus bude považovat za normální.

LOF funkce má tu výhodu, že počítá s lokální hustotou okolí bodu. Dokáže tedy lépe detekovat lokální outliery, které by v globálním pohledu za outliery algortimus neoznačil. Toky s vysokým *LOF* systém označí za anomální a jsou předány bezpečnostnímu analytikovi k dalšímu přezkoumání.

1.2.6 Analýza asociačních pravidel

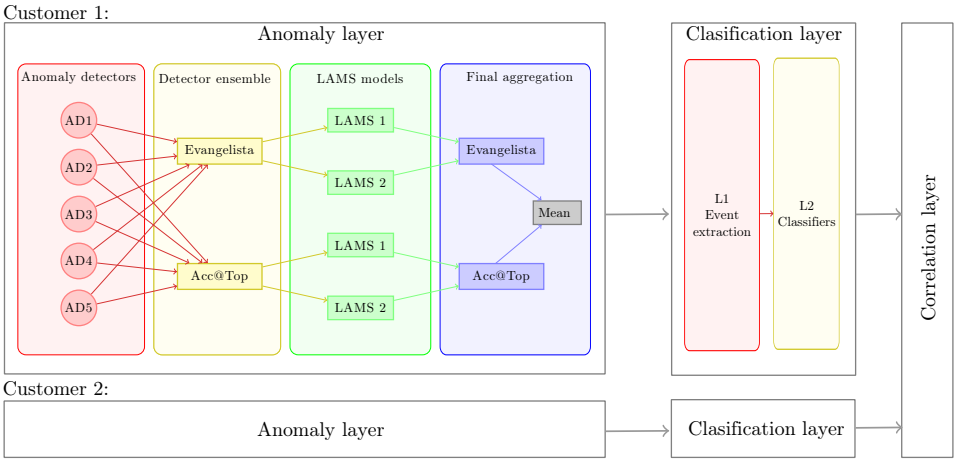
Druhou významnou součást MINDS představuje modul pro vytváření asociačních pravidel. Asociační pravidla lze popsat jako implikace ve tvaru $X \Rightarrow Y$. Více o asociačních pravidel v podsekci 3.1.1.1, kde jsou popsána v souvislosti s algoritmem FISM. Takto extrahovaná pravidla mají potom následující využití:

- ke konstrukci pravidel popisujících anomální provoz detekovaný systémem;
- ke konstrukci pravidel popisujících normální provozu na síti;
- dle opakovaných pravidel v normálním i anomálním provozu lze vytvořit nové features, které jsou použity k zpřesnění předpovědních modelů.

MINDS extrahuje pravidla ze síťového provozu v případě, kdy se jejich support a confidence nachází nad zvolenými prahy, definice supportu a confidence lze nalézt v podsekci 3.1.1.1.

1.2.7 Úspěšnost systému

MINDS se ve své době ukázal jako velmi efektivní v odhalování bezpečnostních incidentů. Tehdejší konkurenční komerční řešení byla většinou založena na pravidlech a signaturách (rule-based, signature-based), a proto nedokázala tak dobře reagovat na nové hrozby jako právě MINDS.



Obrázek 1.2: Schéma frameworku CTA. Zdroj detailu anomální vrstvy [15].

Příkladem může být detekce červa Slammer/Sapphire již 27. ledna 2002, tedy 2 dny poté, co byl červ vypuštěn. MINDS to dokázal díky detekci anomálního provozu na síti, kdy se podezřele mnoho nakažených počítačů mimo síť opakovaně pokoušelo komunikovat s počítači uvnitř sítě. Dále také detekoval pokusy o skenování interní sítě. Komerční detekční systém SNORT připojený do stejné sítě nebyl schopen tuto hrozbu detekovat.

1.3 CTA

Framework Cognitive Threat Analytic nám poslouží jako zdroj dat pro tuto práci. Proto si ho v následující sekci detailně popíšeme.

1.3.1 Základní informace o frameworku

Framework CTA byl vyvíjen společností Cognitive Security od roku 2009. Když v roce 2013 firma Cisco tuto společnost koupila, přešel i další vývoj frameworku pod křídla této americké společnosti.

Jako vstup frameworku slouží kromě NetFlows (které byly představeny v podsekci 1.2.2) také proxy logy, pokud jsou k dispozici. Ty jsou typicky generované webovým proxy serverem umístěným na perimetru sítě a obsahují informace o HTTP(S) požadavcích. K tomu navíc CTA dokáže pracovat i s dalšími zdroji informací, které mají základní flow features. Na rozdíl od MINDS [15] má tedy CTA k dispozici mnohem více zdrojů informací.

Jak můžeme vidět na schématu 1.2 framework se sestává z následujících vrstev:

1. anomální vrstva,

2. klasifikační vrstva,

3. korelační vrstva.

V následujících podsekcích si funkci těchto vrstev popíšeme podrobněji.

1.3.2 Anomální vrstva

Anomální vrstva má na starost výběr množiny nejvíce anomálních dat k dalšímu zpracování. Schéma této vrstvy můžeme najít v obrázku 1.2. Systém nepoužívá jeden výkonný anomální detektor. Místo toho používá kolem sedmdesáti jednodušších anomálních detektorů. Jejich výstupy jsou agregovány a pro každou vstupní sekvenci se počítá skóre odpovídající tomu, jak jsou dané sekvence anomální. Tento přístupu přináší vznik robustnějšího systému, který zvládá dobře odhalovat mnoho druhů anomálií a zároveň zmenšuje význam chyb vzniklých na malém počtu detektorů. Za nespornou výhodou lze také označit obtížnost vytvoření malwaru, který by byl speciálně navržený k obcházení všech druhů anomálních detektorů zároveň.

Detektory této vrstvy se dělí do pěti kategorií:

- **Statistické anomální detektory** které využívají přímé aplikace statistických metod, například PCA (Principal Component Analysis).

- **Detektory založené na časových řadách**, které modelují standardní chování uživatele a detekují odchylky od vytvořeného modelu. Takto například může být rozpoznána aktivita v atypickou hodinu.

- **Detektory založené na WebFlows** které detekují odchylky od standardního chování uživatele na webu. Tyto detektory například označí za anomálii situaci, kdy první do prohlížeče zadaná adresa přesahuje jistou délku (nejde o standardní uživatelské chování). Dalším příkladem může být použití prohlížeče nižší verze, než odpovídá verzi uživatelem standartě používaného prohlížeče. Vyšší verze znamená upgrade a jde o standardní chování.

- **Detektory založené na NetFlows** které jsou založené na detekci odchylek od typického dění na síti. Detektory této kategorie jsou schopné například rozpoznat SPAM bota, který začne ve zvýšené míře komunikovat s mailovým serverem. Dalším příkladem může být detekce horizontálního skenování sítě.

- **Detektory založené na modelování služeb** zkoumající typické chování cloudových služeb a detekující odchylky v tomto chování. Takto například detektor rozpoznává ransomware, který začne šifrovat složku zálohovanou ve službě Dropbox. Zašifrování složky totiž vyvolá změnu obsažených souborů, které se zálohují do cloudu.

Výstupy detektorů jsou následně agregovány. Agregace má za cíl zachovat počet TP (True Positive) nálezů a redukovat počet FP (False Positive) nálezů, tedy chyb jednotlivých detektorů. Tohoto efektu může být dosaženo zprůměrováním jednotlivých výstupů. To ale lze použít pouze pro detektory se stejným rozsahem hodnot, což ovšem detektory anomální vrstvy nesplňují. Proto se zde používá agregace založená na článku Paula F. Evangelisty, viz [16]. Ta počítá průměr z průměrů a maxim jednotlivých detektorů, jak můžeme vidět v následujícím vzorci:

$$H_{evan}(x) = \frac{1}{2}\left(\frac{1}{2}\sum_{i=1}^{m} h_i(x) + \max_i h_i(x)\right),$$

kde $\mathcal{H}_m = \{h_i : \mathcal{X} \mapsto \mathbb{R}\}_{i=1}^{m}$ odpovídá množina m anomálních detektorů. Tato agregace upřednostňuje vyšší anomální skóre a omezuje vliv slabých detektorů. Paralelně s touto agregací se používá také agregace nazvaná Acc@Top [15], která funguje na principu vážených průměrů, jejichž váhy jsou optimalizovány. K optimalizaci dochází pomocí učení na olabelovaných datech (supervised learning).

Výstup z první agregace vstupuje do tzv. LAMS (Local Adaptive Multivariate Smoothing) modelů. Ty jsou zodpovědné za další redukci FP nálezů pomocí vyhlazování výsledků agregací v čase. Tím vytváří ještě robustnější odhad skutečných anomálií. Těchto modelů používá framework několik. Jednotlivé modely se liší dle podmnožiny features. Výstupy těchto LAMS modelů jsou opět agregovány stejným způsobem jako výstupy detektorů.

Anomální vrstva tedy postupně redukuje počet FP nálezů a zachovává počet TP nálezů. Ke každému nálezu se vypočítá jeho anomální skóre. Do klasifikační vrstvy potom pokračuje 10% nejvíce anomálních spojení.

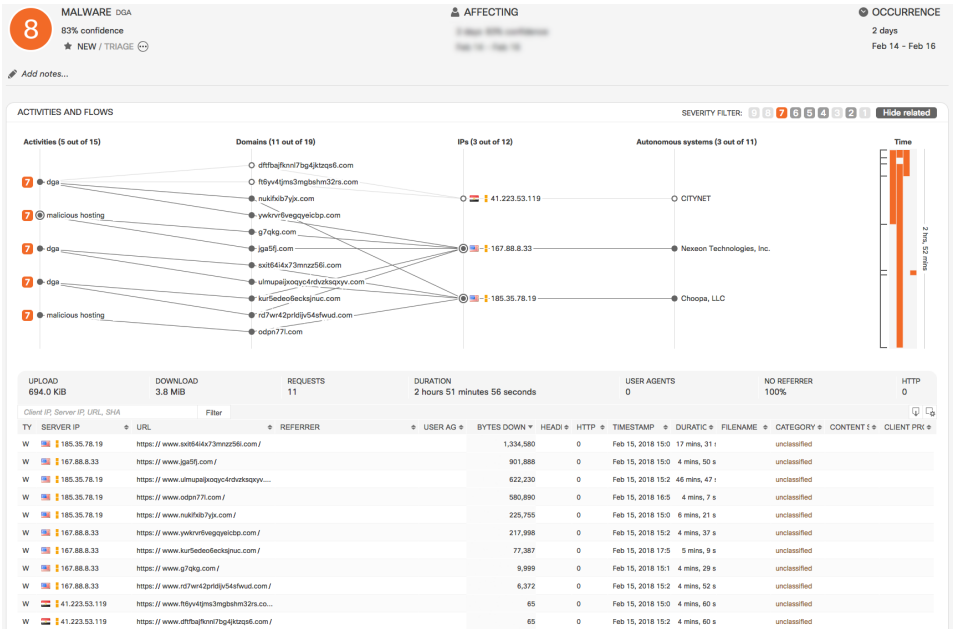
1.3.3 Klasifikační vrstva

Klasifikační vrstva přijímá za vstup výstup z anomální vrstvy, tedy 10% nejvíce anomálních spojení obohacených o skóre jednotlivých detektorů. Sestává se z následujících podvrstev:

- Vrstva **L1-Extrakce event** zodpovídá za extrakci event, které popisují chování na síti.
- Druhá vrstva **L2-Klasifikace** přijímá eventy z L1 vrstvy a rozhoduje o tom, zda jde o incident, či nikoliv.

V následujícím seznamu můžeme vidět jednotlivé skupiny event:

- **Eventy založené na signaturách** generují eventy, které jsou přesně vytvořeny pro detekci konkrétních malware. Mají vysokou precision ale poměrně malý recall. Příkladem může být eventa `ESALIT`, která označuje detekci malware z rodiny Sality. Detekce malware probíhá tak, že



Obrázek 1.3: Ukázka grafického prostředí frameworku CTA.

parametry hlavičky (URL, referrer, User Agent nebo jejich kombinace) odpovídají pevně zadanému regulárnímu výrazu.

- **Supervised eventy** obsahují klasifikační modely naučené z historických dat. Příkladem může být eventa `ECDGA`. Tato eventa se vytvoří, pokud naučený klasifikátor fungující na principu náhodného lesa označí v provozu doménu za doménu vytvořenou pomocí za DGA algoritmu.

- **Kontextové eventy** popisují kontext, který může posloužit dalším klasifikátorům. Takto například eventa `ERAWIP1` informuje o použití přímo IP pro dotázání na server bez použití DNS serveru.

- **Eventy anomálních hodnot** jsou generovány v případě, že spojení jeví známky anomálie. Takto například může vzniknout eventa `EAVOUT1` informující o tom, že jde o nekonzistentní časovou aktivitu uživatele (tedy například přístup v noci atd).

Výše extrahované eventy poté slouží jako vstup do vrstvy L2 klasifikační vrstvy. Klasifikační vrstva obsahuje různé klasifikátory, jejichž výstup se opět agreguje. Výstupy L2 vrstvy jsou prezentovány zákazníkovi jako nalezené incidenty, jak můžeme vidět na obrázku 1.3. Detail incidentu informuje bezpečnostní analytiky o závažnosti incidentu, jeho typu, jistotě s jakou lze říci, že jde skutečně o incident, a dodává další kontext útoku (proč byl daný provoz vyhodnocen jako bezpečnostní incident). Právě L2 část klasifikační vrstvy se

snažíme v naší práci vylepšit tím, že navrhneme nový klasifikátor, který na daných datech bude mít dobré výsledky a bude schopen podávat i kontext daného útoku.

1.3.4 Korelační vrstva

Korelační vrstva agreguje data všech zákazníků využívajících tento framework. Díky této vrstvě se může zákazník dozvědět, zda daný útok cílí pouze na jeho organizaci, nebo zda jde o problém, se kterým se potýká větší možností zákazníků. Navíc tato vrstva poskytuje možnost porovnávat data napříč jednotlivými instancemi detekčních systémů.

1.3.5 Porovnání MINDS a CTA

Framework CTA staví na podobných základech jako MINDS: sleduje a analyzuje provoz na síti a využívá detekce anomálií oproti standardnímu síťovému provozu. V první řadě tvoří rozdíl větší množství vstupů, které CTA uvažuje. CTA jde také mnohem dále v odhalování anomálií vytvořením většího množství úzce specializovaných anomálních detektorů, jejichž výstupy jsou vhodně agregovány. Kromě anomální používá také klasifikační vrstvu, která využívá klasifikátory pro další zpracovávání vstupních dat. V neposlední řadě CTA obsahuje korelační vrstvu, díky které může korelovat výsledky napříč jednotlivými instancemi detekčních systémů. CTA tedy představuje další článek evoluce moderních IDS.

Kapitola 2

Data a jejich analýza

Jak jsme psali v minulé kapitole, v této práci budeme pro experimenty používat data z frameworku CTA. V této kapitole si je blíže představíme a analyzujeme jejich základní vlastnosti.

2.1 Základní popis dat

Data jsou z období července až září roku 2017 a celkem jde o 11 milionů záznamů. Pro experimenty budeme používat podmnožinu těchto dat, u každého experimentu bude tato podmnožina blíže popsána.

K dispozici máme data ve formátu TSV (Tab-Separated Value), jeden řádek obsahuje čísla oddělená tabulátorem. Strukturu jednotlivých souborů

```
data/data.tsv:
    ...
    191 30  18  16  105 139
    191 18  185 16  105 142 178 112
    39  43  139 144
    191 18  148 105 139 114 175 112 147
    30  18  114 175 112
    191 30  5   18  16  105 49  168 35
    ...

data/evetMapping.txt:
    ...
    ECMCD03 235
    EAVUEH1 107
    ECFSA02 207
    EDTRTU1 293
    ECADI28 143
    ...

data/labels.txt:
    MalwareDistribution 195
    InformationStealer  192
    BankingTrojan   2
    ClickFraud  3
```

Obrázek 2.1: Ukázka struktury datových souborů.

můžeme vidět na obrázku 2.1. Tato čísla jsou identifikátory event, které popisují chování jednoho počítače na síti za posledních 24 hodin. Více o eventách v sekci 1.3.3. Mapování čísel na názvy můžeme najít v textovém souboru `eventMapping.txt`.

Řádky olabelované jako útok v sobě obsahují identifikátor eventy odpovídající útoku. Framework rozlišuje 4 základní rodiny malware:

- `InformationStealer`,
- `MalwareDistribution`,
- `ClickFraud`,
- `BankingTrojan`.

Tuto klasifikaci provedla jiná část frameworku. My je můžeme použít pro hodnocení kvality klasifikace. Seznam všech event, které popisují útoky můžeme nalézt v `labels.txt`. Identifikátory indikující útoky z dat mažeme během načítání vstupního datového souboru. To proto, že slouží pro vyhodnocení kvality klasifikátorů, a proto jim nemohou být k dispozici. Během testování jsme zjistili, že labelování dat není dokonalé. Některé útoky nejsou olabelovány, protože v době detekce ještě nebyly jako útoky frameworkem rozpoznány. Proto výsledky jednotlivých experimentů práce představují spodní mez, po zlepšení labelů dat bude precision a recall klasifikace lepší.

V následující kapitole si popíšeme features a podrobněji se podíváme na jednotlivé rodiny malware.

2.2 Features

Jednotlivé záznamy dat si interně budeme reprezentovat vektorem pravdivostních hodnot. *True* na `k`-té pozici vektoru znamená, že daný záznam obsahuje `k`-tou eventu, *False* opak. Velikost tohoto vektoru odpovídá celkovému počtu event, které se v datech vyskytují, v našem případě 202. Záznamy tedy tvoří body v 202-dimenzionálním prostoru. Souřadnice bodu v tomto prostoru budeme nazývat *features*.

Je zřejmé, že všechny features nejsou stejně důležité z pohledu klasifikace. Jejich vliv na rozhodnutí klasifikátoru, zda jde o útok či nikoliv, se výrazně liší. Vezměme si dvojici event `EBLKEV` označující, že u uživatele byl detekován škodlivý spustitelný soubor a `EENCON`, tedy eventa popisující použití šifrovaného spojení. Eventa `EBLKEV` je pro odhalování útoku důležitější než eventa `EENCON`, protože se mnohem častěji váže na skutečný útok. Šifrované spojení je totiž tak obvyklé, že mnoho dalších informací klasifikaci nepřidá. Pomocí podmíněných pravděpodobností můžeme tento fakt zapsat jako:

$$P(D\,is\,attack \,|\, \texttt{EBLKEV} \in D) > P(D\,is\,attack \,|\, \texttt{EENCON} \in D).$$

2.2.1 Důležitost features

Důležitost jednotlivých features je pro tuto práci významná. Vypovídá o kontextu, ve kterém se daný útok většinou odehrává. Nejdůležitější features reprezentují nejčastější kontext. Ukážeme si tedy, jak se důležitost zjišťuje, a představíme si nejdůležitější features jednotlivých rodin malware.

Důležitost features můžeme zjistit několika metodami. Jednou z nich je náhodné permutování hodnot v jednom sloupci (hodnot jedné feature dat) a sledování, jaký vliv má toto permutování na precision a recall klasifikace. Čím větší je tento vliv, tím důležitější je daná feature. Více o této metodě lze nalézt v článku [17].

Další metodou je využití vlastností rozhodovacího stromu. Metoda se nazývá „Gini Importance“ nebo také „Mean Decrease Impurity“ a spočívá v počítání příspěvků jednotlivých features ke snižování znečištění uzlů („decrease in node impurity“) stromů. „Node impurity“ vyjadřuje míru toho, jak hodně uzel data rozděluje. Typickou metrikou měřící tuto míru je „Gini impurity“, kterou vypočítáme jako:

$$I_G(p) = 1 - \sum_{i=1}^{J} p_i^2,$$

kde $i \in \{1, 2, ...J\}$, J odpovídá počtu tříd a p_i označuje zlomek prvků olabelovaných uzlem jako i. Podrobněji je tento algoritmus popsán v knize L. Breimana „Classification and regression trees“ z roku 1984 [18]. Problémem této metody je, že důležité features mohou být i rozhodovací uzly, které pro danou rodinu vždy rozhodnou negativně. Více je tento problém popsán v části 5.1.

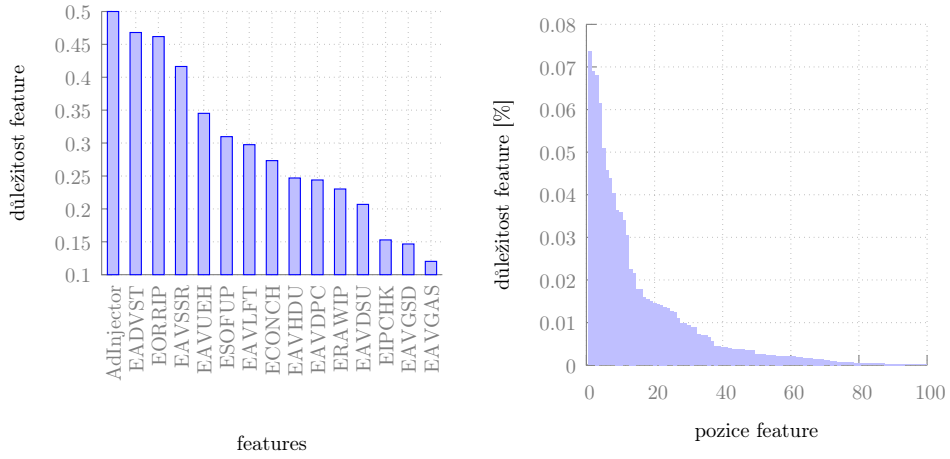
Poslední metodou, kterou si zde představíme, je algoritmus „Explainer“ popsaný v článku [19]. Algoritmus také vychází z náhodných lesů, ale oproti předchozímu algoritmu přináší několik výhod. Tou první je možnost zaměřit se pouze na pozitivní features, tedy features v rozhodovacích uzlech stromu, které jsou důležité při pozitivním rozhodování pro danou rodinu. Jeho další vlastností je schopnost učit se z poměrně malého množství vstupních dat. Výstupy z tohoto algoritmu použijeme v další podsekci k analýze důležitosti features pro jednotlivé rodiny malware.

2.2.2 Důležitost features pro jednotlivé rodiny malware

V této části se blíže zaměříme na rodiny malware a zjistíme míru shody důležitých features jednotlivých rodin malware. Důležitosti jednotlivých features byly získány algoritmem Explainer popsaným výše. Ten vrací jednotlivým features skóre odpovídající jejich důležitosti.

2.2.2.1 `InformationStealer`

Skupina `InformationStealer` popisuje všechen malware, který nějakým způsobem odcizuje uživateli data. Na dvojici grafů 2.2 můžeme vidět 15 nejdůle-



Obrázek 2.2: Nejdůležitější features a graf rychlosti klesání důležitosti feature pro rodiny `InformationStealer`.

žitějších event pro tuto skupinu útoků, které zároveň představují nejčastější kontext. Na druhém grafu můžeme vidět, jak postupně klesá přispění jednotlivých features. Features s nenulovou důležitostí má tato rodina 129. 50% důležitosti zastupuje prvních 10 features a na 95% se dostaneme pomocí 50 features. Z těchto čísel můžeme vidět, že důležitost features klesá téměř exponenciálně. Velké množství features nepřispívá ke klasifikaci téměř vůbec. Stejný jev můžeme pozorovat u všech rodin malware.

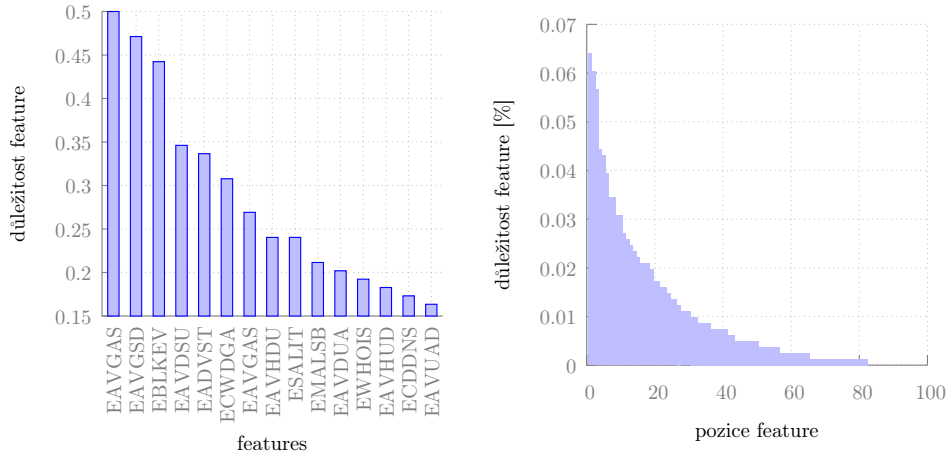
Tři nejdůležitější eventy této rodiny jsou:

- `AdInjector`, eventa popisující detekci škodlivého kódu vkládajícího reklamy;
- `EADVST`, eventa popisující návštěvu reklamní stránky;
- `EORRIP`, popisující kontaktování serveru napřímo bez DNS serveru.

Tyto tři eventy popisují nejčastější kontext, ve kterém je tento útok detekován. Mohou typicky popisovat chování virem nakaženého uživatelského počítače, který se nejprve dotázal C&C (Command and Control) serveru (pomocí raw IP), jakou reklamu má uživateli zobrazit. Následně ji injektoval do legitimní stránky a uživatel na ni klikl. Tím došlo k návštěvě reklamní stránky.

2.2.2.2 `MalwareDistribution`

`MalwareDistribution` reprezentuje detekci programů, které se podílejí na distribuci malware. Důležitosti features týkající se této kategorie můžeme najít na dvojici grafu 2.3. Míra klesání důležitosti u této rodiny je menší než



Obrázek 2.3: Nejdůležitější features a graf rychlosti klesání důležitosti feature pro skupiny `MalwareDistribution`.

u `InformationStealer`. Z celkového počtu 82 nenulových features zastupuje 50% důležitosti prvních 13 features. 95% dosáhneme s využitím 55 features.

Třemi nejdůležitějšími eventami jsou:

- `EAVGAS` a `EAVGSD`, eventy popisující návštěvu neobvyklého autonomního systému (`EAVGAS`) nebo domény (`EAVGSD`);
- `EBLKEV`, popisující detekci škodlivého spustitelného kódu.

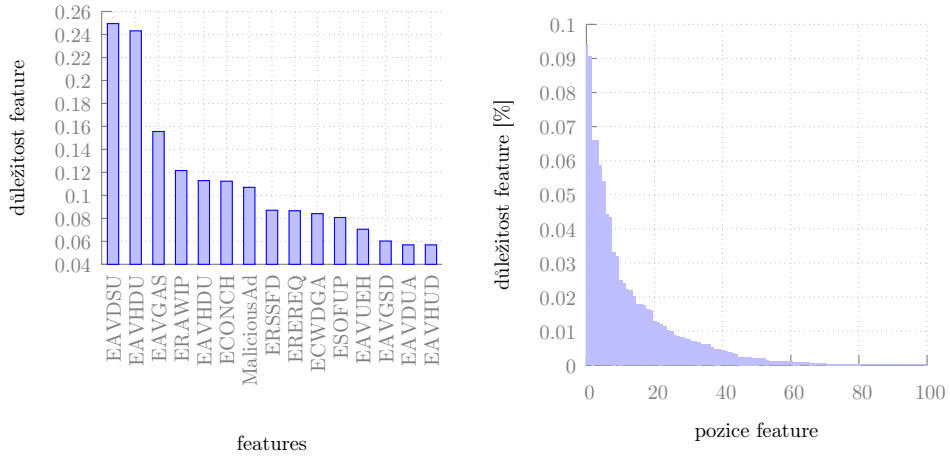
Tyto eventy mohou popisovat kontext viru z rodiny `MalwareDistribution`, přesněji stažení viru z neobvyklé lokace. Nebo naopak může popisovat již probíhající útok, kdy malware kontaktuje C&C server na neobvyklé lokaci za účelem získání dalších úkolů nebo stažení dalších virů.

2.2.2.3 `ClickFraud`

Útoky spadající do kategorie `ClickFraud` vyvolávají u uživatele nevyžádané klikání, které způsobuje nechtěné akce. Typicky může jít o klikání na reklamy, přidávání obsahu na sociálních sítích pod identitou napadeného uživatele, odesílání phishingových emailů atp.

Detaily o důležitosti features této skupin můžeme najít na dvojici grafů 2.4. Rodina má celkem 113 features s nenulovou důležitostí a 50% důležitosti u této skupiny zastupuje prvních 10 nejdůležitějších features. 95% dosáhneme pomocí 57 features.

Nejdůležitějšími eventami této rodiny jsou:



Obrázek 2.4: Nejdůležitější features a graf rychlosti klesání důležitosti feature pro skupiny `ClickFraud`.

- `EAVDSU` popisující detekci tzv. „shadow-user“, tedy programu běžícího na uživatelově počítači, snažícího se vyvolávat akce tak, aby zdály být prováděny uživatelem;
- `EAVHDU` eventa popisující naprosto neobvyklou doménu pro uživatele;
- `EAVGAS` eventy popisující návštěvu neobvyklého autonomního systému.

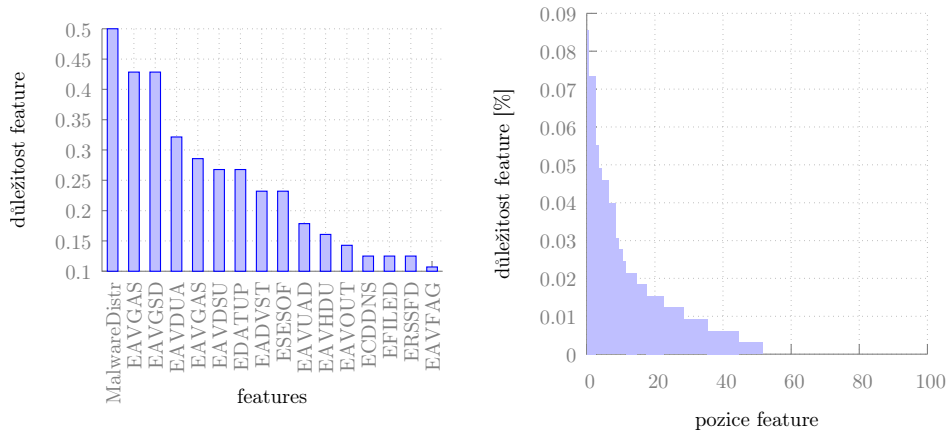
Eventa `EAVDSU` je typická pro malware rodiny `ClickFraud`, protože simulace chování uživatele je jeho hlavním úkolem. Návštěva neobvyklé stránky pak může znamenat začátek útoku na specifickou stránku.

2.2.2.4 `BankingTrojan`

Kategorie `BankingTrojan` obsahuje množinu virů, které útočí na internetová bankovnictví uživatelů. Mohou se například pokoušet ukrást uživatelovy přístupové údaje. Skupina `BankingTrojan` má nejrychleji klesající důležitost features mezi všemi skupinami, což můžeme vidět na grafech 2.5. Má pouze 51 features s nenulovou důležitostí, z čehož 50% důležitosti u této skupiny zastupuje prvních 9 features. 95% dosáhneme pomocí 41 features.

Tři nejdůležitější eventy jsou:

- `MalwareDistribution`, tedy přímo label detekovaného útoku z jiné rodiny.
- `EAVGAS` a `EAVGSD`, eventy popisující návštěvu neobvyklého autonomního systému (`EAVGAS`) nebo domény (`EAVGSD`);



Obrázek 2.5: Nejdůležitější features a graf rychlosti klesání důležitosti feature pro skupiny `BankingTrojan`.

U této rodiny je zajímavou eventou jiný label. U virů typu `BankingTrojan` totiž často nejprve dojde k nakažení jiným typem viru, tzv. „Dropperem“. Ten stáhne potřebné další moduly a pouze v případě, že je detekována možnost zaútočit na online bankovnictví, se stáhne malware z rodiny `BankingTrojan`.

2.2.2.5 Důležitost features

V předchozí podsekci jsme si ukázali, že různé features mají různou důležitost pro klasifikaci dat. Navíc nejvíce důležitých je vždy jen několik features (dle našich experimentů 10-15) a většina features přispívá jen velice malým dílem k celkovému rozhodování (v našem případě 100 features odpovídá méně než 5% důležitosti). Navíc redukce dimenzionality prostoru features, ve kterém se data nachází, přináší zmenšení výpočetní složitosti klasifikace. Z těchto faktů logicky vyplývá, že je vhodné omezit počet uvažovaných features. Závislost kvality výsledků při postupné redukci prostoru features můžeme najít v sekci 6.2.

Dále jsme si ukázali, že důležitosti features jsou rozdílné pro různé skupiny malware. Obecný výběr features tedy není vhodný, protože může upřednostňovat často se vyskytující skupiny a jejich důležité features.

Kapitola 3

Extrakce a evaluace pravidel

V následující kapitole si pro naše data extrahujeme pravidla popisující bezpečností incidenty a navrhneme způsob jejich efektivní evaluace. Tento přístup má výhodu, že pravidlo, které provoz za útok označí, přímo definuje i kontext daného útoku. V první části představíme 2 principy extrakce pravidel: algoritmy Frequent Itemset Mining (FISM) a Logical Itemset Mining (LISM). V další části navrhneme efektivní algoritmus na evaluaci vytvořených pravidel.

3.1 Extrakce pravidel

Naším prvním úkolem je vytvořit množinu pravidel, která označí zda jde o obyčejný provoz, nebo o útok. Tato pravidla lze psát ručně, jde ale o extrémně zdlouhavý proces, který je možné automatizovat. Navíc automatizované vytvoření pravidel má tu výhodu, že může nacházet i pravidla, která nejsou člověku na první pohled zřejmá. V následujících podsekcích si představíme dva algoritmy schopné generovat pravidla z dat, algoritmy FISM (následně i jeho alternativu FP-Growth) a LISM.

3.1.1 FISM

Prvním, kdo přišel s principem extrakce pravidel z dat, byl Petr Hájek a kolektiv článkem [20] z roku 1966. Rakesh Agrawal a jeho tým tyto principy znovuobjevili pro data-miningové účely v roce 1993 v článku „Mining Association Rules between Sets of Items in Large Databases“ [21]. Ve článku byl popsán Frequent Itemset Mining (FISM() algoritmus, který byl navržen pro obchodní účely, poskytoval totiž nástroje k efektivnímu zjišťování nákupního chování lidí. Do dnešního dne se navržený způsob extrakce pravidel stále používá, o čemž svědčí i vysoký počet citací článku, který se pohybuje kolem osmi tisíc.

3.1.1.1 Asociační pravidla

Na začátek si zaveďme entity, se kterými budeme pracovat. $\mathcal{I} = \{I_1, I_2, ..., I_d\}$ označíme množinu binárních atributů, které budeme nazývat *itemy*. T bude značit databázi transakcí, velikost této databáze si označíme jako n. Každá transakce t je repezentována jako binární vektor, kde $t[k] = 1$ v originálním článku značí fakt, že k-tý item byl vložen do košíku, $t[k] = 0$ značí opak. V našem případě bude obsah tohoto košíku popisovat chování uživatele. Jako X si označíme množinu itemů z $\mathcal{I}$, potom o transakci t říkáme, že *splňuje* X pokud pro všechny I_k v X platí $t[k] = 1$. Počet transakcí databáze T, které splňují X, označíme jako $N(X)$.

Implikaci $X \Rightarrow I_j$, kde $X \subseteq \mathcal{I}$ a $I_j \in \mathcal{I} \setminus X$ nazveme *asociačním pravidlem*. Část implikace X nazveme *původcem* (antecedent) a část I_j *následníkem* (consequent). Pravidlo $X \Rightarrow I_j$, kde $X \subseteq \mathcal{I}$ je splněno v databázi transakcí T s *konfidenčním faktorem* $0 \leq c \leq 1$ právě když alespoň $c\%$ transakcí t splňujících X zároveň splňuje i I_j. Tento fakt budeme značit $X \Rightarrow I_j \,|\, c$.

Jako příklad takovéhoto pravidla může posloužit následující nákup zákazníka v obchodě. Pravidlo $\{máslo, chléb\} \Rightarrow \{mléko\}$ říká, že pokud se v košíku objevuje máslo a chléb, potom zde bude s velikou pravděpodobností i mléko. Pokud alespoň 90% všech transakcí t obsahujících (tedy splňujících) máslo a chléb zároveň obsahuje i mléko, zapisujeme: $\{máslo, chléb\} \Rightarrow \{mléko\} \,|\, 0.9$.

Spolu s asociačními pravidly zavádíme:

- *support* pravidla $X \Rightarrow I_j$ vypočítáme jako $sup(X \Rightarrow I_j) = \frac{N(I_j)}{N(X)}$.
- *confidence* pravidla $X \Rightarrow I_j$ vypočítáme jako podmíněnou pravděpodobnost $conf = P(I_j|X)$.

Při vytváření pravidel typicky požadujeme splnění jistých omezujících podmínek:

- **Syntaktická omezení**, která omezují jak původce, tak následníky pravidla. Požadavkem tak například může být, že v původci X musí být obsažen nějaký I_k nebo, že pro I_j musí platit $I_j \in Y$, kde Y označuje nějakou předem definovanou množinu. V našem případě omezení bude spočívat v tom, že nás budou zajímat pouze ty pravidla, jejichž následníci jsou útoky.
- **Omezení na míru supportu**, tedy nastavení prahu sup_{min} takového, že nás budou zajímat pouze pravidla $X \Rightarrow I_j$, pro která platí $sup(X \Rightarrow I_j) > sup_{min}$.

Algoritmus, který extrahuje asociační pravidla musí být schopen brát tyto omezující podmínky v potaz.

3.1.1.2 Apriori algoritmus

Algoritmus, který navrhli autoři FISM [21], byl zpětně nazván apriori algoritmem. Ten vychází z jednoduché myšlenky: nejprve nageneruje všechny možné množiny a pak ověří jejich konfidenční faktor. Tyto dva kroky si v následujících odstavcích popíšeme podrobněji.

V první části nagenerujeme množinu $\mathcal{X}$ všech možných podmnožin itemů X takových, že jejichž četnost $\alpha(X)$ se nachází nad zvolenou hranicí θ_α. Četnost množiny je definovaná jako $\alpha(X) = \frac{N(X)}{n} \in \langle 0, 1 \rangle$. Tyto množiny jsou navíc generovány tak, aby splňovali sémantická omezení. Takže například pokud máme omezení takové, že předchůdce musí obsahovat item I_j budeme generovat pouze podmnožiny X takové, že pro ně platí $I_j \in X$.

V druhé části algoritmu použijeme nagenerované množiny $X \in \mathcal{X}$, kde X obsahuje itemy z $\mathcal{I}$. Generujeme pravidla typu $\bar{X} \Rightarrow I_j$, kde $\bar{X} \subseteq X \setminus I_j$ a platí, že $sup(\bar{X} \Rightarrow I_j) > sup_{min}$, tedy můžeme psát $\bar{X} \Rightarrow I_j \,|\, sup_{min}$. Samozřejmě generujeme pouze taková pravidla, která splňují syntaktická omezení.

Problém apriori algoritmu spočívá v jeho časové složitosti. V první části algoritmu musíme v nejhorším případě generovat $\mathcal{O}(2^d)$ množin, což není upočitatelné již při poměrně malé hodnotě d. Tento počet můžeme redukovat množinou omezení, ale i přesto je algoritmus velice neefektivní. Další nevýhoda spočívá v nutnosti opakovaně během algoritmu procházet vstupní data. Významná vylepšení původního algoritmu tedy na sebe nenechala dlouho čekat, za zmínku stojí:

- Eclat algoritmus [22],
- DCI a kDCI algoritmy [23],
- lcm [24]
- Borgeltovo vylepšení apriori algoritmu [25],
- FP-Growth [26].

Poslední zmíněný algoritmus si blíže popíšeme v následující sekci.

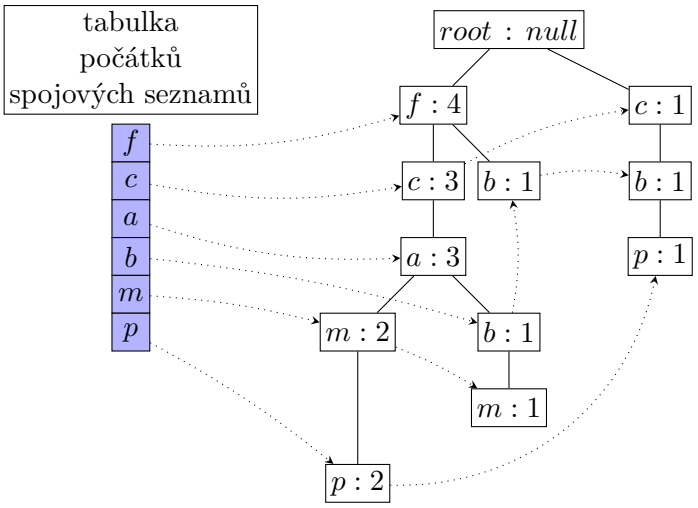
3.1.2 FP-Growth

Algoritmus FP-Growth byl představen týmem Jiawei Hana v roce 2000, viz článek [26]. Vylepšuje původní apriori algoritmus, především jeho tři největší slabiny:

- nutnost generovat obrovskou množinu kandidátů,
- nutnost několikrát iterovat přes data,
- paměťovou neefektivitu.

Tabulka 3.1: Databáze transakcí použitá k sestavení FP-stromu.

Transakce	**Množina itemů**	**Seřazená frekventovaná množina**
T_1	$\{f, a, c, d, g, i, m, p\}$	$\{f, c, a, m, p\}$
T_2	$\{a, b, c, f, l, m, o\}$	$\{f, c, a, b, m\}$
T_3	$\{b, f, h, j, o\}$	$\{f, b\}$
T_4	$\{b, c, k, s, p\}$	$\{c, b, p\}$
T_5	$\{a, f, c, e, l, p, m, n\}$	$\{f, c, a, m, p\}$



Obrázek 3.1: Dokončený FP-strom s tabulkou počátků node-linků.

Algoritmus již neextrahuje asociační pravidla, ale místo toho hledá tzv. frekventované množiny itemů. Jde o takové množiny X, pro které platí, že $N(X) > \theta_{minCount}$. $\theta_{minCount}$ označuje uživatelem zadanou hranici. Funkci algoritmu popíšeme na konkrétním příkladě z článku [26].

3.1.2.1 FP-strom

V první fázi běhu algoritmu se vytvoří tzv. FP-strom, tedy „Frequent-Pattern strom“. Ten komprimuje data sdílením shodných prefixů. Tento strom také představuje paměťově efektivní strukturu.

Mějme transakční databázi T obsahující 5 záznamů $T_1, ..., T_5$. Tyto záznamy obsahují podmnožiny itemů z množiny $\mathcal{I} = \{a, ..., p\}$. Transakce databáze můžeme vidět v tabulce 3.1. Hodnotu proměnné $\theta_{minCount}$ budeme v tomto případě uvažovat rovnu 3. V tuto chvíli se soustřeďme na sloupeček „Množina itemů“. V tomto sloupci napočítáme frekvence výskytu jednotlivých itemů. Dostáváme $\langle (f:4), (c:4), (a:3), (b:3), (m:3), (p:3) \rangle$. Z každé transakce vybereme do sloupečku „Seřazená frekventovaná množina“ všechny itemy I_j takové, že $N(I_j) > \theta_{minCount}$, a seřadíme je podle $N(I_j)$ sestupně.

V tuto chvíli začneme s konstrukcí FP-stromu. Začneme `root` uzlem, jehož hodnotu nastavíme na *null*. Potom dle sloupečku „Seřazená frekventovaná množina“ postupně sestavujeme prefixový strom. První transakci využijeme pro vytvoření větvě stromu, každému novému uzlu nastavíme jeho čítač na 1. Další transakce potom vytváří nové větvě pouze pro část transakce, která nesdílí prefix s žádnou předchozí transakcí. K čítači těch uzlů, které jsou součástí sdíleného prefixu se přičte jednička. Takto druhá transakce sdílí prefix $\langle f, c, a\rangle$, proto k čítačům těchto uzlů přičte jedničku, ale nové nevytváří. Vytvoří novou větev $\langle b, m\rangle$, která sdílená není, nastaví jim čítač na hodnotu 1 a napojí ji na uzel a. Obdobně se přistupuje i k dalším transakcím a vznikne takto strom, který můžeme vidět na obrázku 3.1. Na konec pro všechny itemy ve stromě vytvoříme tabulku počátků spojových seznamů spojující všechny itemy stejného jména napříč stromem.

Vzniklý strom má vlastnosti vhodné k pozdějšímu miningu frekventovaných množin. Obsahuje totiž všechny frekventované množiny itemů, jelikož každá množina obsahující frekventované itemy je zde zastoupena v nějaké cestě stromem. Další výhoda spočívá ve velké komprimaci těchto množin díky sdílení prefixů. Za poslední důležitou vlastnost lze považovat existenci spojových seznamů propojujících jednotlivé itemy stejného jména napříč stromem. Všech těchto vlastností vyžije FP-Growth algoritmus pro generování frekventovaných množin.

3.1.2.2 FP-Growth

Algoritmus FP-Growth využívá vytvořeného FP-stromu k extrakci frekventovaných množin itemů. Pojďme si nyní ukázat funkci tohoto algoritmu na FP-stromu, který jsme si zkonstruovali výše. Všechny frekventované množiny, které získáme, lze rozdělit do následujících skupin:

- množiny obsahující item p;
- množiny obsahující item m a neobsahující p;
- obdobně pokračujeme dále;
- množiny obsahující pouze item f.

Využijeme spojového seznamu a zjistíme všechny cesty od kořenového uzlu do itemu p. V našem případě jde o $\langle(f : 4), (c : 3), (a : 3), (m : 2), (p : 2)\rangle$ a $\langle(c : 1), (b : 1), (p : 1)\rangle$. První cesta naznačuje, že množina (f, c, a, m, p) se v databázi vyskytuje dvakrát. Tedy pro item p se v databázi T nachází prefix (f, c, a, m) dvakrát, což značíme jako $(fcam : 2)$. Obdobně pro druhou cestu stromem pro item p je zde prefix (c, b), který se v databázi vyskytuje jednou, značíme $(cb : 1)$. Tyto dvě prefixové cesty pro item p $\{(fcam : 2), (cb : 1)\}$ tvoří „sub-databázi“ pro item p, které nazýváme podmíněnou databází pro item p. Na těchto datech můžeme vystavět podmíněný FP-strom, jenž povede

Tabulka 3.2: Podmíněná data a odpovídající podmíněné FP-stromy pro jednotlivé itemy.

Item	Podmíněná databáze	Podmíněný FP-strom
p	$\{(fcam:2),(cb:1)\}$	$\{(c:3)\}\|p$
m	$\{(fca:2),(fcab:1)\}$	$\{(fca:3)\}\|m$
b	$\{(fca:1),(f:1),(c:1)\}$	$\emptyset$
a	$\{(fc:3)\}$	$\{(fc:3)\}\|a$
c	$\{(f:3)\}$	$\{(f:3)\}\|c$
f	$\emptyset$	$\emptyset$

k jediné větvi obsahující $\langle(c,3)\rangle$. Odvodíme z něho jedinou frekventovanou množinu $(cp,3)$. Tímto končí hledání množin pro item p.

Pro item m najdeme ve stromě 2 cesty: $\langle(f:4),(c:3),(a:3),(m:2)\rangle$ a $\langle(f:4),(c:3),(a:3),(b:1),(m:1)\rangle$. Všimněme si, že přestože p se s m v transakcích vyskytuje, již ho nemusíme brát v potaz, protože všechny jeho frekventované množiny již byly nalezeny v předchozím kroku. Obdobně jako v předchozím kroku si zjistíme podmíněnou databázi pro item m, která se v našem případě tvoří množinu $\{(fca:2),(fcb:1)\}$. Z těchto dat vytvoříme FP-strom a zjistíme, že obsahuje jednu větev $\langle(fca:3)\rangle$. Jelikož tento strom obsahuje pouze jednu větev nemusíme rekurzivně tvořit další podstromy, ale můžeme jednoduše vygenerovat všechny možnosti: $\{(f:3),(c:3),(a:3),(fc:3),(fa:3),(ca:3),(fca:3)\}$. Toto generování pomáhá zvyšovat efektivitu rekurzivního algoritmu. Obdobným způsobem zpracujeme podmíněnou databázi pro všechny zbývající itemy, vytvoříme jejich FP-strom a vyčteme frekventované množiny. V tabulce 3.2 lze nalézt podmíněnou databázi a odpovídající podmíněné FP-stromy pro jednotlivé itemy.

3.1.2.3 Využití FP-Growth algoritmu v naší práci

Pro experimenty v této práci použijeme pravidla, která byla vygenerována pomocí popsaného algoritmu FP-Growth a jsou používána ve frameworku CTA.

3.1.3 LISM

V roce 2012 vydal Shailesh Kumar ze společnosti Google spolu s dalšími autory článek „Logical Itemset Mining“ [27]. Tento článek popisuje způsob extrakce množin logicky souvisejících itemů v rámci zadaných dat. Článek, dle autorů, představuje efektivnější způsob extrakce pravidel než FISM (Frequent Itemset Mining), který jsme si představili v předchozí sekci. Jak již název napovídá, FISM hledá pouze často se vyskytující množiny, kdežto ty málo vyskytující se (přesto spolu logicky související) není schopen odhalit. Ty jsou ovšem často zajímavější.

3.1.3.1 Problém hledání logicky souvisejících množin

V této části budeme uvažovat databázi transakcí T sestávajících se z itemů ve shodě s předchozí podsekcí o FISM 3.1.1.1. Logicky související množinu článek definuje jako množinu itemů, která doplňuje zákazníkův záměr v obchodní oblasti, nebo sémantický koncept v oblasti zpracování textů nebo obrazů. Pokud bychom například uvažovali prodejnu a její zboží, následující produkty by mohly být označeny jako logicky související:

- kladivo, šroubovák, vrtačka, jde totiž o nástroje do dílny;
- klávesnice, myš, monitor, jde totiž o příslušenství k počítači.

V případě dat z frameworku CTA půjde o množinu eventů, které se vyskytují společně v popisu chování daného koncového zařízení za posledních 24 hodin.

Většinou se ale v transakcích všechny itemy množin nevyskytují společně kvůli následujícím vlastnostem dat:

- **promíchanost** zapřičiňující, že v nákupním košíků můžou být položky, které odpovídají více záměrům;
- **projekce**, jenž způsobuje, že v obyčejném košíku jsou jen některé itemy odpovídající danému záměru.

Navíc data jsou většinou zašuměná spoustou náhodných jevů, které mohou vést ke špatným výsledkům. LISM je navržený tak, že všechny tyto problémy dokáže vyřešit. Navíc je schopen extrahovat i množiny, které mají velice malý (nebo dokonce žádný) support.

LISM zpracovává databázi transakcí v následující posloupnosti:

1. **napočítávání**: v této fázi se mezi všemi dvojicemi itemů jednoprůchodově vypočítá, kolikrát se v transakcích vyskytují společně;
2. **konzistence**: počet výskytů se převede na konzistenční hodnotu, která vyjadřuje statistickou významnost výskytu daného páru v transakcích oproti náhodnému výskytu;
3. **odstranění šumu**: kde dojde k pročištění konzistenčních hodnot pro odstranění vlivu promíchanosti transakcí;
4. **objevování**: z konzistenčních hodnot vytvoříme vytvoříme graf, ve kterém hledáme kliky odpovídající jednotlivým logicky souvisejícím množinám.

Dále si tyto fáze popíšeme podrobněji.

3.1.3.2 Napočítání

V jednom průchodu databází transakcí $T = \{T_1, T_2, ..., T_n\}$ se napočítají tři statistiky. Prvně jde o počet společných výskytů pro všechny páry $(I_i, I_j) \in \mathcal{I} \times \mathcal{I}$, kde $\mathcal{I}$ odpovídá množině všech itemů:

$$\psi(I_i, I_j) = \sum_{k=1}^{n} \delta(I_i \in T_k)\delta(I_j \in T_k),$$

kde funkce $\delta(\alpha)$ nabývá hodnoty 1, pokud platí podmínka α, jinak nabývá hodnoty 0. Všechny společné výskyty menší než hranice θ_{cooc} jsou vynulovány. Tímto parametrem tak můžeme eliminovat šum v transakcích. Takto vznikne matice společných výskytů $\mathbb{M}_{cooc}$, která je řídká a symetrická.

Marginální počet $\psi(I_i)$ definujeme jako počet transakcí, ve kterých se item I_i nachází v páru spolu s jiným itemem. Tento počet můžeme vypočítat jako:

$$\psi(I_i) = \sum_{I_j \in \mathcal{I}, I_i \neq I_j} \psi(I_i, I_j).$$

Celkový součet všech párů si označíme jako ψ_0 a vypočítáme ho:

$$\psi_0 = \frac{1}{2} \sum_{I_i \in \mathcal{I}} \psi(I_i).$$

Předchozí tři počty můžeme použít pro výpočet pravděpodobnosti společného výskytu páru $P(I_i, I_j)$ a marginální pravděpodobnosti $P(I_i)$, které vypočítáme jako:

$$P(I_i, I_j) = \frac{\psi(I_i, I_j)}{\psi_0}, P(I_i) = \frac{\psi(I_i)}{\psi_0}.$$

3.1.3.3 Konzistence

Tato fáze má za úkol odstranit vliv často se vyskytujícího šumu, ale zachovat málo se vyskytující validní množiny. Proto LISM nepočítá se sdruženou pravděpodobností jako FISM, ale místo toho počítá *konzistenci společného výskytu*, která se definuje jako míra podobnosti distribuce společných výskytů a náhodné veličiny. Pro tento výpočet lze použít několik podobnostních funkcí jako je Jaccardova, Point-Wise Mutual Information, normalizovaná Point-Wise Mutual Information nebo kosinová podobnost. Pro jednoduchost uvádíme vzorec pouze poslední jmenované, která se vypočítá jako

$$\phi_{cos}(I_i, I_j) = \frac{P(I_i, I_j)}{\sqrt{P(I_i)P(I_j)}} \in \langle 0, 1 \rangle .$$

Všechny konzistence společného výskytu otestujeme, zda nejsou menší než práh θ_{consy}. Pokud ano, dané hodnoty vynulujeme. Parametr θ_{cooc} řadíme mezi další parametry, kterými lze eliminovat míru šumu v datech. Danými výpočty vznikne nová *matice konzistencí společného výskytu*, kterou si označíme $\mathbb{M}_{consy}$.

3.1.3.4 Odstranění šumu

Některé logicky nesouvisející páry, které mají vysoké množství výskytu pouze kvůli promíchanosti dat, již byly odstraněny převodem počtů společných výskytů na konzistence společných výskytů. Nejsou však takto redukovány všechny šumy v datech, proto LISM zavádí další fázi, a to iterativní odstranění šumů v matici konzistencí.

Funkce $\psi^{(t)}(I_i, I_j)$ a $\phi^{(t)}(I_i, I_j)$ označíme jako počet společných výskytů a konzistenci společných výskytů t-té iterace algoritmu. Potom jednotlivé kroky algoritmu popíšeme jako $\forall(I_i, I_j) \in \mathcal{I} \times \mathcal{I} : \psi^{(0)}(I_i, I_j) > \theta_{cooc}$,

$$\psi^{(t+1)}(I_i, I_j) \leftarrow \psi^{(0)}(I_i, I_j)\delta(\phi^{(t)}(I_i, I_j) > \theta_{consy}).$$

Po každém kroku iterace jsou přepočítány marginální a celkové počty. Dále se přepočítá matice pravděpodobností a marginální pravděpodobnosti. Všimněme si, že během těchto iterací využíváme pouze data z již napočítaných matic a není potřeba další průchod databází, který by výpočty značně zpomaloval.

Postupně můžeme sledovat, jak se zvyšuje kvalita matice konzistencí společného výskytu při t-té iteraci $Q(\mathbb{M}^{(t)}_{consy})$, kterou můžeme vypočítat jako:

$$Q(\mathbb{M}^{(t)}_{consy}) = \sum_{(I_i,I_j)\in V\times V} P^{(t)}(I_i, I_j)\phi^{(t)}(I_i, I_j).$$

3.1.3.5 Objevování

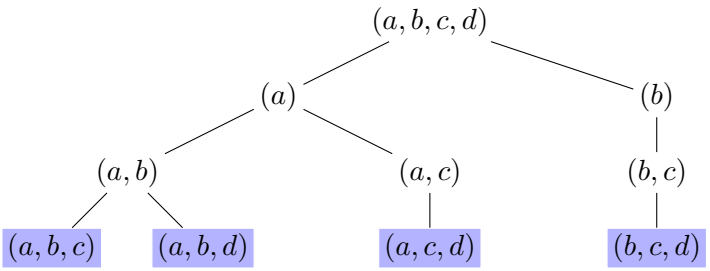
V předchozích fázích postupně došlo k odstranění šumu v datech a k posílení konzistencí společného výskytu mezi itemy, které jsou součástí logicky souvisejících množin. Matici konzistence společných výskytů $\mathbb{M}_{consy}$ nyní použijeme ke konstrukci grafu G. Neorientovaný graf si nadefinujeme jako dvojici $G = (\boldsymbol{V}, \boldsymbol{E})$ kde:

- $\boldsymbol{V}$ odpovídá množině vrcholů a obsahuje všechny itemy z $\mathcal{I}$;
- $\boldsymbol{E}$ odpovídá množině hran obsahující všechny hrany $\{I_i, I_j\}$ takové, že $I_i, I_j \in \boldsymbol{V}$ a zároveň $\psi(I_i, I_j) > \theta_{consy}$.

V grafu G nyní potřebujeme najít množiny, které jsou vzájemně propojené, tedy tvoří kliku. Můžeme tedy využít algoritmy pro hledání maximálních klik grafu. Každá nalezená klika odpovídá jedné množině logicky souvisejících itemů. Problém hledání maximálních klik řadíme mezi NP-úplné problémy a časová složitost řešení tohoto problému se rovná $\mathcal{O}(3^{d/3})$, kde $d = |\mathcal{I}|$, viz [28].

3.1.3.6 Použití LISM

LISM lze s úspěchem aplikovat na textových datech, což autoři článku demonstrují velice dobrými výsledky při nacházení souvisejících popisků obrázků



Obrázek 3.2: Demonstrace principu algoritmus expandujícího kliku o velikosti 4 na kliky o velikosti 3.

z databáze Flicker. My použijeme LISM pro generování pravidel, jejichž úspěšnost v odhalení útoků budeme testovat na reálných datech.

V našem případě budou hledanými logicky souvisejícími množinami skupiny event, které se v datech často vyskytují spolu, tedy popisují nějaké chování koncových zařízení dohromady. Pokud LISM nalezne v datech logicky související množiny itemů, jejichž součástí je i eventa označující útok (label), budeme moci tyto množiny využít jako pravidla pro odhalování těchto útoků. Navíc ostatní itemy množiny budou tvořit kontext danému útoku.

LISM byl navržen primárně pro maloobchodní účely a pro práci s textovými a obrazovými daty. Otestujeme tedy, zda dokáže efektivně extrahovat pravidla popisující síťové útoky.

3.1.3.7 Expanze pravidel

Při implementaci v jazyce `Python` jsme pro sestavení grafu použili knihovny `networkx`. Použitá funkce knihovny

```
networkx.cliques_containing_node(graph, nodes=labels)
```

vrací množinu všech nalezených maximálních klik pro všechny jednotlivé labely (útoky). Tyto kliky odpovídají logicky souvisejícím množinám, a proto by teoreticky samy o sobě mohly sloužit jako pravidla. Jak ovšem ukázalo měření je vhodné rozdělit tyto maximální kliky na menší kliky. Data totiž průměrně obsahují 5,88 event na řádek, a proto dlouhá pravidla spoustu řádků jistě za útok neoznačí. Platí při tom, že s rostoucí délkou pravidla roste precision a klesá recall a opačně. Proto bude délka expandovaných klik dalším parametrem, který u LISM algoritmu budeme optimalizovat.

Pro expandování kliky na menší kliky jsme použili algoritmus demonstrovaný na obrázku 3.2. Původní kliku o velikosti l expandujeme na všechny kliky o velikost m. Jde o algoritmus 1. V prvním kroku začneme rekurzi pro všech prvních $l-m+1$ event pravidla. V dalších krocích rekurze vždy expandujeme daný stav tak, že přidáváme postupně další eventy, které jsou na řade do již

existující kliky. Zbytečně neexpandujeme ty stavy, které jistě nevedou k validním klikám (nebudou dostatečně veliké). Rekurzi ukončíme ve chvíli, kdy již klika dosáhla požadované velikost. Celkem takto vygenerujeme

$$\binom{l}{n} = \frac{l!}{(l-n)!n!}$$

různých klik velikosti n.

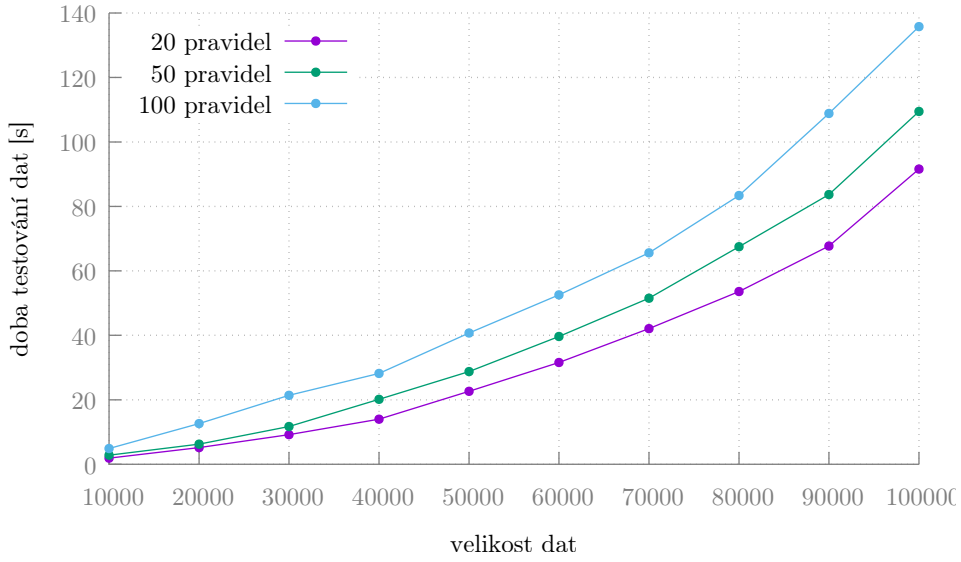
Algoritmus 1: Algoritmus expandující větší kliku na menší.

```
   Data: origClq, newSize
   Result: Množina všech expandovaných klik požadované velikosti
 1 newClqs ← ∅ ;                                  // Inicializace výstupního pole
 2 forall i ∈ ⟨, origClq.size − newSize + 1) do
       // Začni rekurzi pro eventu
 3     expandClqRec(origClq[i], i + 1, newSize, newClqs, origClq)
 4 end
 5 return newClqs    function expandClqRec(tmpClq, next, newSize,
     newClqs, origClq)
 6     if tmpClq.size == newSize then
           // Pokud je vytvořena klika požadované velikosti, ulož ji a konči
 7         newClqs ← newClqs ∪ tmpClq
 8         return
 9     end
10     forall i ∈ ⟨next, origClq.size − newSize + tmpClq.size + 1) do
           // Expandování
11         expandClqRec(tmpClq ∪ origClq[i], i + 1, newSize, newClqs)
12     end
```

3.2 Efektivní evaluace pravidel

Již máme vytvořená pravidla popisující útoky, nyní potřebujeme efektivní způsob, jak pravidla na datech vyhodnocovat. Podívejme se nejprve na naivní řešení problému, viz algoritmus 2. Problém tohoto algoritmu spočívá v jeho časové složitosti. Pokud mají data N_{data} řádků a maximálně m_{data} eventů na řádek a mají pravidla N_{rules} řádků a maximálně m_{rules} eventů na řádek, potom se celková časová složitost tohoto algoritmu rovná $\mathcal{O}(N_{data} \times N_{rules} \times min\{m_{data}, m_{rules}\})$, což není pro reálné použití dostačující.

Závislost doby testování na velikosti vstupních dat a na počtu pravidel můžeme vidět na grafu 3.3. Můžeme zde vidět, že s počtem dat roste čas kvadraticky. S počtem pravidel roste doba provádění lineárně, s koeficientem menším než jedna.

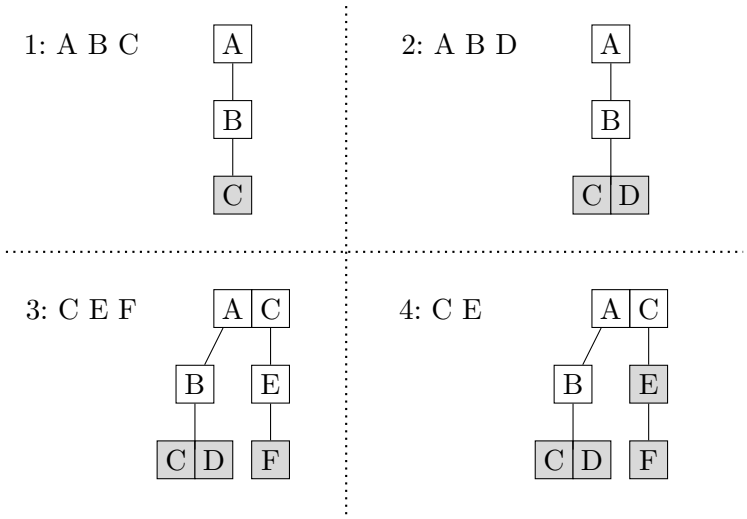


Obrázek 3.3: Závislost doby testování na velikosti vstupních dat a na počtu pravidel naivního algoritmu.

Algoritmus 2: Naivní algoritmus pro evaluaci pravidel na datech.

Data: *rules*, *data*
Result: Pole predikcí dat dle zadaných pravidel

```
1  prediction ← [Ø] ;                          // Inicializace výstupního pole
2  forall row ∈ data do
3      row ← sort(row)
4      forall rule ∈ rules do
5          rule ← sort(rule)
6          forall i ∈ ⟨0, len(rule)⟩ do
7              if i == len(rule) then
                   // Pravidlo splněno, označ útok
8                  prediction.append(true)
9                  break
10             else if i == len(row) or row[i] ≠ rule[i] then
11                 continue ;          // Pravidlo nesplněno, pokračuj na další
12             end
13         end
14     end
       // Žádné pravidlo nesplněno, nejde o útok
15     prediction.append(false)
16 end
17 return prediction
```



Obrázek 3.4: Ukázka postupné stavby prefixového stromu. Šedé buňky odpovídají terminálním uzlům.

Algoritmus 3: Sestavení prefixového stromu z pravidel.

Data: *rules*
Result: Prefixový strom reprezentující všechna pravidla

```
1  root ← Node() ;                                  // Inicializace kořenového uzlu
2  forall rule ∈ rules do
3      i ← 0
4      tmpRoot ← root
5      lastNodeItem ← ∅
6      while i < len(rule) do
7          actualItem ← rule[i] ;                   // Načti si novou položku pravidla
8          if actualItem ∉ tmpRoot.items then
               // Pokud v aktuálním uzlu neexistuje aktuální položka
                  pravidla, tak ji přidej
9              newNodeItem ← NodeItem(actualItem)
10             newNodeItem.children ← Node()
11             tmpRoot.items ← tmpRoot.items ∪ newNodeItem
12         end
           // Pokračuj hlouběji do stromu
13         tmpRoot ← tmpRoot.items[actualItem].children
14         lastNodeItem ← newNodeItem i ← i + 1
15     end
16     lastNodeItem.isTerminal = true ;  // poslední označ za terminální
17 end
```

Pro daný problém jsme schopni navrhnout efektivnější algoritmus. Algoritmus bude využívat principu n-árního prefixového stromu. Všechna pravidla seřadíme abecedně. Jednotlivé uzly stromu (v pseudoalgoritmu nazvané $Node()$) obsahují seznam všech elementů ($node.items$ obsahující $NodeItem()$ prvky), které se nacházely alespoň v jednom pravidlu tak, že cesta z kořenového uzlu do aktuálního uzlu tvořila prefix danému prvku. Každý $NodeItem()$ si drží odkaz na svého potomka ($nodeItem.children$) a příznak, zda se v některém pravidlu nacházel jako poslední ($nodeItem.isTerminal$).

Prefixový strom sestavíme z pravidel dle algoritmu 3. Názorný obrázek postupné konstrukce prefixového stromu můžeme vidět na obrázku 3.4. Takto vytvořený strom bude mít hloubku maximálně r_2. To odpovídá maximálnímu počtu kroků, který pro každý řádek potřebujeme, abychom mohli rozhodnout, zda jde o útok či nikoliv.

Časovou složitost vyjádříme jako $\mathcal{O}(N_{data} \times m_{rules} \times N_{features})$, kde $N_{features}$ odpovídá počtu features. Pro n_{data} řádků dat musíme projít stromem hloubky m_{rules}, přičemž v každém zanoření hledáme v seznamu feature, který může být maximálně $N_{features}$ dlouhý (ale v praxi bývá mnohem kratší). Pravidla mají řádově tisíce řádků, proto námi navržený algoritmus je řádově tisíckrát rychlejší než naivní řešení.

Při testování potom používáme algoritmus 4. Na začátku řádek dat abecedně seřadíme. Potom postupně iterujeme před položky řádku a zanořujeme se hlouběji do stromu. Pokud další zanoření není možné (v daném uzlu není přítomen prvek, který je právě na řadě), ukončíme testování a vracíme $false$, stejně jako když skončíme v netermálním uzlu. Naopak $true$ vracíme ve chvíli, kdy řádek úspěšně prošel stromem a dostal se do terminálního uzlu.

Navržená metoda je schopná sestrojit strom z jakýchkoliv pravidel, u kterých lze dílčí prvky pravidel seřadit. Bez možnosti seřazení totiž nelze mluvit o prefixech. Strom jistě budeme schopni sestavit z pravidel, která máme vygenerovaná algoritmy LISM a FISM.

3.3 Implementace prefixového stromu

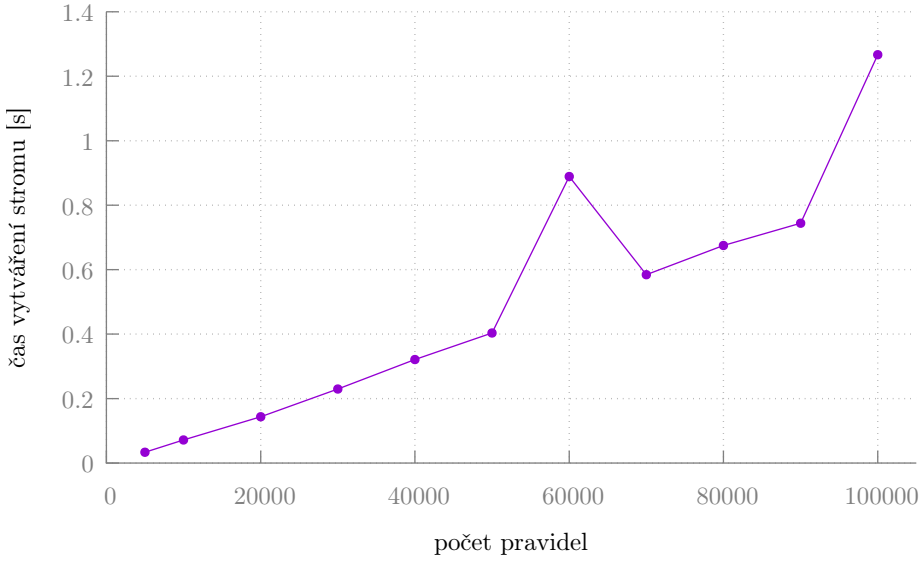
Prefixový strom jsme implementovali v programovacím jazyce `Python` ve verzi 3.5. Podrobili jsme ho zátěžovým testům a sledovali jsme dobu, za kterou je algoritmus schopen vytvořit strom v závislosti na počtu pravidel. Dále jsme sledovali dobu trvá testování dat stromem v závislosti na počtu pravidel a na velikosti testovaných dat. Testování probíhalo na souborech obsahujících unikátní pravidla (popis jejich generování v sekci 3.1.3.7), ve kterých se vyskytuje celkem 325 features. Byla použita reálná data z frameworku CTA.

Na grafu 3.5 můžeme vidět závislost doby vytvoření prefixového stromu na počtu pravidel. Vidíme, že doba roste přibližně lineárně s rostoucím počtem pravidel.

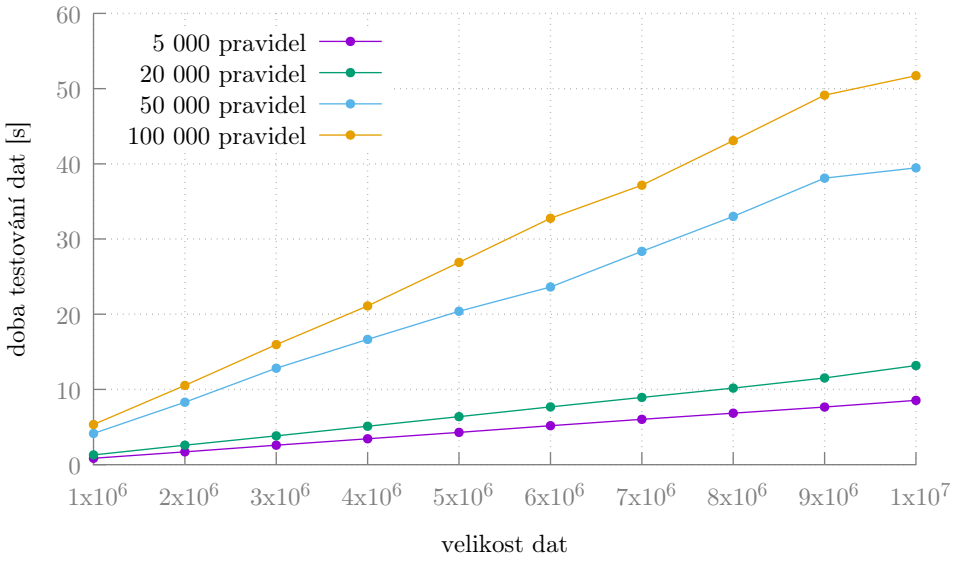
Algoritmus 4: Klasifikace dat pomocí prefixového stromu pravidel.

Data: $prefixTreeRoot, data$
Result: Pole predikcí dat dle zadaných pravidel

```
1  prediction ← [Ø] ;                          // Inicializace výstupního pole
2  forall row ∈ data do
3      tmpRoot ← prefixTreeRoot ;               // Každý řádek začíná v kořenu
4      forall item ∈ row do
5          if item ∉ tmpRoot.items then
               // Žádné pravidlo nesplněno, nejde o útok
6              prediction.append(false)
7          else if tmpRoot.items[item].isTerminal then
               // Pravidlo splněno, označ útok
8              prediction.append(true)
9          else
10             tmpRoot = tmpRoot.items[item].children
11         end
12     end
13 end
14 return prediction
```



Obrázek 3.5: Závislost doby vytvoření prefixového stromu na počtu pravidel.



Obrázek 3.6: Závislost doby testování prefixovým stromem na velikosti vstupních dat a na počtu pravidel.

Závislost doby testování prefixovým stromem na velikosti vstupních dat a na počtu pravidel můžeme vidět na grafu 3.6. Prvním z výsledků je: doba testování s rostoucí velikostí dat roste lineárně. Dále stojí za zmínění, že testování trvá velice krátkou dobu. Otestování 10 milionů dat 100 000 pravidly trvá 51,7 sekundy. Naivní algoritmus potřeboval odpovídající čas pro vyhodnocení 100 pravidel na 60 000 datech (tedy vyhodnocení tisíckrát méně pravidel na 160krát méně datech). Také si můžeme všimnout, že doba testování závisí na množství pravidel, ale tato závislost je „lepší“ než lineární. V našem případě otestování 10 milionů dat stromem vytvořeným z 5 000 pravidel trvalo 8,5 sekundy. Oproti tomu otestování stejného množství dat 100 000 pravidly trvalo 51,7 sekundy, tedy pouze 6krát déle. To proto, že mnoho pravidel nevytváří vlastní položky v uzlech, ale jsou sdíleny napříč pravidly.

Testováním evaluace pravidel prefixovým stromem jsme potvrdili, že jde o velice efektivní způsob vyhodnocování dat. Pro úplnost musíme dodat, že měření by na jiných datech a jiných pravidlech mohla dopadnout jinak, protože prefixový strom je datově citlivý a různá data mohou generovat různé stromy s odlišnými vlastnostmi. Vždy ale budeme použitím prefixového stromu dostávat řádově lepší výsledky než použitím naivního řešení.

Kapitola 4

Optimalizace a srovnáni klasifikátorů

V této kapitole nejprve optimalizujeme parametry všech klasifikátorů, se kterými budeme později experimentovat. Následně provedeme srovnání všech klasifikátorů na základě jejich vlastností na měřených datech. Všechny experimenty proběhnou cross-validací. Celková množina dat se rozdělí na `N` částí, jedna část se vždy použije jako testovací a ostatní části jako trénovací. Testovací část postupně rotuje přes všech `N` částí. Nakonec spočítáme průměr z jednotlivých pokusů. V našem případě budeme data rozdělovat na 5 částí.

Optimalizaci, stejně jako testováním, implementujeme v programovacím jazyce `Python` ve verzi 3.5. Jednotlivé klasifikátory jsou součástí námi používané knihovny `scikit-learn` ve verzi 0.19.1, která obsahuje veliké množství nástrojů strojového učení, pro více informací viz [29]. Optimalizace parametrů proběhne na cloudovém virtuálním stroji s následující konfigurací:

- 8 jader,
- 16 GB RAM paměti,
- 64 bitový operační systém Ubuntu 14.04.

Jednotlivé klasifikátory budeme porovnávat z pohledu precision, recall a doby naučení a otestování (v součtu).

4.1 Popis a optimalizace klasifikátorů

V první části experimentů vybereme vhodné parametry jednotlivým klasifikátorům. Problém hledání nejlepších parametrů klasifikátoru lze převést na prohledávání nekonečného stavového prostoru. Jako stav tohoto prostoru označíme kombinace jednotlivých parametrů klasifikátoru. Jednotlivé stavy porovnáváme podle jejich precision, recall a rychlosti na společné podmnožině dat.

Stavový prostor je nekonečný proto, že parametry mohou nabývat reálných čísel.

Jelikož nejsme schopni prohledat celý tento stavový prostor, zavádíme pro hledání parametrů následující heuristickou funkci:

- na začátku necháme všechny parametry nastavené na výchozí hodnoty dle nastavení knihovny;
- postupně iterujeme přes námi zvolené parametry, které zkoušíme nastavit na omezený počet námi zvolených hodnot;
- každou jednotlivou konfiguraci parametrů otestujeme na speciálně vybraných datech (viz níže) pomocí křížové validace;
- v každém běhu budeme sledovat precision, recall a dobu trvání křížové validace klasifikátorů, na základě kterých se rozhodneme pro nejlepší hodnotu parametru;
- jakmile zjistíme hodnotu jednoho parametru, ponecháme tento parametr nastaven na danou hodnotu a přejdeme k testování dalšího parametru.

Čas, který zaznamenáváme u všech měření, odpovídá době trvání jednoho natrénování a otestování klasifikátoru na dále zmíněných datech. Tuto dobu vypočítáme jako průměrnou dobu jednoho foldu (testování na jedné `N`tině dat) cross-validace.

4.1.1 Data

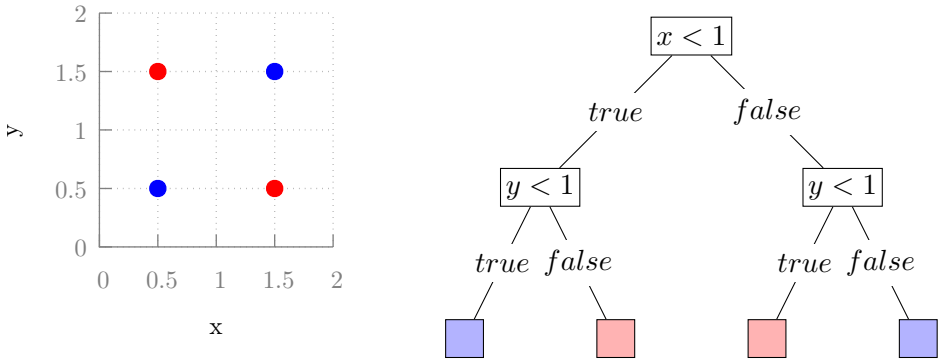
Při testování parametrů klasifikátorů budeme potřebovat provádět mnoho testů, proto nemůžeme pro experimenty použít celou množinou dat. Vybereme náhodně 200 tisíc řádků, které odpovídají asi 1% celku, takže jde o dobrý reprezentační vzorek. Data jsou vybrána následujícím způsobem:

- 180 000 řádků neolabelovaných jako útok;
- 20 000 řádků odpovídající útoku.

Originální data obsahují asi 0,5% řádků popisujících infikovaný provoz. Zvýšením hustoty pozitivních nálezů budeme lépe schopni porovnávat jednotlivé klasifikátory mezi sebou na více různých typech infekce.

4.1.2 Náhodné lesy

Prvním klasifikátorem, jehož parametry budeme nastavovat, je klasifikátor využívající náhodné lesy. V knihovně jde o `RandomForestClassifier`. Náhodné lesy využívají množiny rozhodovacích stromů, které se vytvoří v trénovací fázi. Každý vnitřní vrchol stromu obsahuje podmínku na vlastnost elementu, po



Obrázek 4.1: Ukázka řešení klasifikace problému XOR pomocí rozhodovacího stromu.

jejíž splnění pokračujeme do jednoho podstromu tohoto vrcholu, v případě nesplnění pokračujeme do druhého podstromu. Listy stromu potom obsahují informaci, do které třídy zařadit prvky, které (ne)splnili podmínky na cestě od tohoto listu do kořenového uzlu. Ukázku rozhodovacího stromu můžeme vidět na obrázku 4.1, můžeme vidět strom řešící XOR klasifikační problém. Při testování všechny stromy klasifikují element a většinové rozhodnutí určí výslednou třídu elementu.

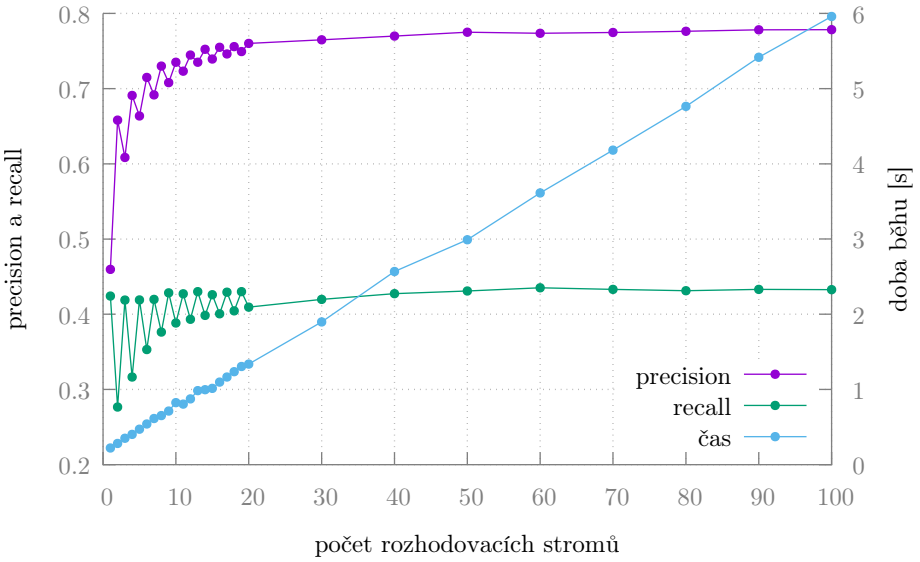
U tohoto klasifikátoru budeme experimentovat s následujícími parametry:

- počet stromů lesa (`n_estimators`; výchozí hodnota 10);
- maximální počet vlastností, podle kterých se bude v jednotlivých uzlech rozhodovat (`max_features`; výchozí hodnota `auto`);
- maximální hloubka, které strom může dosáhnout (`max_depth`; výchozí hodnota `None`).

Prvním měřeným parametrem bude počet rozhodovacích stromů. Závislost času, precision a recallu na hodnotě parametru můžeme vidět na grafu 4.2. Vidíme, že zatímco doba běhu přibližně lineárně roste, precision a recall se ustaluje přibližně u počtu 19 stromů, potom se už významněji nemění. Dále také vidíme, že při lichém počtu stromů roste recall a mírně klesá precision, protože nikdy nedojde k remíze v hlasování. Na základě měření volíme 19 stromů.

Dalším parametrem je parametr maximální počet features, které se berou v potaz při hledání vhodných podmínek do vnitřních uzlů. Tento parametr můžeme nastavit na:

- fixní hodnotu n takovou, že $n \in (0, N_{tot}\rangle$, kde N_{tot} odpovídá celkovému počtu features;
- zlomek čísla N_{tot};



Obrázek 4.2: Závislost precision, recall a doby provádění klasifikátoru `RandomForestClassifier` na počtu použitých rozhodovacích stromů.

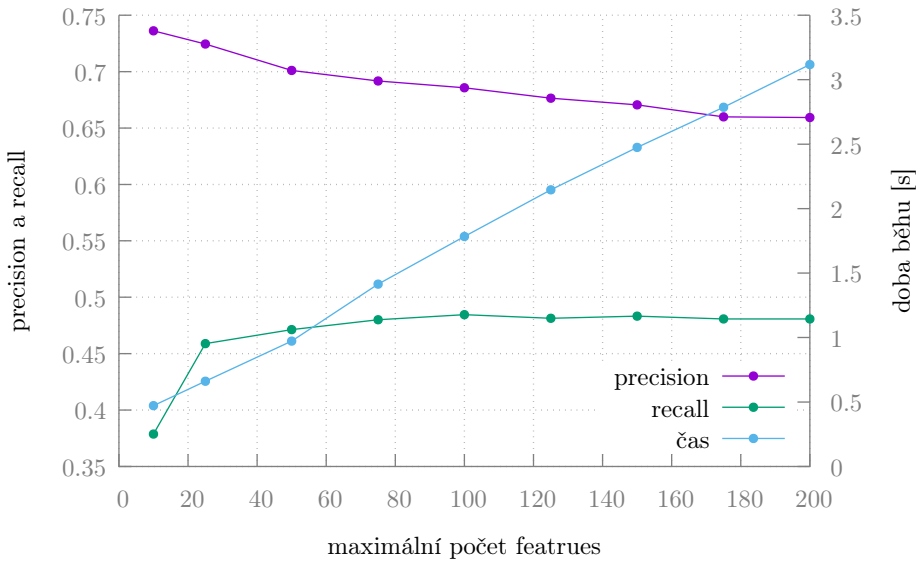
- nebo na jednu z následujících hodnot: `sqrt`($=\sqrt{N_{tot}}$), `log2`($=\log_2 N_{tot}$), `None`($=N_{tot}$).

Graf 4.3 zobrazující výsledky nám ukazuje, že s rostoucím počtem uvažovaných features roste recall a klesá precision. Nakonec volíme kompromis a budeme operovat se 100 features.

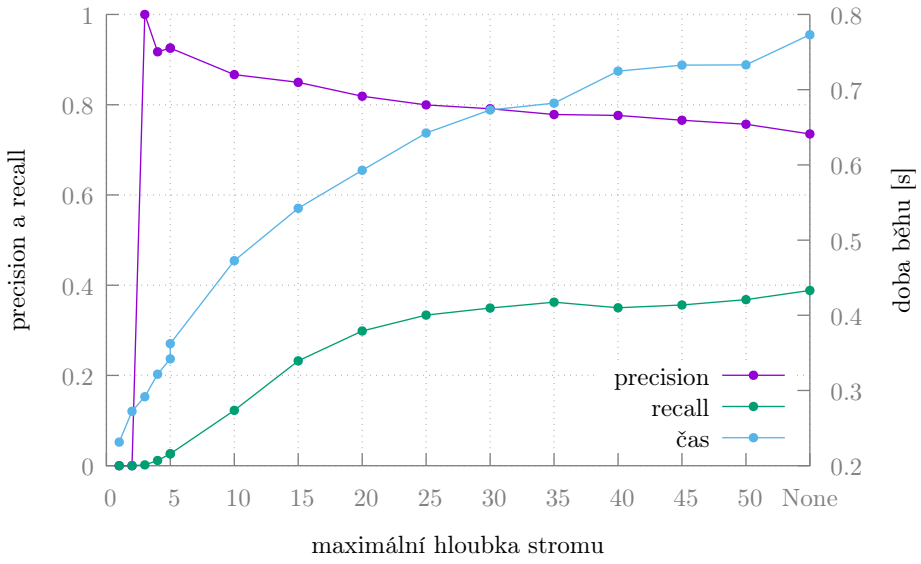
Posledním řešeným parametrem u tohoto klasifikátoru bude maximální hloubka stromu, který se v lese může vyskytovat. Tu je obecně vhodné omezit, aby nedošlo k přeučení klasifikátorů. Parametr můžeme nastavit na libovolné celé číslo nebo hloubku neomezovat. Výstupy experimentu můžeme vidět na grafu 4.4. Z grafu vyplývá, že po prvotním ustálení precision konstantně klesá a recall nejprve rychle roste a kolem hloubky 30 se téměř ustaluje. Precision ale považujeme za důležitější, volíme tedy hloubku 25.

Na základě předchozích experimentů tedy budeme používat následující klasifikátor s parametry:

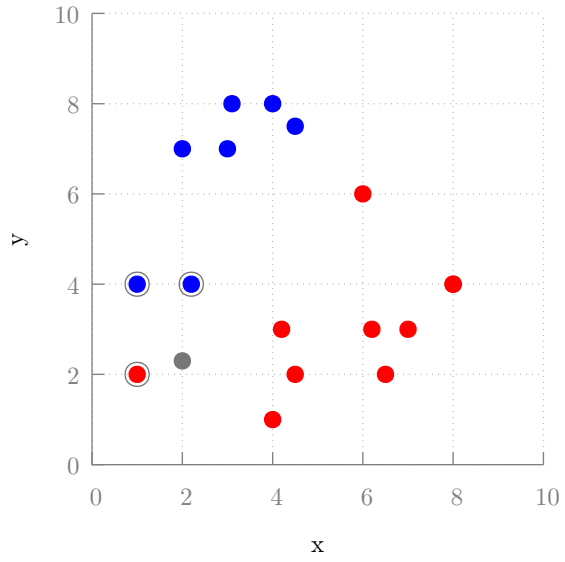
```
RandomForestClassifier(
        n_estimators = 19,
        max_features = 100,
        max_depth = 25
).
```



Obrázek 4.3: Závislost precision, recall a doby provádění klasifikátoru `RandomForestClassifier` na počtu uvažovaných features.



Obrázek 4.4: Závislost precision, recall a doby provádění klasifikátoru `RandomForestClassifier` na maximální hloubce stromů.



Obrázek 4.5: Klasifikace bodu klasifikátorem K-nejbližších sousedů, kde `K = 3`. Klasifikace šedého bodu bude probíhat tak, že se naleznou 3 nejbližší sousedé (zakroužkované body). Z těch se vybere majoritní třída, v tomto případě bude bod zařazen do modré třídy.

4.1.3 K-nejbližších sousedů

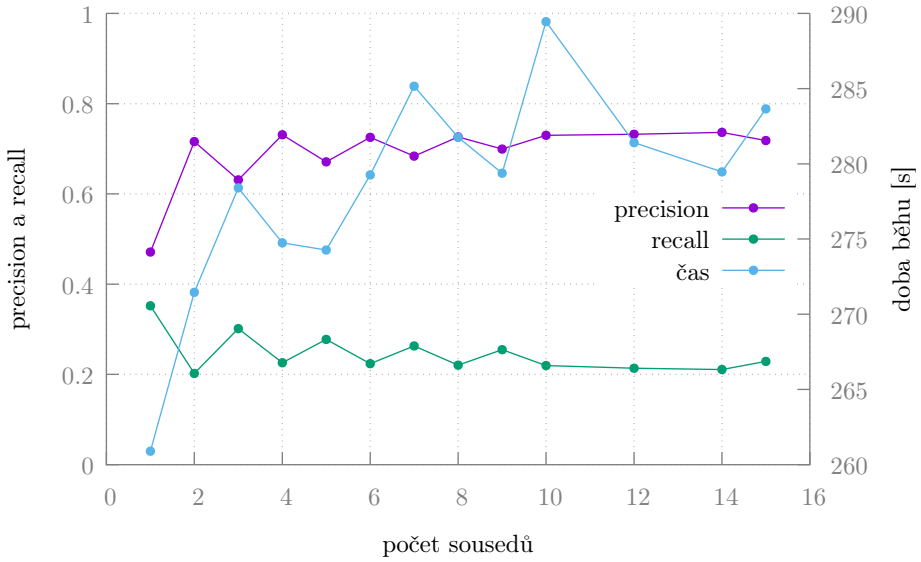
Klasifikátor `KneighborsClassifier` si ve fázi vytvoří množinu olabelovaných bodů v `n`-dimenzionálním prostoru, kde `n` je počet features jednotlivých elementů. V testovací fázi potom pro každý nový bod najde množinu `N` nejbližších bodů. Pro nalezenou množinu zjistí, do jaké třídy většina bodů náleží. Do této třídy zařadí i testovaný bod. Ukázku fungování klasifikátoru můžeme najít na obrázku 4.5.

Problémem tohoto klasifikátoru je jeho neefektivita. Testování trvá velice dlouho, a proto již dopředu víme, že tento klasifikátor nebude tím nejlepším. Proto jediný parametr, který budeme sledovat, bude počet sousedů, který klasifikátor bere v potaz při klasifikaci.

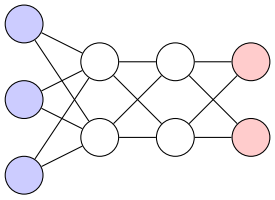
Závislost precision, recallu a doby běhu můžeme viděl na grafu 4.6. Na grafu vidíme, že precision významněji rostla asi do počtu 10 sousedů, potom zůstala přibližně konstantní. Recall postupně klesala se zvyšujícím se počtem sousedů. Také můžeme vidět zvětšování precision a zmenšování recallu při sudém počtu sousedů. Důvod bude obdobný, jako u náhodných lesů. Volíme kompromis mezi dostatečnou precision a recallem počtem sousedů 9.

Používat tedy budeme tento klasifikátor s parametry:

```
KneighborsClassifier(n_estimators = 19).
```



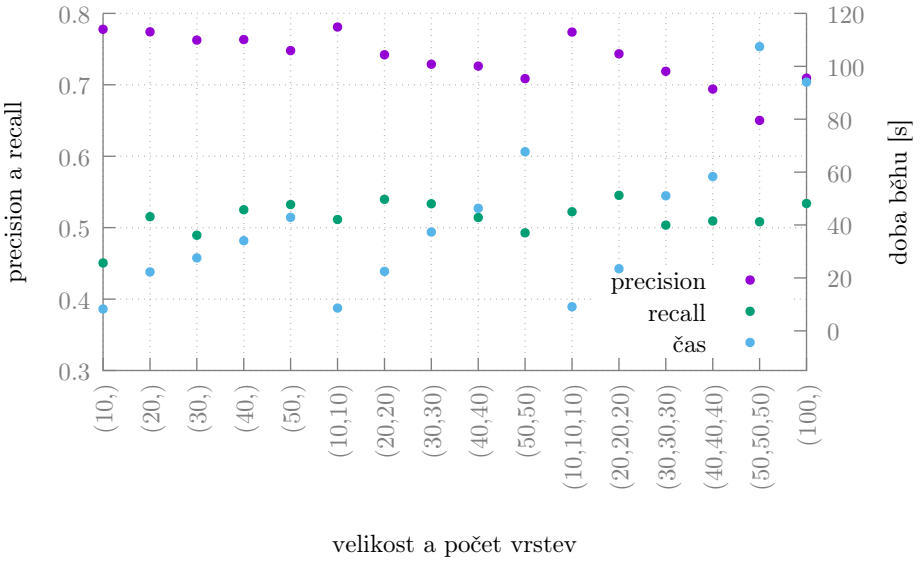
Obrázek 4.6: Závislost precision, recall a doby provádění klasifikátoru `KneighborsClassifier` na počtu uvažovaných sousedů.



Obrázek 4.7: Příklad konfigurace neuronové sítě se dvěma skrytými vrstvami vždy o velikosti 2 neurony odpovídající konfiguraci (2,2) v dalším testování. Modře vyznačujeme vstupní vrstvu a červeně tu výstupní.

4.1.4 Neuronové sítě

Klasifikátor `MLPClassifier` použijeme jako zástupce neuronových sítí. Neuronové sítě fungují na principu propojení neuronů podobným způsobem, jako jsou propojeny biologické neurony v mozku. Síť se sestává ze vstupní vrstvy, na kterou přivádíme data. Dále síť obsahuje několik různě velikých skrytých vrstev. Poslední je výstupní vrstva obsahující neurony odpovídající jednotlivým klasifikačním třídám. Na obrázku 4.7 můžeme vidět příklad konfigurace neuronové sítě se dvěma skrytými vrstvami vždy o velikosti 2 neurony, odpovídající konfiguraci (2,2) v dalším testování. Neurony můžou být různě nabuzené a propojení mezi nimi (tzv. synapse) mohou mít různou váhu. Ve fázi trénování si síť nastaví jednotlivé váhy synapsí tak, aby síť co nejlépe dokázala klasifikovat data. Ve fázi testování potom sledujeme, který z neuronů výstupní sítě



Obrázek 4.8: Závislost precision, recall a doby provádění klasifikátoru využívající neuronové sítě na konfiguraci sítě. Čísla v závorce znamenají počet neuronů ve skryté vrstvě, čísla odělená čárkou znamenají počty v jednotlivých vrstvách. Popisek (10,10) tedy odpovídá síti, která má kromě vstupní a výstupní vrstvy také dvě skryté, obě o velikosti 10 neuronů.

se nejvíce nabudil po přivedení vstupních dat.

U MLP klasifikátoru budeme optimalizovat tyto parametry:

- velikost a počet skrytých vrstev sítě (`hidden_layer_sizes`; výchozí hodnota `(100,)`, tedy 1 skrytá vrstva o 100 neuronech);
- aktivační funkci neuronu (`activation`, výchozí hodnota `adam`);
- optimalizátor vah synapsí (`solver`, výchozí hodnota `relu`).

Parametr `hidden_layer_sizes` udává počet a velikost vnitřních vrstev sítě. Zkoušeli jsme nastavoval jednu až tři skryté vrstvy vždy s deseti až padesáti vnitřními neurony. Navíc jsme porovnali měření s referenčním nastavením knihovny, které využívá jednu skrytou vrstvu se 100 neurony. Nejlépe vyšla kombinace tří skrytých vrstev vždy s deseti neurony. Ta má lepší precision než výchozí a téměř stejnou recall. Je ale 16krát rychlejší, proto volíme právě tuto konfiguraci sítě.

Parametr `solver` určuje algoritmus, který bude zvolen pro optimalizace vah synapsí. Testovali jsme celkem 3 algoritmy a to:

- `lbfgs`, optimalizátor využívající Kvazi-Newtonovu metoda, viz [30];
- `sgd`, odpovídá metodě stochastického gradientního sestupu, viz [31];

Tabulka 4.1: Závislost precision, recall a doby provádění neuronové sítě na optimalizačním algoritmu.

Algoritmus	**Precision**	**Recall**	**Doba běhu** [s]
lbfgs	0.761	0.499	23.863
sgd	0.747	0.491	15.956
adam	0.793	0.488	7.099

Tabulka 4.2: Závislost precision, recall a doby provádění neuronové sítě na aktivační funkci neuronu.

Aktivační funkce	**Precision**	**Recall**	**Doba běhu** [s]
identity	0.786	0.399	6.244
logistic	0.767	0.536	156.302
tanh	0.704	0.542	154.521
relu	0.709	0.534	94.295

- `adam`, což označuje optimalizátor založený na stochastickém gradientu, viz [32].

V tomto případě uděláme výjimku oproti ostatním testováním. Optimalizace velice závisí na konfiguraci, kterou síť má. Proto testujeme na zvolené konfiguraci $(10, 10, 10)$. Výstup z experimentu můžeme najít v tabulce 4.1. Dle měření optimalizátor s největší precision je algoritmus `adam`, na druhou stranu má ale nejnižší recall. Řešitel `adam` se ale navíc, dle dokumentace `sklearn` [33], používá pro veliké datasety, proto volíme právě tento algoritmus.

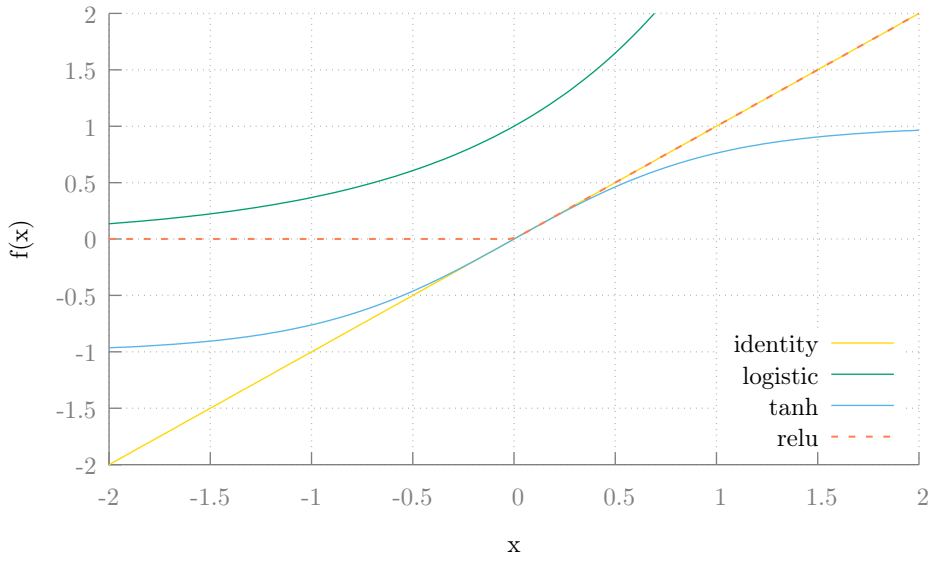
Parametr `activation` volí aktivační funkci neuronu, tedy závislost míry nabuzení neuronu na vstupu. Ukázky průběhů všech aktivačních funkcí, které `sklearn` nabízí, můžeme vidět na grafu 4.9. Jde o funkce:

- `identity`, funkce $f(x) = x$;
- `logistic`, funkce $f(x) = 1/(1 + e^{-x})$;
- `tanh`, funkce $f(x) = tanh(x)$;
- `relu`, funkce $f(x) = max\{0, x\}$.

Výsledek testování můžeme najít v tabulce 4.2. Dle testování vyšla nejlépe aktivační funkce `logistic`, doba běhu programu byla nejdelší, ale má nejlepší precision a recall.

Tento klasifikátor tedy budeme používat s následujícími parametry:

```
MLPClassifier(
        hidden_layer_sizes = (10,10,10),
        solver = "adam",
```



Obrázek 4.9: Ukázky průběhů aktivačních funkcí neuronu.

```
        activation = "logistic"
).
```

4.1.5 Support Vector Classifier

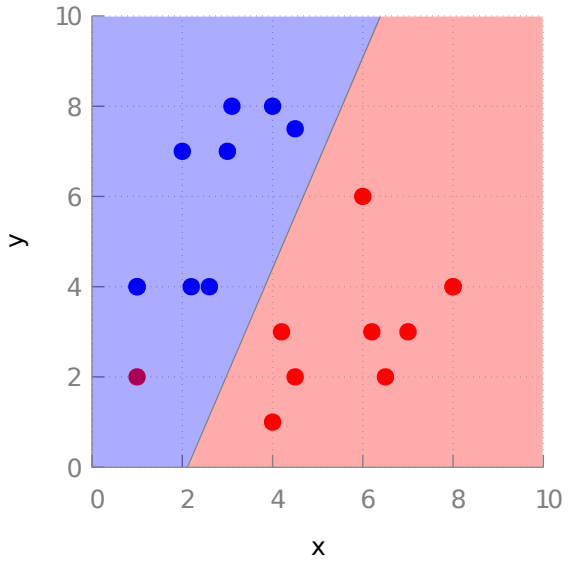
Zkratka SVC (Support Vector Classifier) představuje klasifikátor založený na binárním rozdělení prostoru nadrovinou tak, že od sebe oddělíme data spadající do odlišných tříd. Nejdůležitějším parametrem tohoto klasifikátoru je kernel, který určuje, jak bude vypadat nadrovina rozdělující prostor. Příklad rozdělení prostoru na základě dat můžeme vidět na obrázku 4.10.

U klasifikátoru budeme optimalizvat následující parametry:

- kernel (`kernel`; výchozí hodnota `(100,)`, tedy 1 skrytá vrstva o 100 neuronech);
- zda mají být povoleny pravděpodobnostní odhady (`probability`, výchozí hodnota `False`).

Jako první budeme hledat vhodný kernel. Knihovna nabízí následující možnosti:

- `linear`, který rozdělí prostor lineární funkcí;
- `poly`, který rozdělí prostor polynomiální funkcí;
- `rbf`, který použije funkci „Radial basis“;



Obrázek 4.10: Ukázka klasifikace klasifikátorem SVC. Vidíme, že klasifikátor rozdělil prostor na 2 podroviny. Jakýkoliv nový bod bude klasifikován dle jeho pozice v tomto prostoru.

Tabulka 4.3: Závislost precision, recall a doby provádění SVC na použitém kernelu.

Kernel	**Precision**	**Recall**	**Doba běhu [s]**
linear	0.755	0.325	513.548
poly	0.000	0.000	192.768
rbf	0.744	0.217	475.164
sigmoid	0.772	0.190	233.769

- `sigmoid`, který použije sigmoidní funkci.

Tabulce 4.3 obsahující výsledky měření nám ukazuje, že na datech má `linear` nejlepší poměr mezi precision a recallem. Proto volíme pro parametr `kernel` hodnotu `linear`.

Další parametr, který budeme testovat je `probability`, parametr určující, zda mají být povoleny pravděpodobnostní odhady, více v dokumentaci knihovny [34]. Výstup z experimentu lze nalézt v tabulce 4.4. Testováním jsme ale přišli na to, že pokud tuto funkci nepoužijeme, dostaneme stejné výsledky 4krát rychleji, proto `probability` nastavujeme na `False`.

Při experimentech tedy budeme používat následující klasifikátor:

```
SVC(
        kernel = "linear",
```

Tabulka 4.4: Závislost precision, recall a doby provádění SVC na použití pravděpodobnostních odhadů.

Odhad použit	Precision	Recall	Doba běhu [s]
Ano	0.744	0.217	1485.643
Ne	0.744	0.217	464.071

```
        probability = False,
).
```

4.1.6 Naivní Bayes

Dalším klasifikátorem, který budeme v rámci této sekce optimalizovat, bude naivní bayesovský klasifikátor. Klasifikátor staví svoje predikce na Bayesově teorému, tedy na vzorci:

$$P(A\ |B) = \frac{P(B|A)P(A)}{P(A)}.$$

Během trénovací fáze klasifikátor zjišťuje podmíněné pravděpodobnosti. Při testování jsou data olabelovaná tou třídou, jejíž pravděpodobnost podmíněná vstupními daty je nejvyšší. Klasifikátor předpokládá, že všechny vlastnosti dat jsou vzájemně nezávislé, proto se nazývá naivní. Předpokládá se tedy, že platí

$$P(X_1, X_2, ...X_N|C) = P(X_1|C)P(X_2|C)\cdots P(X_N|C),$$

kde data X_i jsou jednotlivé features a C označuje třídu.

Knihovna nám dává k dispozici 3 klasifikátory:

- `GaussianNB`, předpokládající normální rozdělení dat;
- `MultinomialNB`, předpokládající multinomické rozdělení dat;
- `BernoulliNB`, předpokládající Bernoulliho rozdělení dat.

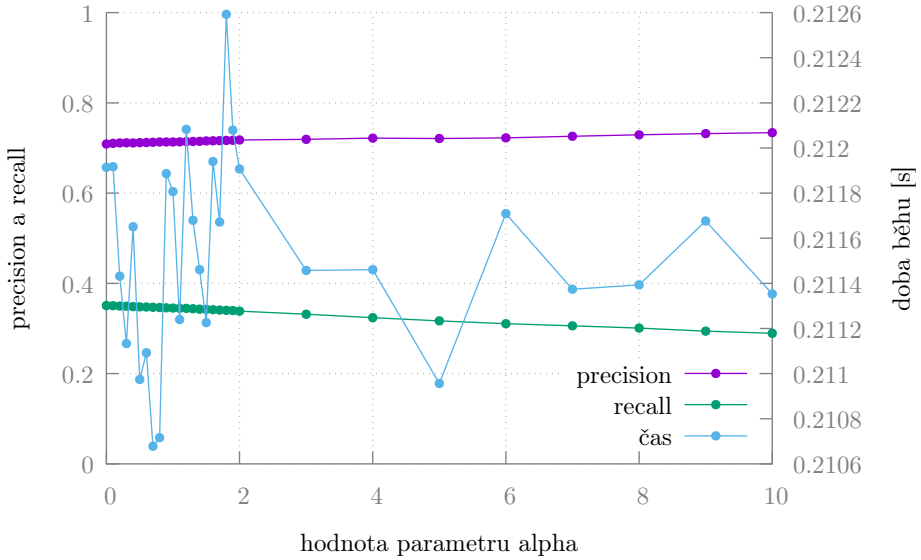
Nejprve vybereme vhodný model a dále pro něho budeme hledat vhodné parametry. Co se týče doby běhu, tak byly všechny modely přibližně stejně rychlé, jak můžeme vidět na tabulce 4.5. Nejvyšší precision měl `MultinomialNB`, nejvyšší recall měl `BernoulliNB`. Nakonec jsme zvolili klasifikátor `MultinomialNB`.

U klasifikátoru `MultinomialNB` budeme dále optimalizovat jeho parametry:

- hodnotu vyhlazovacího parametru (`alpha`; výchozí hodnota `1.0`);
- parametr udávající, zda se mají učit tzv. prior pravděpodobnosti (`fit_prior`; výchozí hodnota `True`).

Tabulka 4.5: Závislost precision, recall a doby provádění Naivního Bayesu na modelu dat.

Model dat	Precision	Recall	Doba běhu [s]
Gaussian	0.406	0.500	0.372
Multinomial	0.713	0.345	0.212
Bernoulli	0.431	0.512	0.252



Obrázek 4.11: Závislost precision, recall a doby provádění klasifikátoru `MultinomialNB` na parametru `alpha`.

Parametr `alpha` udává dle dokumentace hodnotu adaptivního (Laplaceho/ Lidstoneova) vyhlazovacího parametru, více například v [35]. Výstup z experimentu můžeme vidět na grafu 4.11. S rostoucí hodnotou parametru pomalu roste precision a klesá recall, doba provádění kolísá. Volíme kompromisní hodnotu 1,5. Posledním parametrem tohoto klasifikátoru je parametr `fit_prior`, udávající, zda se má klasifikátor učit tzv. prior pravděpodobnosti, více například v [36]. Ukázalo se, že použití tohoto principu zvyšuje precision na úkor recallu (viz tabulka 4.6). Dáváme přednost precision a volíme hodnotu tohoto parametru `True`. Rozdíly v době běhu programu byly minimální.

Tento klasifikátor tedy budeme používat jako:

```
MultinomialNB(
        alpha = 1.5,
        fit_prior = True,
).
```

Tabulka 4.6: Závislost precision, recall a doby provádění Naivního Bayesu na použití `fit_prior` parametru.

`fit_prior` použit	Precision	Recall	Doba běhu [s]
Ano	0.713	0.345	0.212
Ne	0.326	0.748	0.211

4.1.7 LISM pravidla

Klasifikátorem založeným na vyhodnocování LISM pravidel končíme část optimalizací parametrů. Tato pravidla budeme generovat pomocí FP-Growth algoritmu, jehož myšlenky a implementaci jsme si představili v podsekci 3.1.3. Optimalizace tohoto klasifikátoru bude spočívat v optimalizaci parametrů algoritmu LISM, jenž generuje pravidla. Každý vygenerovaný set pravidel poté budeme vyhodnocovat pomocí prefixového stromu, který byl představen v sekci 3.2.

Pro velikou efektivitu LISM a prefixového stromu si můžeme dovolit optimalizovat parametry na celé množině dat. Proto si ji rozdělíme na 8 milionů dat, která vyžijeme pro extrakci pravidel, a 2 miliony dat, která použijeme pro vyhodnocení kvality extrahovaných pravidel.

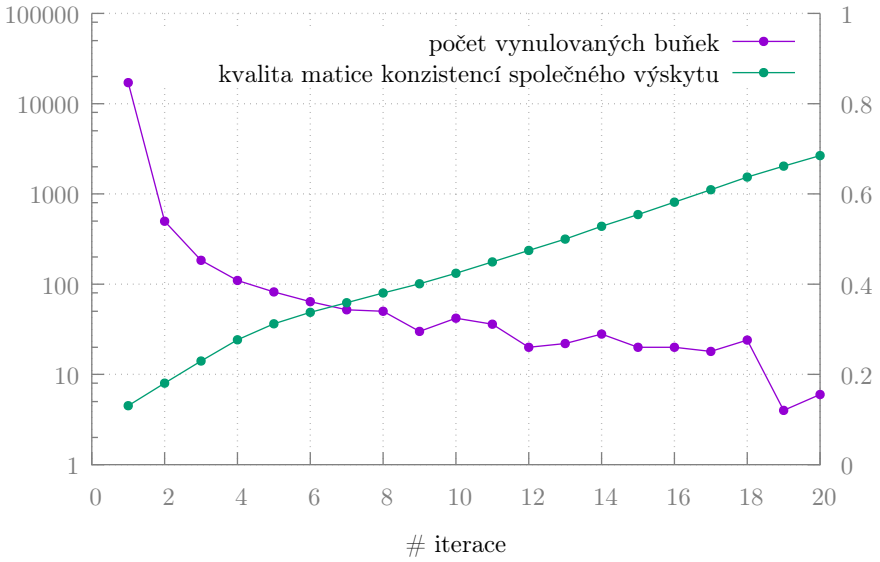
Algoritmus LISM má velké množství optimalizovatelných parametrů k optimalizaci:

- práh pro počet společných výskytů θ_{cooc};
- práh pro konzistenci společných výskytů θ_{consy};
- funkce podobnosti $\phi(\alpha, \beta)$;
- počet iterací funkce odstranění šumu;
- maximální velikost generovaných pravidel.

Parametry spolu vzájemně souvisí, a proto není možná oddělená optimalizace, jako u ostatních parametrů v této sekci. Díky rychlosti extrahování pravidel a vyhodnocování můžeme zvolit grid-search strategii. Zvolíme omezenou množinu vhodných hodnot pro jednotlivé parametry a poté postupně zkoušíme všechny kombinace těchto hodnot.

Během testování jsme zjistili následující poznatky:

- Pro naše data se jako nejlepší ukázala funkce podobnosti ϕ_{cos}, tedy kosinová. Několik pravidel bylo vygenerováno i s použitím Jaccardovy podobnosti, neměla ale tak dobré výsledky.
- Počet iterací funkce odstranění šumu není příliš podstatný, jak můžeme vidět na obrázku 4.12 (všimněme si, že osa počtu vynulovaných elementů je logaritmická). Již v prvních 2 iteracích se ve velkém procentu případů



Obrázek 4.12: Příklad rychlosti konvergence funkce odstranění šumu LISM. Osa počtu vynulovaných elementů je logaritmická.

odstraní 95% všech výskytů šumu a další iterace již velký smysl nemají. Tímto potvrzujeme vlastnost algoritmu, kterou popsali autoři článku. Postupně se zvyšuje kvalita matice, ta ale příliš veliký vliv na kvalitu generovaných pravidel nemá.

- 3-7 eventů na pravidlo se ukázalo jako ideální počet. Menší pravidla mají extrémně malou precision, větší zase mají malý recall (řádově 0,02%).
- Měření ukázala, že parametr θ_{consy}, tedy práh pro konzistenci společných výskytů, má největší vliv na kvalitu generovaných pravidel. Hodnota parametru větší než 0.0005 způsobí to, že žádná pravidla nevzniknou, protože jsou vymazány všechny důležité informace o konzistencích společných výskytů. Naopak při hodnotě menší než 10^{-7} už vzniká spousta pravidel, která způsobují veliký počet FP nálezů, a tím pádem významně klesá precision.
- Naopak význam parametru θ_{cooc} se ukázal jako poměrně marginální. Významněji neovlivňoval kvalitu generovaných pravidel.

Právě parametr θ_{consy} odhaluje problém LISM: nutnost vzniku kliky grafu pro vznik pravidla. Aby byla klika sestavena, musí vzniknout hrana i mezi features, které sice souvisí s útoky, ale nemusí nutně souviset sami se sebou. Proto je vhodné nastavit nižší hodnoty θ_{cooc} a θ_{consy} a vhodným postprocesingem odstranit nadbytečná pravidla. Za účelem zvýšení confidence pravidel jsme podrobili vygenerovaná pravidla následující selekci. Množinu pravidel

otestujeme řádek po řádku. Pro každý jednotlivý řádek sestavíme jeho prefixový strom a otestujeme na něm náhodně vybraný milion dat. Vygenerujeme 2 soubory, které budou obsahovat pouze ty řádky pravidel, které mají míru confidence nad 50% (respektive nad 80%) a které označili více než 2 řádky dat.

LISM budeme trénovat na 6,5 milionu řádcích dat. Jde o zbytek z celkového množství 11,5 milionu po odečtení 5 milionů řádků na celkové porovnání klasifikátorů.

Na základě experimentů a zjištění jsme zvolili pravidla vygenerována při následujících hodnotách parametrů:

- $\theta_{cooc} = \{10^{-4}, 7.5 \cdot 10^{-5}, 5 \cdot 10^{-5}, 2.5 \cdot 10^{-5}, 10^{-5}, 7.5 \cdot 10^{-6}, 5 \cdot 10^{-6}, 2.5 \cdot 10^{-6}, 10^{-6}\}$;
- $\theta_{consy} = \{1, 2, 3\}$;
- funkci podobnosti použijeme ϕ_{cos}, tedy kosinovou podobnost a ϕ_{jacc}, tedy Jaccardovu;
- 5 iterací funkce odstranění šumu;
- maximální velikost generovaných pravidel z množiny $3, 4, 5, 6, 7, 8, 9, 10$.

Tyto parametry použijeme pro natrénování pravidel na 6,5 milionech dat. Výše popsanou selekcí byly z těchto dat vybrány 2 sety pravidel, které pro testování označíme jako $\text{LISM}_{0.5}$ (pravidla s confidence větší než 50%) a $\text{LISM}_{0.8}$ (pravidla s confidence větší než 80%).

4.2 Srovnání klasifikátorů

V předchozí sekci jsme optimalizovali parametry jednotlivých klasifikátorů tak, aby podávaly dobré výsledky na našich datech. V této sekci provedeme celkové srovnání všech klasifikátorů na větší množině dat.

Jelikož potřebujeme větší výpočetní výkon, než byl potřeba při optimalizaci parametrů, budeme měřit tyto data na cloudovém virtuálním stroji s následující konfigurací:

- 32 jader,
- 64 GB RAM paměti,
- 64 bitový operační systém Ubuntu 16.04.

Zároveň všechny další experimenty v této práci budou prováděny na tomto virtuálním stroji.

Klasifikátory budeme porovnávat na dvou podmnožinách dat podle jejich efektivity a schopnosti v rozumném čase vytvořit model a otestovat data:

Tabulka 4.7: Závislost precision, recall a časů natrénování a otestování na testovaném klasifikátoru. Časy s hvězdičkou odpovídají pouze době sestavení prefixového stromu. Buňky označené dvěma hvězdičkami jsme nemohli vyplnit, protože nemáme k dispozici data z odpovídající konfigurace.

Klasifikátor	**Velikost dat**	**Prec**	**Rec**	t_{train}[s]	t_{test}[s]
Náhodný les	5 000 000	0,778	0,964	4630,988	25,886
Neuronová síť	5 000 000	0,982	0,928	4176,270	18,358
Naivní Bayes	5 000 000	0,577	0,262	27,035	11,333
kNN	500 000	0,823	0,976	5025,848	61487,518
SVC	500 000	0,820	0,748	55748,353	8261,518
$LISM_{0.5}$	5 000 000	0,617	0,366	0,010*	5,762
$LISM_{0.8}$	5 000 000	0,855	0,080	0,001*	5,707
$FISM_{0.5}$	5 000 000	0,689	0,014	**	**
$FISM_{0.8}$	5 000 000	0,839	0,027	**	**

- Klasifikátory založené na pravidlech, neuronových sítích, náhodných lesích a Naivní Bayes budeme testovat na datech obsahující 5 milionů náhodných řádků.

- Klasifikátory k-nejbližších sousedů a SVC budeme testovat na datech obsahujících 500 000 náhodných řádků zdrojových dat. Jelikož tyto klasifikátory nejsou efektivní, 5 milionů řádků dat nebyly tyto klasifikátory schopné zvládnout pod 100 hodin.

Budeme sledovat stejné parametry, jako jsme sledovali u optimalizace parametrů. Rozvedeme si ale čas, nyní si ho rozdělíme na dobu potřebnou k vytvoření modelu a dobu testovaní. Doba testování hraje totiž významnější roli, model si můžeme natrénovat dopředu, ale testování musí být rychlé, aby zvládalo real-time vyhodnocování dat. Všechny klasifikátory, až na ty založené na pravidlech, budeme testovat cross-validací, stejně jako v předchozích experimentech.

4.2.1 Vyhodnocení

Výše zmíněné klasifikátory jsme otestovali na datech, změřili jejich precision, recall a časy natrénování modelů a následné otestování dat. Výstupy všech testů jsou k nalezení v tabulce 4.7. V této podsekci si blíže popíšeme výsledky i vlastnosti jednotlivých klasifikátorů.

Náhodné lesy měly největší recall ze všech měřených tradičních klasifikátorů a to 96,4%. Měl také poměrně dobrý čas testování dat v porovnání s klasifikátory kNN a SVC. Precision bychom v kontextu ostatních výsledků označili za průměrnou, odpovídala 77,8%.

Neuronové sítě si v experimentu vedly velice dobře. Měly největší precision (98,2%) a navíc také velice dobrý recall (92,8%). Stejně tak i doba testování dat byla poměrně nízká.

Naivní Bayes se ukázal jako nejrychlejší klasifikátor, nepočítaje klasifikátory založené na pravidlech z pohledu testování. Má velice dobrý čas i trénovací fáze a to 27 sekund. Bohužel jeho precision (57,7%) a recall (26,2%) v porovnání s ostatními klasifikátory jsou podprůměrné.

Časová neefektivita K-nejbližších sousedů způsobuje, že není možné použít tento klasifikátor na větších datech. Cross-validace na 500 000 řádcích trvala v našem testovacím prostředí celkem 92 hodin. I v našem případě jsme museli využít menší množinu dat, abychom dostali alespoň nějaké výsledky v rozumném čase. Na druhou stranu se ale tento klasifikátor ukázal jako poměrně efektivní, jeho precision byla 82% a recall měl nejvyšší ze všech: 97,6%.

Stejně jako u předchozího klasifikátoru neefektivita SVC způsobuje nevhodnost použití klasifikátoru na větších datech. Dalším problémem SVC je schopnost pouze binární klasifikace a jeho rozšíření na multi-class není jasně definováno. Jak doba trénování, tak testování je vysoká i na poměrně malých datech. Celková doba běhu cross-validace na zmíněných datech trvala celkem 89 hodin. Přitom precision (82%) a recall (74,8%) tohoto klasifikátoru jsou průměrné.

Klasifikátory založené na evaluaci pravidel LISM a FISM nevynikají z pohledu precision ani recall. Jejich výstupy jsou:

- pravidla $\text{LISM}_{0.5}$ s confidence větší než 0,5 měla precision 61,7% a recall 36,6%;
- pravidla $\text{LISM}_{0.8}$ s confidence větší než 0,8 měla precision 85,5% a recall 8%;
- pravidla $\text{FISM}_{0.5}$ s confidence větší než 0,5 měla precision 68,9% a recall 1,4%;
- pravidla $\text{FISM}_{0.8}$ s confidence větší než 0,8 měla precision 83,9% a recall 2,7%;

Čím tyto klasifikátory vynikají je časová efektivita. Čas testování je dvakrát kratší než nejrychlejší klasifikátor, který není založen na pravidlech a to klasifikátor Naivní Bayes.

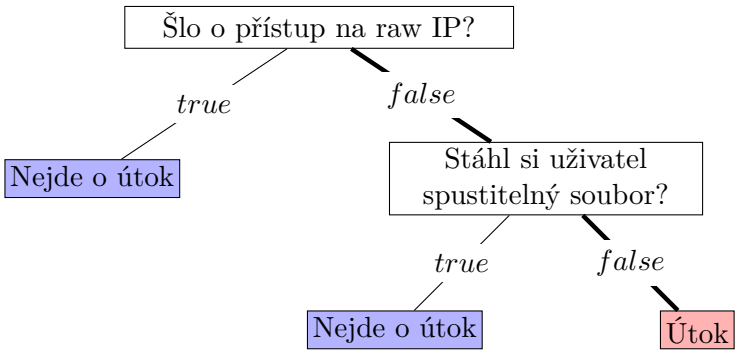
Kapitola 5

Srovnání kontextu

V předchozí kapitole jsme si porovnali všechny klasifikátory z pohledu jejich precision a recall. V této kapitole nás bude zajímat jiná kvalita algoritmů a to jejich schopnost poskytovat kontext incidentů. Postupně si projdeme všechny klasifikátory a u každého si popíšeme, zda je schopen poskytovat kontext a v čem tento kontext spočívá.

5.1 Náhodné lesy

Z pohledu extrahovaného kontextu jsou rozhodovací stromy schopny podat omezenou míru kontextu. Tento kontext totiž tvoří cesta od listu (odpovídajícího klasifikační třídě) do kořene stromu, jak můžeme vidět na obrázku 5.1. Problémem ovšem je, že tato cesta obsahuje jak splněné, tak i nesplněné podmínky rozhodovacích uzlů. Náš kontext představují pouze splněné podmínky, protože zákazníka v naprosté většině případů zajímá to, co se na síti děje, ne to, co se tam neděje.



Obrázek 5.1: Ukázka problému extrakce kontextu rozhodovacích stromů. Tučnou čarou je zobrazena cesta od listu reprezentujícího útok do kořene.

Problém extrakce kontextu náhodných lesů ilustruje rozhodovací strom 5.1. Na základě dat byl sestaven tento jednoduchý rozhodovací strom odhalující útoky. Dejme tomu, že strom označí nějaká data za útok. Následně poskytne kontext: tento provoz byl označen za útok, protože nešlo o přístup na raw IP a zároveň si uživatel nestáhl žádný spustitelný soubor. Tedy přidaná hodnota ke klasifikaci je nulová.

5.2 Neuronové sítě

Z pohledu extrakce kontextu jsou neuronové sítě nevhodné. Jedinou informaci, kterou z neuronové sítě získáme, je nabuzení vnitřních neuronů a nastavení vah synapsí. V jistých případech lze z těchto informací kontext extrahovat. Obecně jde ale o velice složitou úlohu, jejíž řešení je mimo rozsah této práce.

5.3 Naivní Bayes

Naivní Bayes je dalším z klasifikátorů schopných poskytnout omezený kontext útoku. Víme které features přispěli k rozhodnutí, že jde o útok. Navíc více i mírou s jakou k tomto rozhodnutí přispěly. To proto, že si klasifikátor ukládá pro každou feature podmíněnou pravděpodobnost, že souvisí (respektive nesouvisí) s útokem. Features s největší podmíněnou pravděpodobností potom můžeme zákazníkovi prezentovat jako kontext útoku, dokonce je můžeme seřadit podle významnosti. Naivní Bayes má ale stejný problém jako náhodné lesy. Podává jak pozitivní, tak negativní kontext, který nás většinou nezajímá.

5.4 K-nejbližších sousedů

K-nejbližších sousedů není příliš vhodný k extrakci kontextu. Jediný poskytnutý kontext může obsahovat informace o vlastnostech historicky podobných provozů, které útoky skutečně byly. Takto například jsme schopní detekovat malware, jehož parametry byly mírně poupraveny, ale většina vlastností se rovná příbuzným útokům. Obecně ale neumí poskytnout informace o konkrétních features, které jsou pro útok signifikantní.

5.5 SVC

Jak jsme si ukázali v 4.1.5, Support Vector Classifier rozděluje prostor na dvě části, které odpovídají dvou binárním třídám. Podle tohoto rozdělení potom klasifikuje data. Navíc prostor, do kterého jsou body projektovány (v závislosti na požitém kernelu), neodpovídá původnímu prostoru, ve kterém se body popisující data nacházejí. Proto je proces extrakce kontextu komplikovaný a ve velkém množství případů nemožný.

5.6 LISM a FISM

Oproti předchozím klasifikátorům jsou pravidla nejvhodnější pro dobrou extrakci kontextu. Stačí totiž jednoduše použít řádek, který označil provoz za útok, a vysvětlit jednotlivé features. Tento kontext navíc můžeme rozšířit na úkor doby běhu testování. Testování neskončíme po splnění prvního pravidla, ale najdeme všechna pravidla, která řádek za útok označují. Pravidla mají také tu výhodu, že netrpí problémem s negativními features, protože jsou definována pozitivně. Důkazem kvality kontextu generovaných pravidel je i fakt, že právě díky analýze vygenerovaných pravidel bylo zjištěno, že framework CTA nelabeluje data 100% správně.

Pro demonstraci schopnosti extrakce kontextu pravidel si ukážeme a popíšeme 2 pravidla, která byla extrahovaná z dat. Vždy si popíšeme eventy pravidla a kontext, který popisují.

5.6.1 Spustitelný soubor komunikující s C&C serverem

Jako první si popíšeme jednodušší útok, který popisuje pravidlo obsahující 3 eventy:

- `ESUDOM`, eventa popisující podezřelou doménu;
- `ERERER`, opakovaný request na raw IP;
- `EMALSB`, popisující detekci škodlivého programu.

Toto pravidlo popisuje situaci, kdy program, jehož hash byl nalezen v databázi škodlivých programů, komunikuje napřímo s IP adresou, která je navíc podezřelá. Jde pravděpodobně o kontaktování C&C serveru. Získání takovéhoto kontextu není složité, ale jasně naznačuje další postup.

5.6.2 Stažení viru z DGA domény

Dalším pravidlem je:

- `ERERREQ`, popisující opakovaný request;
- `EFILED`, stažení souboru;
- `ECWDGA`, detekce DGA domény;
- `ECONCH`, kontrola připojení;
- `EAVUUA`, použití nepravděpodobného browseru.

Pravidlo popisuje situaci, kdy se uživatel dostal na DGA doménu, odkud si stáhl virus. Následovala kontrola připojení provedená virem a opakované requesty z prohlížeče, který tento uživatel nikdy předtím nepoužil. Pravděpodobně další případ kontaktování C&C serveru.

5.7 Závěr porovnání klasifikátorů

V předchozí kapitole jsme optimalizovali parametry jednotlivých klasifikátorů a následně jsme je vzájemně porovnali z pohledu precision, recall a doby běhu. Z pohledu poměru precision a recall vyšly nejlépe neuronové sítě. Mezi klasifikátory schopnými extrahovat alespoň nějaký kontext měly nejlepší výsledky náhodné lesy.

V této kapitole jsme porovnávali kontext. Ukázali jsme si, že většina tradičních klasifikátorů (kNN, SVC, Naivní Bayes a náhodné lesy) jsou schopné extrahovat kontext, ale pouze v omezené míře. Zvládají extrahovat na úrovni celých rodin malware, spíše než na úrovni jednotlivých útoků. Mají totiž problémy s negativními features, které poskytuje v kontextu a které nás často nezajímají. Z pohledu kontextu jsou výrazně nejlepší klasifikátory založené na pravidlech. Dokáží dodat kontext každému identifikovanému útoku. Navíc je možnost získat podrobnější kontext tak, že necháme data otestovat všemi pravidly.

Pro další zkoumání proto zvolíme neuronové sítě jako nejpřesnější klasifikátor. Dále náhodné lesy jako nejlepší klasifikátor poskytující kontext. A nakonec klasifikátor založený na pravidlech, protože má nejlepší schopnost obohacení detekce o kontextu.

Kapitola 6

Vlastnosti vybraných klasifikátorů

V předchozích kapitolách jsme porovnali všechny klasifikátory dle jejich základních vlastností, tedy podle:

- precision a recall,
- doba trénování a testování,
- schopnost extrakce kontextu.

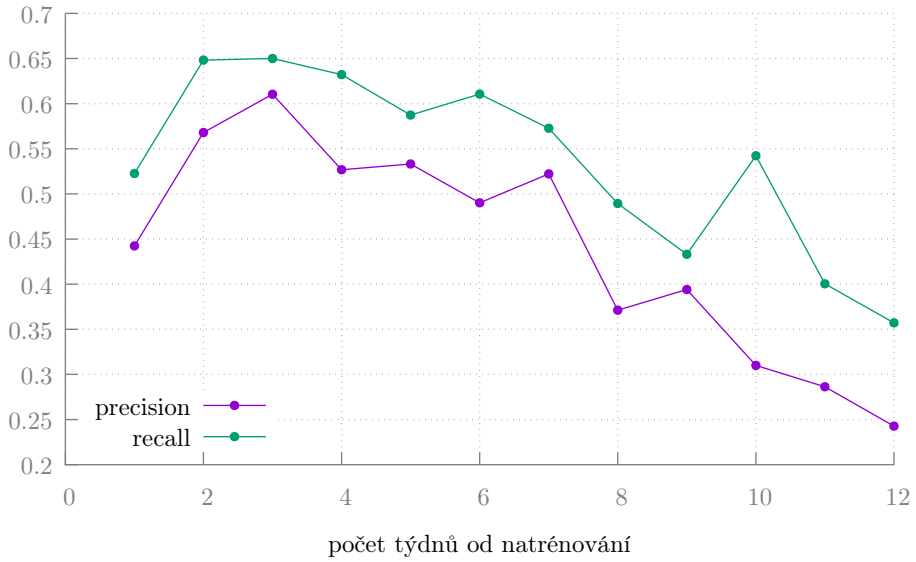
Nejlepšími klasifikátory v těchto ohledech se ukázaly neuronové sítě, náhodné lesy a klasifikátory založené na pravidlech. V této kapitole vlastnosti těchto klasifikátorů prozkoumáme podrobněji.

Přesněji nás bude zajímat schopnost udržet si precision a recall v čase. Malware se totiž v čase vyvíjí a tím pádem se mění i jeho projevy, podle kterých je klasifikátory rozpoznávají. Podíváme se na to, jak klesá precision a recall v závislosti na uplynulé době od doby natrénování.

Další testovanou vlastností bude schopnost odhalovat útoky i při snížení počtu features v datech. Vybereme omezenou množinu features, které jsou nejdůležitější pro rozhodování klasifikátorů, a ostatní budeme ignorovat. Tato redukce dává smysl, protože jí značně redukujeme prostor, ve kterém se data nacházejí, a tím výrazně snižujeme složitost zpracování dat.

6.1 Precision a recall v čase

V tomto experimentu porovnáme jednotlivé klasifikátory z pohledu schopnosti udržet precision a recall v čase. K dispozici máme celkem 13 týdnů dat, viz sekce viz sekce 2. Všechny klasifikátory natrénujeme na datech z prvního týdne a potom budeme měřit, jak se bude jejich precision a recall měnit v dalších týdnech. V tomto experimentu nepoužijeme pravidla FISM, protože nemáme



Obrázek 6.1: Schopnost náhodných lesů udržet precision a recall v čase.

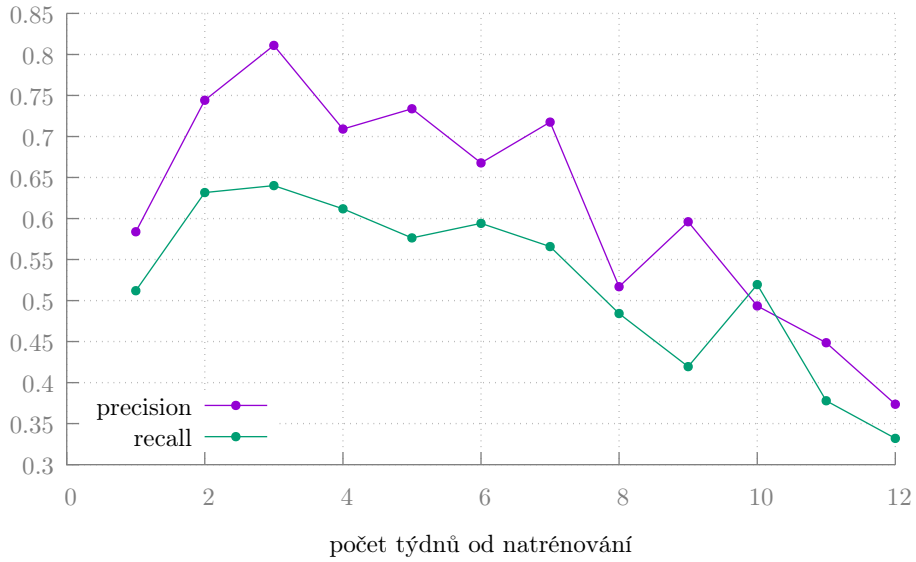
možnost jejich generování a máme k dispozici pravidla pouze z celých třech měsíců.

Pravidla LISM budeme generovat také pouze z prvního týdne a vygenerovaná pravidla vybereme stejnou selekcí, jakou jsme je vybrali v předchozím experimentu. Budeme testovat pouze vygenerovaná pravidla $\text{LISM}_{0.5}$, protože na takto malém vzorku dat, který tvoří jeden týden, je problematické vygenerovat dostatečný počet pravidel $\text{LISM}_{0.8}$ s vyšší confidence. Navíc v tuto chvíli nás více zajímá schopnost udržet precision a recall v čase, tedy změny hodnot spíše než absolutní hodnoty.

Očekáváme postupné klesání jak precision, tak recall, jak se budeme postupně vzdalovat od doby trénování modelů. Toto klesání dle očekávání způsobí proměny malware, který se neustále vyvíjí a upravuje svoje fungovaní tak, aby bojoval proti detekcím antivirových programů.

6.1.1 Náhodné lesy

Výstup z experimentu s náhodnými lesy najdeme na grafu 6.1. Na grafu můžeme vidět, že precision i recall klesá v čase poměrně rychle. Největší klesání můžeme pozorovat (až na výjimku v desátém týdnu) mezi šestým a dvanáctým týdnem. Náhodné lesy je tedy potřeba v pravidelných intervalech znovu přetrénovat na aktuálních datech. Dle našeho měření by měla být perioda opětovného učení klasifikátoru asi jeden měsíc, aby si klasifikátor zachoval svoje dobré vlastnosti.



Obrázek 6.2: Schopnost klasifikátoru používajícího neuronovou síť udržet precision a recall v čase.

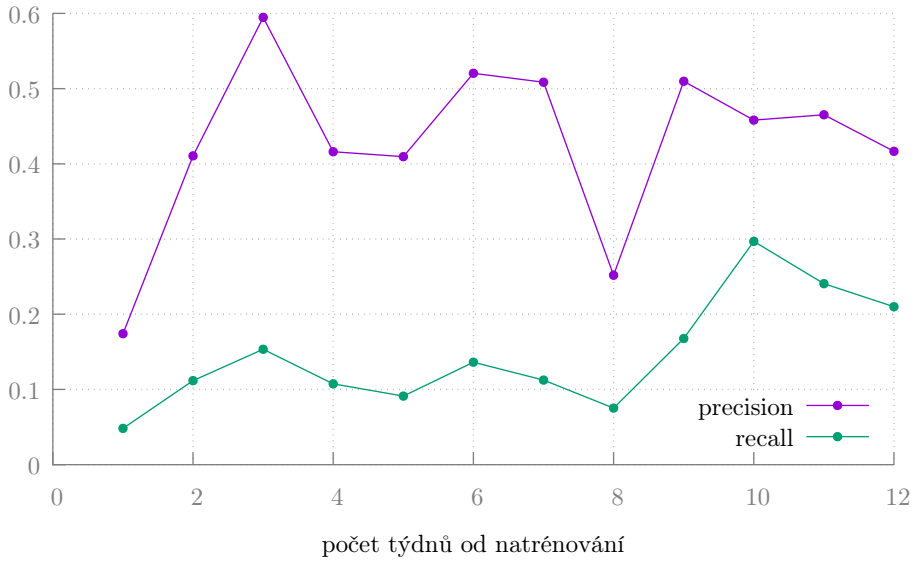
6.1.2 Neuronové sítě

Jak si v tomto experimentu vedly neuronové sítě můžeme vidět na grafu 6.2. Podobně jako v případě náhodných lesů zde můžeme vidět poměrně prudký pád precision a recall v čase. Ze stejného důvodu bude potřeba i tento klasifikátor pravidelně znovu učit na aktuálních datech, ideálně v intervalech menších než jeden měsíc.

6.1.3 LISM

Výstup z testování pro pravidla vygenerovaná pomocí LISM algoritmu najdeme na grafu 6.3. Vidíme, že precision i recall poměrně kolísají, a proto nelze jednoduše říci, že se kvalita pravidel v průběhu doby výrazně zhoršuje. Pozorovat můžeme postupné klesání precision, na druhou stranu ale vidíme, že recall mírně roste.

O klasifikátoru založeném na LISM pravidlech můžeme tvrdit, že poměrně dobře zvládá udržení precision a recall v čase. To může být způsobeno tím, že pravidla popisují obecnější vztahy mezi jednotlivými features a útoky, které se v čase příliš nemění. Tedy i kontext, který popisují, je časově poměrně stálý. Obecně by ale bylo vhodné pravidla vždy po určitém čase vygenerovat nové.



Obrázek 6.3: Schopnost pravidel vygenerovaných pomocí LISM algoritmu udržet precision a recall v čase.

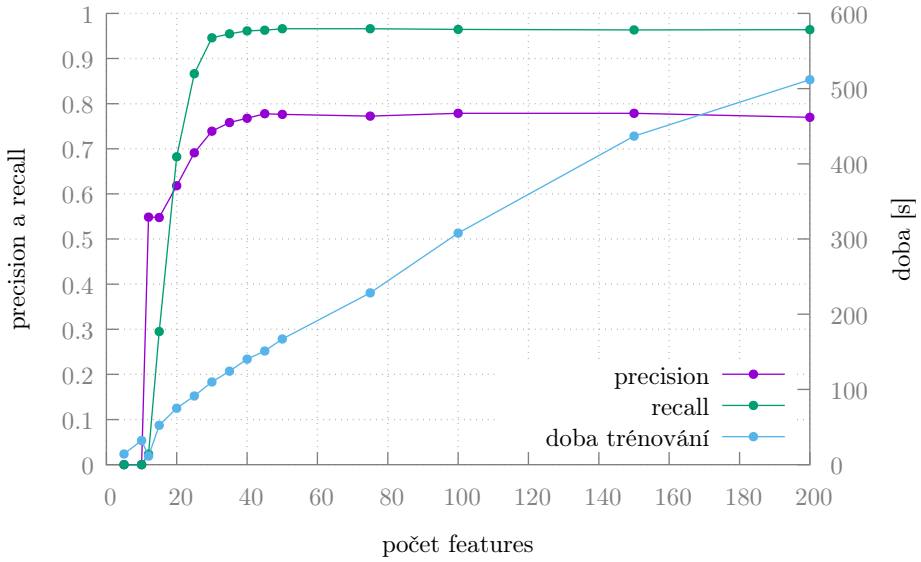
6.1.4 Zhodnocení klasifikátorů

V této sekci jsme si ukázali, že klasifikátory je potřeba průběžně znovu učit na nových datech. Nejdelším takovým intervalem je dle našeho měření jeden měsíc. Co se týče schopností jednotlivých klasifikátorů, tak neuronové sítě a náhodné lesy si vedly podobně: lze u nich vidět velkou tendenci k postupné degeneraci v čase. LISM pravidla nebyla ve výsledcích tak stabilní, ale větší tendenční zhoršení v čase jsme nezjistili. Oproti ostatním klasifikátorům testovaným v této práci mají pravidla lepší schopnost uchovat si kontext a tím pádem i kvalitu detekce v čase.

6.2 Klasifikátory a redukovaný prostor features

Jak jsme si ukázali v podsekci 2.2.1, features dat mají různou důležitost. Z toho plyne, že není potřeba uvažovat všechny features, stačí pouze ty nejdůležitější. Navíc většinu informací o klasifikaci nese pouze poměrně malá podmnožina z nich. Tohoto faktu bychom mohli využít k redukci prostoru, ve kterém se data nachází. To by mohlo mít značný vliv na snížení složitosti klasifikace těchto dat. V následující podsekci prozkoumáme, jak náhodné lesy a neuronové sítě zvládají klasifikovat právě při redukovaném počtu features.

Redukci provedeme tak, že si seřadíme všechny features dle jejich důležitosti a vybereme několik nejlepších. Využijeme pro to algoritmus Gini importance, popsaný v sekci 2.2.1. Budeme sledovat, jak tato redukce ovlivňuje



Obrázek 6.4: Vliv limitace počtu features pro náhodné lesy.

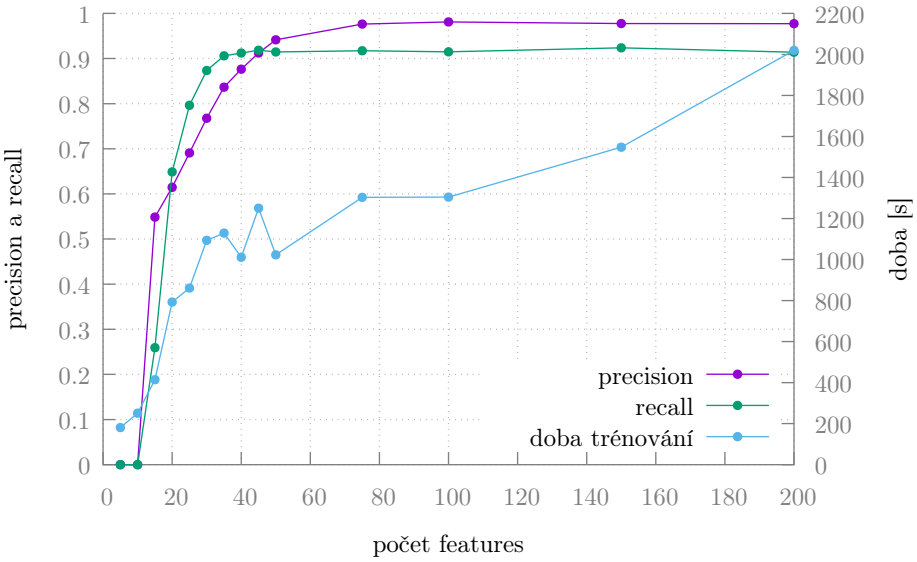
kvalitu klasifikace. Vstupní data klasifikátorů jsme upravili tak, že jsme vymazali všechny features, které nepatří mezi N nejdůležitějších. Parametr N jsme postupně zvyšovali a měřili jsme precision, recall a dobu trénování obou klasifikátorů. Použili jsme dva miliony náhodných řádků dat. Doba testování není příliš zajímavá, při experimentu vyšlo, že limitace počtu features ovlivňuje jen minimálně, proto ji z grafů vynecháváme.

6.2.1 Náhodné lesy

Vliv limitace počtu features pro náhodné lesy můžeme vidět na grafu 6.4. Z grafu je patrné, že pro klasifikaci je potřeba minimálně 12 features. Při menším počtu není klasifikátor schopen úspěšně naučit model, a proto má precision i recall roven nule. Od 12 do 30 features precision i recall rychle roste a ustaluje se na počtu 50 features. To odpovídá i zjištění ze sekce 2.2.1, že prvních 50 features reprezentuje 95% vší důležitosti. Zároveň si můžeme všimnout, že s počtem features i lineárně roste doba trénování klasifikátoru.

6.2.2 Neuronové sítě

Závislost precision, recall a doby trénování neuronových sítí můžeme vidět na grafu 6.5. Zde vidíme, že závislost jednotlivých veličin není tak přímočará jako u náhodných lesů. Podobně jako u lesů je potřeba více než 10 features pro to, aby byl klasifikátor schopen alespoň nějaké klasifikace. Potom jak precision, tak recall prudce roste. Recall se ustaluje již na 40 features, ale precision se ustaluje až na 75. Lesy mají tedy lepší schopnost vyrovnat se s menším počtem



Obrázek 6.5: Vliv limitace počtu features pro neuronové sítě.

features než neuronové sítě. Doba učení v roste od 5 features do 50 poměrně strmě, poté již roste pomaleji.

6.2.3 Zhodnocení

V této podsekci jsme potvrdili závěr předchozí podsekce, že pro klasifikaci není potřeba uvažovat kompletní množinu features. Náhodné lesy totiž disponují téměř stejnou precision a recall při 50 použitých features jako při 200. Jenom rychlost učení je méně než poloviční. Neuronové sítě vykazují podobné vlastnosti, jenom potřebují pro dosažení maximální kvality asi 75 features.

6.3 Navržený systém kontextové detekce

Na základě experimentů v této kapitole navrhujeme systém detekce síťových incidentů obsahující více klasifikátorů. Tím prvním jsou neuronové sítě. Neuronové sítě se ukázaly jako velice dobrý klasifikátor pro náš problém, precision mají kolem 95% a recall nad 90%. Proto v našem systému budou zastupovat pozici primárního detektoru incidentů. Samy ale neposkytují kontext, proto do systému navrhujeme další klasifikátory.

Náhodné lesy jsou schopné podávat omezený kontext incidentu, jak jsme si ukázali v podsekci 5.1. Výhodou náhodných lesů je, že jsou schopné plně klasifikovat již při redukci dimenze prostoru features na 50, což umožňuje výpočetní efektivitu. Jsou tedy první instancí, která k incidentům detekovaným neuronovými sítěmi přidává kontext. Bohužel ale poskytovaný kontext

není tak obsáhlý a hodí se spíše na popis chování celé rodiny než jednotlivých útoků. Proto potřebujeme lepší zdroj kontextu.

Hlavním zdrojem kontextu incidentů našeho systému budou klasifikátory založené na pravidlech. Zdrojem těchto pravidel může být jak FISM, tak LISM, ale i jakýkoliv jiný algoritmus, který je schopen generovat asociační pravidla (např. algoritmus Explainer). Největší předností klasifikátorů založených na pravidlech je jejich extrémně rychlá evaluace při použití prefixového stromu. Mají také velice dobrou kvalitu podávaného kontextu, který totiž tvoří množina všech pravidel, která data splňují. Úkolem pravidlových klasifikátorů tedy bude obohacování incidentů detekovaných předchozími klasifikátory o kontext. Menší přesnost pravidel můžeme redukovat právě pomocí neuronových sítí tak, že pravidlové klasifikátory nebudou detekovat útoky, ale již detekované útoky budou obohacovat o kontext.

Závěr

Cílem této práce bylo porovnat vlastnosti různých klasifikátorů na reálných síťových datech z frameworku CTA (Cognitive Threat Analytics). Speciálně jsme se zaměřili na schopnost extrahovat kontext útoků. Pro účely této práce jsme jako klasifikátory vybrali náhodné lesy, neuronové sítě, k-nejbližších sousedů, SVC (Support Vector Classifier), Naivní Bayes a pravidla vygenerovaná pomocí algoritmů LISM (Logical Itemset Mining) a FISM (Frequent Itemset Mining).

V první části práce jsme si představili data, na kterých budeme provádět experimenty. Ukázali jsme si jejich strukturu a popsali jsme si rodiny malware, kterými jsou data olabelovaná. Pro tyto rodiny jsme si zjistili důležitost jednotlivých features. Prvním zjištěním je, že jednotlivé rodiny malware mají rozdílné nejdůležitější features. To souvisí s rozdílností kontextu incidentů jednotlivých rodin. Dalším výstupem této části byl fakt, že pouze poměrně malá množina features se podílí na klasifikaci. Z celkové množiny 202 features se na klasifikaci významně podílí asi pouze 50. Z nich pouhých 10-15 mají v součtu více než 50% důležitosti.

V další části jsme se zaměřili na generování a evaluaci pravidel. Představili jsme si algoritmy FISM a jeho variantu FP-Growth. Dále jsme si ukázali principy algoritmu LISM. Z experimentů se ukázalo, že LISM není příliš vhodný pro generování klasifikačních pravidel, protože potřebuje existující vztahy mezi všemi elementy pravidla (tedy že tvoří kliku grafu), což se často neděje. Proto jsme zvolili metodu nízkých prahů algoritmu a následné selekce relevantních pravidel. V poslední části jsme demonstrovali metodu efektivní evaluace pravidel pomocí prefixového stromu, který dle experimentů je až 1000krát rychlejší, než naivní algoritmus.

Abychom byli schopni objektivně porovnat jednotlivé klasifikátory, v první části jsme optimalizovali jejich parametry na omezené podmnožině dat tak, aby podávaly co nejlepší výsledky. Následně jsme klasifikátory s optimalizovanými parametry porovnali měřením na velké množině dat. Měřili jsme jejich precision, recall, dobu trénování a testování. Nakonec jsme porovnali i jejich

schopnost extrahovat kontext útoku. Některé klasifikátory se ukázaly jako nepoužitelné, kvůli své časové neefektivitě, šlo o SVC a k-nejbližších sousedů. Klasifikátor Naivní Bayes se ukázal jako nevhodný kvůli malé precision a recall. Nejlepší poměr precision a recall měly neuronové sítě a dále náhodné lesy. Z pohledu poskytování kontextu se ukázaly jako nejlepší klasifikátory založené na pravidlech. Jsou totiž schopné podat kontext všech detekovaných incidentů a zároveň nemají problém s negativními features. Pro další experimenty jsme tedy zvolili tři klasifikátory: neuronové sítě, náhodné lesy a pravidla.

Zvolené klasifikátory jsme v další části zkoumali z pohledu jejich schopnosti udržet jejich precision a recall v čase. Malware se totiž neustále vyvíjí, a proto je validní předpoklad, že klasifikátory postupně ztrácejí schopnost incidenty detekovat. Náš experiment tuto domněnku potvrdil. Neuronové sítě a náhodné lesy je potřeba nejdéle po měsíci znovu natrénovat na aktuálních datech tak, aby si udržely svoje dobré klasifikační vlastnosti. Experiment také ukázal, že dobře navržená pravidla udrží precision a recall v čase déle díky lepší schopnosti udržet validní kontext v čase.

Klasifikátory náhodné lesy a neuronové sítě jsme potom porovnávali dle jejich schopnosti klasifikovat data nacházející se v redukovaném prostoru features. Zjistili jsme, že náhodné lesy potřebují pro plnou kvalitu asi 50 features. Neuronové sítě těchto features potřebují 75.

Na základě výše uvedených experimentů jsme navrhli vlastní systém detekce síťových incidentů poskytující jejich kontext. Tento systém se sestává z neuronové sítě jako primárního detektoru incidentů, dále z náhodných lesů, jako prvního zdroje kontextu a další úrovně detekce. V poslední řade jsou zde pravidlové klasifikátory, které obohacují detekované incidenty o jejich kontext.

Na základě experimentů zmíněných v této práci se nám otevřelo několik dalších otázek. Při analýze LISM jsme zjistili, že by bylo vhodné vylepšit tento algoritmus tak, aby nehledal pouze kliky v grafu ale efektivněji extrahoval informace z napočítané matice konzistencí. Dále by bylo zajímavé navrhnout vhodný systém redukce features tak, aby reflektoval rozdílnost důležitosti pro jednotlivé skupiny malware. Dalším zajímavou oblastí, které jsme se v této práci dotkli, je metoda extrakce kontextu z neuronových sítí. Pokud by totiž neuronové sítě byly schopné podat i klasifikační kontext, staly by se dle našich experimentů bezkonkurenčně nejlepším klasifikátorem. V neposlední řadě by také bylo vhodné implementovat a více rozpracovat koncept námi navrženého detekčního systému.

Literatura

[1] Yagoda, B.: A Short History of "Hack". Jun 2017. Dostupné z: https://www.newyorker.com/tech/elements/a-short-history-of-hack

[2] Perlroth, N.: All 3 Billion Yahoo Accounts Were Affected by 2013 Attack. Oct 2017. Dostupné z: https://www.nytimes.com/2017/10/03/technology/yahoo-hack-3-billion-users.html

[3] Francis, R.: MySpace becomes every hackers' space with top breach in 2016, report says. Feb 2017. Dostupné z: https://www.csoonline.com/article/3166846/data-breach/myspace-becomes-every-hackers-space-with-top-breach-in-2016-report-says.html

[4] Storm, D.: Biggest hack of 2016: 412 million FriendFinder Networks accounts exposed. Nov 2016. Dostupné z: https://www.computerworld.com/article/3141290/security/biggest-hack-of-2016-412-million-friendfinder-network-accounts-exposed.html

[5] Cost Of Cyber Crime Study. Jan 2018. Dostupné z: https://www.accenture.com/t20170926T072837Z__w__/us-en/_acnmedia/PDF-61/Accenture-2017-CostCyberCrimeStudy.pdf

[6] Bradley, T.: Gartner Predicts Information Security Spending To Reach $93 Billion In 2018. Aug 2017. Dostupné z: https://www.forbes.com/sites/tonybradley/2017/08/17/gartner-predicts-information-security-spending-to-reach-93-billion-in-2018

[7] Zákon č. 181/2014 Sb., Zákon o kybernetické bezpečnosti a o změně souvisejících zákonů. 2014. Dostupné z: https://www.zakonyprolidi.cz/cs/2014-181

[8] Anderson, J. P.: Computer Security Threat Monitoring and Surveillance. 1980. Dostupné z: http://seclab.cs.ucdavis.edu/projects/history/papers/ande80.pdf

[9] Bruneau, G.: The History and Evolution of Intrusion Detection. 2001. Dostupné z: https://www.sans.org/reading-room/whitepapers/detection/history-evolution-intrusion-detection-344

[10] Heberlein, L. T.; Dias, G. V.; Levitt, K. N.; aj.: A network security monitor. In *Research in Security and Privacy, 1990. Proceedings., 1990 IEEE Computer Society Symposium on*, IEEE, 1990, s. 296–304. Dostupné z: http://seclab.cs.ucdavis.edu/papers/pdfs/th-gd-90.pdf

[11] Schwab, P.: The History of Intrusion Detection Systems (IDS) – Part 1 – Threat Stack. Sep 2015. Dostupné z: https://www.threatstack.com/blog/the-history-of-intrusion-detection-systems-ids-part-1/

[12] Eilertson, E.; Lazarevic, A.; Tan, P.-N.; aj.: Minds-minnesota intrusion detection system. 2004. Dostupné z: http://minds.cs.umn.edu/papers/minds_chapter.pdf

[13] NetFlow Export Datagram Format. Mar 2015. Dostupné z: https://www.cisco.com/c/en/us/td/docs/net_mgmt/netflow_collection_engine/3-6/user/guide/format.html#wp1006108

[14] Breunig, M. M.; Kriegel, H.-P.; Ng, R. T.; aj.: LOF: identifying density-based local outliers. In *ACM sigmod record*, ročník 29, ACM, 2000, s. 93–104. Dostupné z: http://www.dbs.ifi.lmu.de/Publikationen/Papers/LOF.pdf

[15] Grill, M.: Combining Network Anomaly Detectors. 2016. Dostupné z: https://dspace.cvut.cz/bitstream/handle/10467/66683/Disertace_Grill_2016.pdf?sequence=1

[16] Evangelista, P. F.; Embrechts, M. J.; Szymanski, B. K.: Data fusion for outlier detection through pseudo-ROC curves and rank distributions. In *Neural Networks, 2006. IJCNN'06. International Joint Conference on*, IEEE, s. 2166–2173. Dostupné z: https://www.cs.rpi.edu/~szymansk/papers/ijcnn.06.pdf

[17] Altmann, A.; Toloşi, L.; Sander, O.; aj.: Permutation importance: a corrected feature importance measure. *Bioinformatics*, ročník 26, č. 10, 2010: s. 1340–1347. Dostupné z: https://academic.oup.com/bioinformatics/article/26/10/1340/193348

[18] Breirnan, L.; Friedman, J.; Olshen, R.; aj.: Classification and regression trees. *Belrnont: WadsWorth*, 1984.

[19] Kopp, M.; Holena, M.: Evaluation of Association Rules Extracted during Anomaly Explanation. In *ITAT*, 2015, s. 143–149. Dostupné z: https://pdfs.semanticscholar.org/c74f/c2a240229260d21b32d2376a1a89313d0286.pdf

[20] Hájek, P.; Havel, I.; Chytil, M.: The GUHA method of automatic hypotheses determination. *Computing*, ročník 1, č. 4, 1966: s. 293–308.

[21] Agrawal, R.; Imieliński, T.; Swami, A.: Mining association rules between sets of items in large databases. In *Acm sigmod record*, ročník 22, ACM, 1993, s. 207–216. Dostupné z: http://www.it.uu.se/edu/course/homepage/infoutv/ht08/agrawal93mining.pdf

[22] Zaki, M. J.: Scalable algorithms for association mining. *IEEE transactions on knowledge and data engineering*, ročník 12, č. 3, 2000: s. 372–390. Dostupné z: http://www.cs.ecu.edu/~dingq/CSCI6905/readings/zaki00scalable.pdf

[23] Lucchese, C.; Orlando, S.; Palmerini, P.; aj.: kDCI: A multi-strategy algorithm for mining frequent sets. In *Proceedings of the IEEE ICDM Workshop of Frequent Itemset Mining Implementations (FIMI), Melbourne, Florida*, Citeseer, 2003. Dostupné z: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.1.8460&rep=rep1&type=pdf

[24] Uno, T.; Kiyomi, M.; Arimura, H.: LCM ver. 2: Efficient mining algorithms for frequent/closed/maximal itemsets. In *Fimi*, ročník 126, 2004. Dostupné z: http://www.philippe-fournier-viger.com/spmf/LCM2.pdf

[25] Borgelt, C.: Recursion Pruning for the Apriori Algorithm. In *FIMI*, ročník 126, 2004. Dostupné z: http://borgelt.net/papers/fimi_04.ps.gz

[26] Han, J.; Pei, J.; Yin, Y.: Mining frequent patterns without candidate generation. In *ACM sigmod record*, ročník 29, ACM, 2000, s. 1–12. Dostupné z: http://www.cse.ust.hk/~raywong/comp5331/References/MiningFrequentPatternsWithoutCandidateGeneration.pdf

[27] Kumar, S.; Chandrashekar, V.; Jawahar, C.: Logical itemset mining. In *Data Mining Workshops (ICDMW), 2012 IEEE 12th International Conference on*, IEEE, 2012, s. 603–610.

[28] Tomita, E.; Tanaka, A.; Takahashi, H.: The worst-case time complexity for generating all maximal cliques and computational experiments. *Theoretical Computer Science*, ročník 363, č. 1, 2006: s. 28–42. Dostupné z: https://snap.stanford.edu/class/cs224w-readings/tomita06cliques.pdf

[29] scikit-learn | Machine Learning in Python. Dostupné z: http://scikit-learn.org/stable/

[30] Dennis, J. E., Jr; Moré, J. J.: Quasi-Newton methods, motivation and theory. *SIAM review*, ročník 19, č. 1, 1977: s. 46–89. Dostupné z: https://hal.archives-ouvertes.fr/hal-01495720/document

[31] Bottou, L.: Large-scale machine learning with stochastic gradient descent. In *Proceedings of COMPSTAT'2010*, Springer, 2010, s. 177–186. Dostupné z: http://leon.bottou.org/publications/pdf/compstat-2010.pdf

[32] Kingma, D. P.; Ba, J.: Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014. Dostupné z: https://arxiv.org/pdf/1412.6980.pdf

[33] scikit-learn documentation |sklearn.neural_network.MLPClassifier. Dostupné z: http://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPClassifier.html

[34] 1.4. Support Vector Machines. Dostupné z: http://scikit-learn.org/stable/modules/svm.html#scores-and-probabilities

[35] Chen, S. F.; Goodman, J.: An empirical study of smoothing techniques for language modeling. *Computer Speech & Language*, ročník 13, č. 4, 1999: s. 359–394. Dostupné z: https://arxiv.org/pdf/cmp-lg/9606011.pdf

[36] Jaynes, E. T.: Prior probabilities. *IEEE Transactions on systems science and cybernetics*, ročník 4, č. 3, 1968: s. 227–241. Dostupné z: http://knuthlab.rit.albany.edu/courses/2014/BayesianDataAnalysis/papers/jaynes68.pdf

Příloha **A**

Seznam použitých zkratek

MIT Massachusetts Institute of Technology

US-CERT United States Computer Emergency Readiness Team

MINDS Minnesota Intrusion Detection System

DGA Domain Generating Algorithm

CTA Cognitive Threat Analytics

FISM Frequent Itemset Mining

LISM Logical Itemset Mining

RF Random Forrest

SVM Support Vector Machines

SVC Support Vector Classifier

kNN K-nearest Neighbors

IDS Intrusion Detection System

LOF Local Outlier Factor

DOS Denial of Service

TSV Tab-Separated Value

RAM Random Access Memory

HDD Hard Disc Drive

PCA Principal Component Analysis

LAMS Local Adaptive Multivariate Smoothing

TP True Positive

FP False Positive

TSV Tab Separated Value

USB Universal Serial Bus

HTTP Hypertext Transfer Protocol

Příloha **B**

Obsah přiloženého CD

- `readme.txt` ..... popis obsahu CD
- `programs` ..... složka se spustitelnými Python soubory
  - `01_LISM` ..... implementace LISM algoritmu
  - `02_prefixTree` ..... implementace prefixového stromu
  - `03_parameterOptimalization` .. optimalizace parametrů klasifikátorů
  - `04_evaluateRuleByRule` ..... testování řádků pravidel
  - `05_qualityInTime` ..... testování kvality klasifikátorů v čase
  - `06_reductedFeatures` ..... testování redukovaných features
  - `data` ..... soubory s daty
- `thesis`
  - `thesis.pdf` ..... práce ve formátu PDF
  - `thesis.tex` ..... zdrojový soubor práce ve formátu LaTeX