Vizualizace algoritmu Isolation Forest

Ing. Alena Strnadová

Bakalářská práce
2024

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky
Ústav informatiky a umělé inteligence

Akademický rok: 2023/2024

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

(projektu, uměleckého díla, uměleckého výkonu)

Jméno a příjmení:	**Ing. Alena Strnadová**
Osobní číslo:	**A21708**
Studijní program:	**B0613A140020 Softwarové inženýrství**
Forma studia:	**Kombinovaná**
Téma práce:	**Vizualizace algoritmu Isolation Forest**
Téma práce anglicky:	**The Visualization of the Isolation Forest Algorithm**

Zásady pro vypracování

1. Vypracujte literární rešerši na téma algoritmus Isolation Forest.
2. Popište použité pracovní prostředí.
3. Vytvořte knihovnu implementující Isolation Forest, včetně nastavení vstupů.
4. Vytvořte vizualizaci výsledků do různých formátů dle volby uživatele.
5. Knihovnu otestujte, zdokumentujte a zveřejněte ve veřejném repozitáři.

Forma zpracování bakalářské práce: **tištěná/elektronická**

Seznam doporučené literatury:

1. ROSEN, K H, 2019. *Discrete Mathematics and Its Applications*. McGraw-Hill. ISBN 9781259676512. Dostupné také z: https://books.google.cz/books?id=efMmtAEACAAJ
2. MADHAVAPEDDY, A a Y MINSKY, 2022. *Real World OCaml: Functional Programming for the Masses: Functional Programming for the Masses*. Cambridge University Press. ISBN 9781009125802. Dostupné také z: https://books.google.cz/books?id=v-WBEAAAQBAJ
3. MINSKY, Y, A MADHAVAPEDDY a J HICKEY, 2013. *Real World OCaml: Functional programming for the masses: Functional programming for the masses*. O'Reilly Media. ISBN 9781449324759. Dostupné také z: https://books.google.cz/books?id=kajtAQAAQBAJ
4. LIU, Fei Tony, Kai Ming TING a Zhi-Hua ZHOU, 2012. Isolation-based anomaly detection. *ACM Transactions on Knowledge Discovery from Data (TKDD)*. ACM New York, NY, USA, **6**(1), 1-39.
5. LIU, Fei Tony, Kai Ming TING a Zhi-Hua ZHOU, 2008. *Isolation forest*. 413-422.

Vedoucí bakalářské práce: **Ing. Adam Ulrich**
Ústav informatiky a umělé inteligence

Datum zadání bakalářské práce: **5. listopadu 2023**
Termín odevzdání bakalářské práce: **13. května 2024**

UNIVERZITA TOMÁŠE BATI VE ZLÍNĚ
Fakulta aplikované informatiky
DĚKAN
L.S.

doc. Ing. Jiří Vojtěšek, Ph.D. v.r.
děkan

prof. Mgr. Roman Jašek, Ph.D., DBA v.r.
ředitel ústavu

Ve Zlíně dne 5. ledna 2024

Jméno, příjmení: Ing. Alena Strnadová
Název bakalářské práce: Vizualizace algoritmu Isolation Forest

Prohlašuji, že

- beru na vědomí, že odevzdáním bakalářské práce souhlasím se zveřejněním své práce podle zákona č. 111/1998 Sb. o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších právních předpisů, bez ohledu na výsledek obhajoby;
- beru na vědomí, že bakalářská práce bude uložena v elektronické podobě v univerzitním informačním systému dostupná k prezenčnímu nahlédnutí, že jeden výtisk bakalářské práce bude uložen v příruční knihovně Fakulty aplikované informatiky Univerzity Tomáše Bati ve Zlíně;
- byl/a jsem seznámen/a s tím, že na moji bakalářskou práci se plně vztahuje zákon č. 121/2000 Sb. o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon) ve znění pozdějších právních předpisů, zejm. § 35 odst. 3;
- beru na vědomí, že podle § 60 odst. 1 autorského zákona má UTB ve Zlíně právo na uzavření licenční smlouvy o užití školního díla v rozsahu § 12 odst. 4 autorského zákona;
- beru na vědomí, že podle § 60 odst. 2 a 3 autorského zákona mohu užít své dílo – bakalářskou práci nebo poskytnout licenci k jejímu využití jen připouští-li tak licenční smlouva uzavřená mezi mnou a Univerzitou Tomáše Bati ve Zlíně s tím, že vyrovnání případného přiměřeného příspěvku na úhradu nákladů, které byly Univerzitou Tomáše Bati ve Zlíně na vytvoření díla vynaloženy (až do jejich skutečné výše) bude rovněž předmětem této licenční smlouvy;
- beru na vědomí, že pokud bylo k vypracování bakalářské práce využito softwaru poskytnutého Univerzitou Tomáše Bati ve Zlíně nebo jinými subjekty pouze ke studijním a výzkumným účelům (tedy pouze k nekomerčnímu využití), nelze výsledky bakalářské práce využít ke komerčním účelům;
- beru na vědomí, že pokud je výstupem bakalářské práce jakýkoliv softwarový produkt, považují se za součást práce rovněž i zdrojové kódy, popř. soubory, ze kterých se projekt skládá. Neodevzdání této součásti může být důvodem k neobhájení práce.

Prohlašuji,

- že jsem na bakalářské práci pracovala samostatně a použitou literaturu jsem citovala. V případě publikace výsledků budu uvedena jako spoluautor.
- že odevzdaná verze bakalářské práce a verze elektronická nahraná do IS/STAG jsou totožné.

Ve Zlíně, dne 9.5.2024 Alena Strnadová v.r.

ABSTRAKT

Tato bakalářská práce se věnuje vizualizaci algoritmu Isolation Forest, který je určen pro detekci anomálií. V teoretické části práce je popsán princip algoritmu, včetně základních informací o strojovém učení a detekci anomálií. V praktické části je navržena a implementována knihovna pro vizualizaci jednotlivých binárních stromů, která je otestována zdokumentována a zveřejněna jako balíček npm. Teoretická část i zveřejněná knihovna může sloužit jak odborníkům z praxe, tak studentům a dalším zájemcům pro získání povědomí o algoritmu Isolation Forest.

Klíčová slova: Isolation Forest, vizualizace, strojové učení, detekce anomálií, anomálie, npm, JavaScript, Graphviz

ABSTRACT

This bachelor thesis focuses on the visualization of the Isolation Forest algorithm, designed for anomaly detection. The theoretical part of the thesis describes the principle of the algorithm, including basic information about machine learning and anomaly detection. In the practical part, a library for visualizing individual binary trees is designed, implemented, tested, documented, and published as an npm package. Both the theoretical part and the published library can serve practitioners, students, and other interested parties to gain an understanding of the Isolation Forest algorithm.

Keywords: Isolation Forest, visualization, machine learning, anomaly detection, anomaly, npm, JavaScript, Graphviz

Tímto děkuji vedoucímu své bakalářské práce, Ing. Adamu Ulrichovi, za cenné rady a připomínky k vypracování. Dále děkuji každému, kdo mi kdy jakkoliv pomohl nebo pomáhá a drží mi palce. Také děkuji Tomáši Baťovi za to, že můj rodný Zlín není malým zapadlým městečkem, jakým by bez něj byl.

Prohlašuji, že odevzdaná verze bakalářské práce a verze elektronická nahraná do IS/STAG jsou totožné.

OBSAH

ÚVOD

Algoritmus Isolation Forest, sloužící k detekci anomálií byl poprvé představen v roce 2008. Využívá strojové učení bez učitele a jeho činnost má dvě fáze. V první fázi algoritmus vygeneruje na základě vzorku dat a jejich izolace sadu binárních rozhodovacích stromů, kdy jsou data postupně dělena podle náhodně zvoleného parametru a jeho náhodné hodnoty. Ve druhé fázi jsou pak všechna data analyzována pomocí těchto stromů a na základě zařazení prvků do uzlu v určité hloubce stromu je rozhodnuto o jejich skóre anomálie, tedy o tom, jak pravděpodobné je, že daný prvek dat je nebo není anomálií. Jelikož má algoritmus Isolation Forest pouze dva vstupní parametry, je celkem jednoduchý na pochopení i implementaci a těší se v posledních letech velké oblibě. Jeho výhodou je lineární závislost na čase s nízkou konstantou a nízké nároky na paměť. Na rozdíl od jiných metod, které vytvářejí profil normálních dat a anomálie nacházejí jako data od nich odlišná, Isolation Forest využívá princip přímé izolace anomálií bez nutnosti profilování normálních dat.

Tato práce se nejprve věnuje vysvětlení souvisejících pojmů z oboru detekce anomálií a strojového učení, poté je popsán samotný algoritmus Isolation Forest, následuje stručný popis zvolených technologií – jazyka JavaScript pro implementaci v praktické části práce. Úkolem práce je zejména vizualizace algoritmu. K tomuto účelu byl zvolen software *Graphviz*, kdy si uživatel může zvolit export do různých výstupních formátů. Celá knihovna pro vygenerování Isolation Forest a vizualizaci jednotlivých stromů je vytvořena jako balíček *npm*. Zmíněný software a metody práce jsou také v následujících kapitolách popsány.

V praktické části práce je vizualizace algoritmu Isolation Forest představena formou návrhu knihovny a popisu její implementace. Následně je knihovna otestována, uložena na veřejném depozitáři a zveřejněna jako balíček *npm*. Práci uzavírá dokumentace knihovny s uvedením příkladů použití.

I. TEORETICKÁ ČÁST

1 DETEKCE ANOMÁLIÍ

Detekce anomálií, která má velký význam ve zpracování dat, se zabývá odhalováním neobvyklých objektů, jevů nebo pozorování. Anomálie se mohou vyskytovat např. u bankovních transakcí, v medicíně či ve výrobě. Jejich rozeznání má pro zmíněné obory různé důsledky, vždy je ale důležité. Lze takto odhalit podvodné transakce, zdravotní riziko či nesprávnou funkci stroje. [1]

1.1 Anomálie

V literatuře existují různé definice anomálie, přičemž v angličtině existují výrazy *anomaly* a *outlier*. Zatímco někteří autoři tyto pojmy používají jako synonyma [2][3], jiní je rozlišují. Např. Adari [4] vysvětluje rozdíl tak, že bychom měli očekávat, že data budou obsahovat *outlier* prvky, ale neměli bychom očekávat *anomaly* prvky. Přestože jsou pojmy někdy zaměňovány, *anomaly* není vždy *outlier*, a ne všechny prvky *outlier* jsou *anomaly*. Jako příklad rozdílu je zde uvedeno pozorování stovek labutí u jezera, kdy dosud byly všechny bílé a najednou po letech přiletí černá labuť. Tato černá labuť je považována za *anomaly*, kdežto velmi velká nebo velmi malá bílá labuť by spadala do kategorie *outlier*. K vymezení pojmů uvádí Feasel [5], že *outlier* má hodnoty, které se výrazně odlišují od normálních hodnot. Na olympijských hrách v roce 2008 zvítězil Usian Bolt v závodě na 200 metrů s časem 19,30s s náskokem 0,66s před druhým nejlepším atletem. Druhý až šestý sprinter (více závod nedokončilo) byly vměstnáni do dalších 0,63s. Boltův velmi odlišný výkon lze považovat za *outlier*, kdežto výkony ostatních 5 atletů jsou *inlier* (normální hodnoty). Dále je zde uvedeno, že *outlier* se dělí na *anomaly* a *noise* (šum) a rozdíl mezi těmito dvěma podkategoriemi je věcí lidského zájmu. *Anomaly* je prvek, který je zajímavý, naproti tomu *noise* je nezajímavý. Podle tohoto přístupu se tedy dá říct, že *anomaly* je zvláštní případ *outlier*, a to takový, který je z nějakého důvodu zajímavý.

Pro kontext této práce bude dále používáno výhradně výrazu anomálie. Dle Kishana [3] je významnou vlastností anomálií, že se odchylují od normálu. Příkladem je skóre IQ (inteligenční kvocient), u kterého je za střední hodnotu považováno 100 se směrodatnou odchylkou 15. Tak lze IQ skóre 145 považovat za anomálii, protože se od střední hodnoty odchyluje o trojnásobek směrodatné odchylky. Je celkem snadné intuitivně vytušit anomálie u IQ skóre, protože se jedná o problém v jedné dimenzi. Přestože u vícedimenzionálních dat je problematika komplexnější, stále pro anomálie platí, že se výrazně odchylují od normálních dat.

1.1.1 Kategorie anomálií

Anomálie lze dělit na tři základní kategorie podle toho, co je jejich základem: data (*data-based*), kontext (*context-based*), vzor (*pattern-based*). [4]

Anomálie založené na datech mohou být například údaje pro diagnostiku ve zdravotnictví. V sadě dat naměřených při vyšetření mnoha pacientů je většina hodnot normálních a anomálie mohou znamenat zjištění určité diagnózy.

Anomálie založené na kontextu by za určitých okolností byly považovány za normální hodnoty, ale na základě vyhodnocení kontextu jsou anomální. Pokud klienti banky utrácejí více v době, kdy je to typické (např. v období svátků), budou tyto transakce vyhodnoceny jako normální. V období, kdy většina lidí utrácí průměrně však velké výdaje klienta mohou vzbudit pozornost.

Anomálie založené na vzoru se odlišují od předchozího pozorování. Extrémní počasí může způsobit velký úbytek zájmu o určitou aktivitu ve srovnání s průměrnými hodnotami v určitém týdnu v roce pozorovanými v předchozích letech.

1.1.2 Využití detekce anomálií

Detekce anomálií hraje roli v rozhodovacím procesu mnoha oborů a může být využita téměř všude, kde dochází ke sběru dat. V této podkapitole jsou stručně zmíněny některé obory a oblasti, ve kterých se typicky uplatňuje. [4][5]

Finančnictví využívá detekce anomálií pro odhalování podezřelých transakcí i neobvyklých změn cen akcií, které by mohly naznačovat nelegální činnost.

Medicína poskytuje mnoho způsobů uplatnění pro detekci anomálií. Diagnostika různých nemocí analýzou výsledků vyšetření je typický příklad. Srovnávání hodnot jako jsou teplota těla, krevní tlak a okysličení krve s normálními hodnotami (od jiných pacientů, nebo dříve naměřenými u stejného pacienta) může odhalit problémy se zdravotním stavem.

Sport může být profitabilní způsob podnikání a majitelé velkých klubů investují nemalé částky do analýzy dat. Snaží se tak zajistit zisky i v dalších sezónách.

Únik dat z firem ve formě neúmyslného nebo úmyslného odeslání tajných informací může být odhalen pomocí detekce anomálií, ať už se jedná o malou společnost nebo mezinárodní korporaci.

Krádeže identity jsou v dnešní online době velkým problémem, zejména v podobě ukradených dat k platebním kartám. Jak již bylo zmíněno u anomálií založených na kontextu, lze pomocí detekce odhalit neobvyklé transakce klientů bank, které mohou naznačovat, že s údaji ke kartě disponuje někdo jiný.

Průmyslová výroba využívá detekce anomálií zejména při kontrole kvality výrobků, ať už se jedná o strojírenskou, potravinářskou, textilní či jinou produkci.

Networking představuje jedno z nejtypičtějších využití detekce anomálií [4], často se jedná o zjištění útoků hackerů na vládní weby kvůli úniku citlivých dat. V menším měřítku se hackeři zaměřují i na lokální sítě. Vždy je však třeba automatizovaného systému pro odhalení těchto útoků.

1.2 Strojové učení

Strojové učení je jedním z oborů umělé inteligence a vzniklo z potřeby provádět časově náročné a monotónní operace pomocí počítače. Umožňuje strojům učit se ze stávajících informací a toto učení využít pro provedení obdobných úkolů. Stroj se učí z předchozích dat a předvídá nová data. [6]

Umělá inteligence i strojové učení se v posledních letech těší velké pozornosti médií i odborníků. Díky výpočetnímu výkonu a vývoji algoritmů tento obor mění celou společnost, a i když si to mnozí neuvědomují, vyskytuje se v dnešním světě na každém kroku. Typickým příkladem je nabízení produktů uživatelům na webových stránkách, hlasoví asistenti nebo vyhledávač Google. [4]

Pro potřeby detekce anomálií se využívají tři druhy strojového učení: *supervised*, *unsupervised* a *semisupervised*. [3][4]

1.2.1 Supervised – učení s učitelem

Metoda využívá pro učení data, která jsou označená jako normální nebo anomální. Všechna data musí být předem zpracována a označena. Model je tedy obeznámen s tím, jak mají jednotlivé druhy dat vypadat a na základě toho se učí, jak postupovat u nových dat. Jedná se o binární klasifikaci, kdy systém rozhoduje, do které ze dvou kategorií mají data patřit.

1.2.2 Unsupervised – učení bez učitele

Učení u této metody probíhá na datech, která nejsou označena. Od systému je očekáváno, že po fázi učení rozhodne, která data jsou normální a která anomální. Příkladem metody je i algoritmus Isolation Forest, kterým se zabývá tato práce.

1.2.3 Semi-Supervised

Tato metoda využívá učení na částečně označených datech. V kontextu detekce anomálií se může jednat o data, u kterých jsou označeny pouze normální hodnoty. Systém se má naučit, jaká data jsou normální a označit v nových datech anomální data, jako ta, která jsou od normálu odlišná. [4]

2 ISOLATION FOREST

Algoritmus Isolation Forest (dále jen IF) byl poprvé zveřejněn v roce 2008 [7], stejní autoři poté představili v roce 2012 IF, který nepočítá s tvorbou částečného modelu ve fázi učení, ale izoluje všechna data. [8] V této práci je nejprve představena původní varianta, následně je vysvětlen rozdíl mezi oběma variantami. V praktické části práce je pro implementaci a vizualizaci použito původní verze.

IF je tradiční metoda strojového učení, tzn. nevyužívá neuronové sítě. [4] Pracuje na principu strojového učení bez učitele a přistupuje k detekci anomálií odlišným způsobem než většina metod. Zatímco běžně se nejdříve vytvoří profil normálních dat a na základě této znalosti se potom identifikují anomálie jako odlišná data, IF přímo izoluje anomálie bez nutnosti analýzy všech dat. S využitím náhodných vzorků původních dat je vytvořena sada binárních rozhodovacích stromů (autory zvaných *iTree*), čímž vznikne les IF (*iForest*). Následně jsou do těchto stromů vložena všechna data a vyhodnocením průměrné hloubky jednotlivých prvků dat v binárních stromech se určuje, zda se jedná o anomálie. Algoritmus je lineárně závislý na čase a má nízké nároky na paměť, je dobře škálovatelný i pro obrovské objemy dat. [7] Fungování IF je podrobně popsáno v následujících podkapitolách.

2.1 Princip izolace prvků u IF

Základní myšlenkou algoritmu IF je izolace prvků v rámci binárního stromu. Vychází se z předpokladu, že anomálií je málo a jsou odlišné, což je předurčuje k snadnější izolaci. Ve stromové struktuře jsou anomálie izolovány v menší hloubce, zatímco normální prvky ve větší.

IF vytváří soubor binárních rozhodovacích stromů, na základě dvou parametrů, kterými je počet stromů a velikost vzorku původních dat – náhodný výběr prvků bez náhrady. Dobrých výsledků analýzy je dosahováno při velikosti vzorku $2^8 = 256$ a počtu stromů 100, při větších hodnotách těchto parametrů se již výsledky výrazně nezlepšují.

Výhodou je, že IF nemusí projít všechna data, tedy izolovat veškeré prvky, ale stačí stromy vygenerovat do určité hloubky – vytvořit částečný model. Díky tomu velká část dat nemusí být vůbec zpracována. [7]

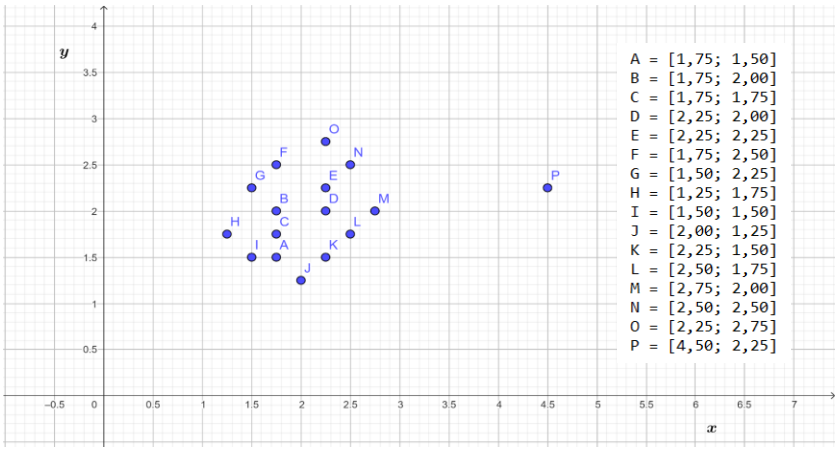
2.1.1 Izolace a iTree

Izolovat prvek dat znamená separovat ho od ostatních prvků. V binárním stromu jsou data rekurzivně dělena podle náhodně zvolené veličiny (parametru) a její náhodně zvolené hodnoty. V dané množině dat se volí hodnota pouze mezi minimální a maximální hodnotou zvolené veličiny. Rekurze dělení dat je zastavena, pokud nastane jedna ze tří možností: všechna data v uzlu mají stejnou hodnotu, je dosaženo maximální hloubky stromu, velikost dat je 1. V posledním jmenovaném případě dochází k izolaci. Vytvořený strom je *proper binary tree*, každý uzel má přesně nula (externí uzel) nebo dva (interní uzel) potomky. Protože izolovaný uzel nemá další potomky, jedná se o externí uzel. Interní uzel obsahuje test a dva potomky – levý a pravý uzel. Test je informace o zvoleném parametru a hodnotě, podle kterých se data dále dělí. Bylo zjištěno, že anomálie mají zřetelně kratší cestu od kořene stromu (jsou méně hluboko) než normální prvky. Je to způsobeno tím, že jsou osamocenější, a tudíž je k jejich izolaci potřeba menšího počtu dělení dat. Právě proto existuje předpoklad, že postačí vytvoření stromů pouze do určité hloubky, protože anomální body jsou izolovány dříve. [7]

Výše popsaným způsobem se vybuduje sada stromů, pokaždé pro jiný vzorek dat. Každý takový strom má odlišné testy v uzlech, protože tyto hodnoty jsou voleny náhodně. Pokud IF (sada stromů) v průměru vykazuje nižší hodnoty pro hloubku určitých dat ve stromu, s vysokou pravděpodobností se jedná o anomálie.

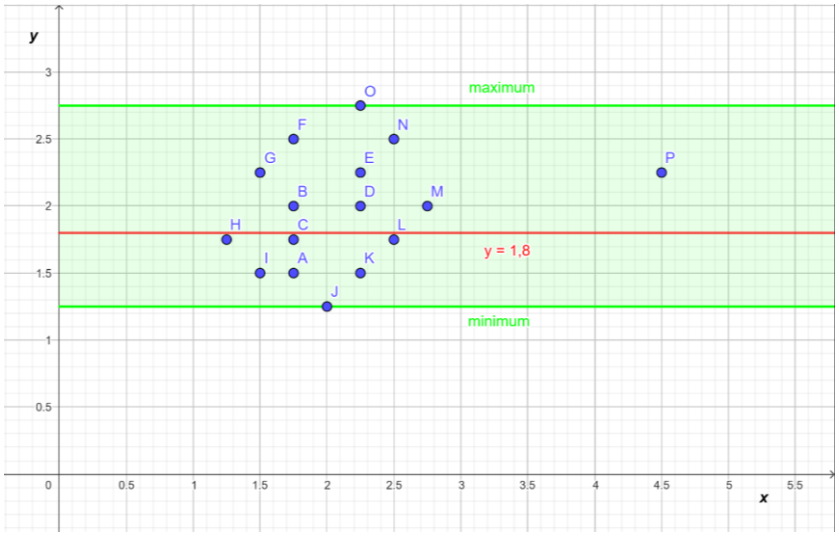
2.1.2 Příklad izolace prvků

Celý princip izolace je demonstrován na následujícím zjednodušeném příkladu. Bylo zvoleno 16 bodů A až P rozmístěných tak, že patnáct bodů A až O jsou normální data v jednom shluku a bod P představuje anomálii, viz obrázek 1.



Obrázek 1 Izolace dat – zvolené body

Izolace u IF probíhá postupným dělením množiny dat podle náhodně zvolené veličiny a její náhodné hodnoty do podmnožin až dojde k izolaci každého prvku dat. Veličiny v tomto zjednodušeném příkladu představují osy *x* a *y*, pro přehlednost budou tyto dvě veličiny v následujících krocích střídány. V reálné náhodné situaci je samozřejmě možné, že se data dělí opakovaně podle stejné veličiny. Na obrázku 2 je vidět první dělení dat podle veličiny *y*. Zeleně zvýrazněná část představuje minimální a maximální hodnoty, ze kterých se pak vybírá hodnota k dělení dat. Zde je minimum určeno souřadnicí *y* bodu J a maximum souřadnicí *y* bodu O, tedy náhodná hodnota *y* musí být v intervalu (1,25; 2,75). Pro demonstraci je zvolena hodnota $y = 1{,}8$ ve střední třetině intervalu. Na obrázku 2 zobrazeno červenou čarou.

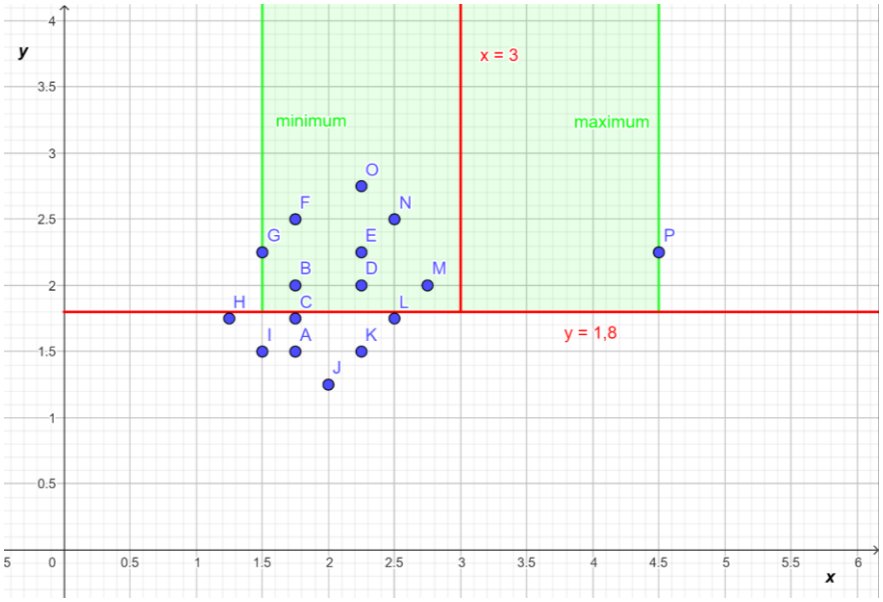


Obrázek 2 Izolace dat – první dělení dat

Jak je z obrázku 2 patrné, body jsou červenou čarou ($y = 1{,}8$) rozděleny na dvě části. V binárním stromu IF je toto dělení dat z kořene stromu provedeno pomocí testu, tj. podmínek zvolené veličiny. Data se souřadnicí $y < 1{,}8$ jsou zařazena do levého potomka, data se souřadnicí $y \geq 1{,}8$ do pravého potomka. Levý potomek tedy získá podmnožinu bodů {A, C, H, I, J, K, L} a pravý potomek podmnožinu {B, D, E, F, G, M, N, O, P}. Takto se dále rekurzivně pokračuje, dokud nejsou izolovány všechny prvky dat, tj. dokud data nejsou dále dělitelná.

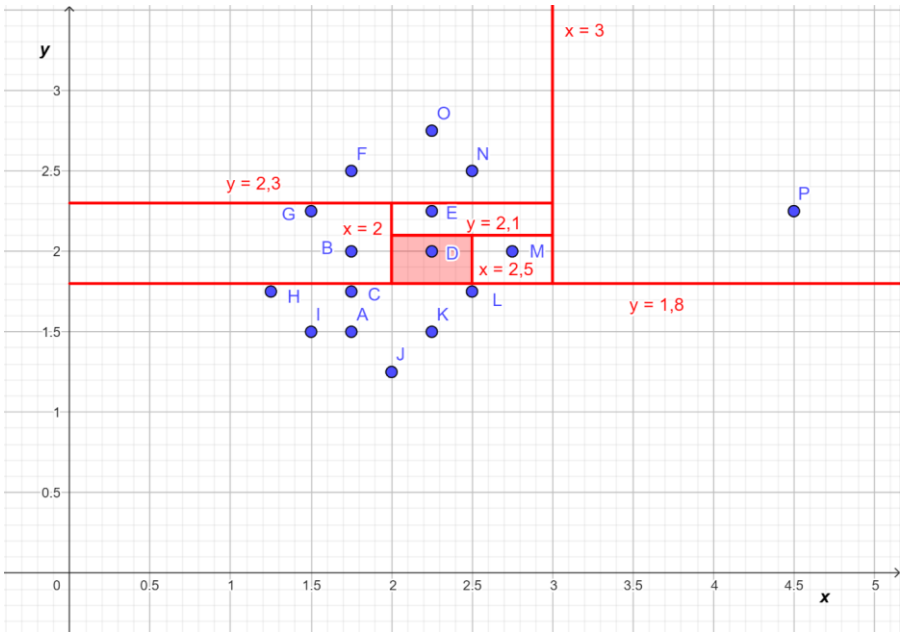
Obrázek 3 znázorňuje brzkou izolaci anomálního bodu P, již druhým dělením. Jedná se o dělení dat v pravém potomkovi kořene stromu. Zde je zvolena jako veličina pro dělení osa *x*, interval mezi minimální a maximální hodnotou, určený body G a P, je (1,5; 4,5) a v grafu je opět vyznačen zelenou plochou. Pro dělení dat byla zvolena hodnota $x = 3$. Levý potomek uzlu obdrží data, kde $x < 3$, tedy množinu {B, D, E, F, G, M, N, O}, pravý potomek obdrží

data, kde $x \geq 3$, tedy samotný bod P. Tímto je bod P izolován, vzniká externí uzel stromu IF o velikosti dat 1, který se už dál nedělí. K izolaci bodu P bylo potřeba 2 dělení dat, jeho hloubka v binárním stromu IF je tedy 2.



Obrázek 3 Izolace dat – izolace anomálního bodu P

Postupná izolace pro normální bod D je vidět na obrázku 4. Výsek s izolovaným bodem je zvýrazněn červenou barvou.



Obrázek 4 Izolace dat – izolace normálního bodu D

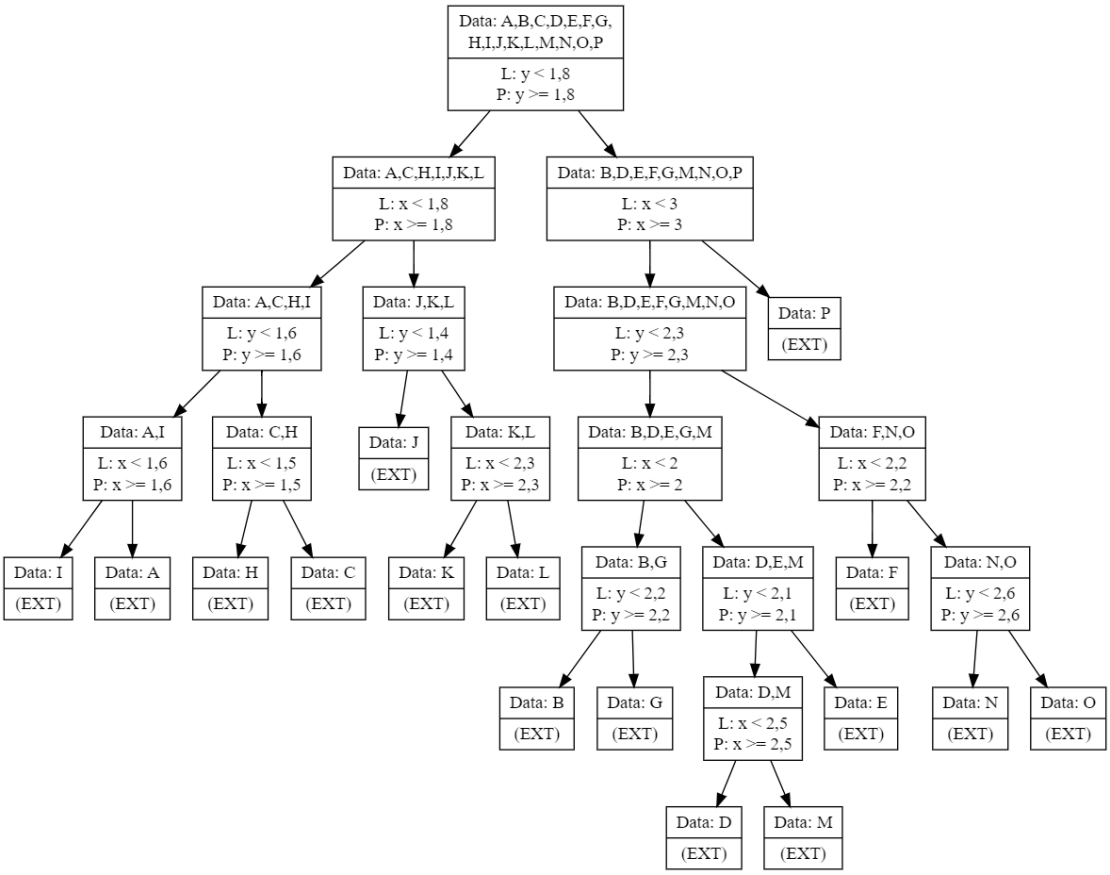
Jelikož se jedná o bod uvnitř shluku normálních bodů, jeho izolace vyžadovala 6 dělení dat na základě následujících podmínek:

$y \geq 1{,}8$
$x < 3$
$y < 2{,}3$
$x \geq 2$
$y < 2{,}1$
$x < 2{,}5$

Každá podmínka představuje hranu v binárním stromu. Všechny podmínky společně určují cestu od kořene stromu k externím uzlu izolovaného bodu D.

Izolace všech šestnácti bodů, včetně výše podrobněji popsaných bodů D a P, je na obrázku 5, kde je zobrazen celý binární rozhodovací strom. Každý uzel obsahuje informaci o bodech v něm obsažených a o rozhodovací podmínce pro dělení dat do jeho levého a pravého potomka. Anomální bod P je v hloubce 2, normální body D a M jsou v hloubce stromu 6, což je nejvyšší hloubka dosažená v tomto stromu. Ostatní normální body jsou izolovány v různých hloubkách, nejvíce bodů, a to sedm v hloubce 4. Takovýto strom je *proper binary tree*, jehož uzly mají buď žádného nebo přesně dva potomky. Strom má n = 16 externích uzlů (pro každý izolovaný prvek dat), n – 1 = 15 interních uzlů a celkový počet uzlů 2n – 1 = 31.

Obdobná vizualizace stromu jako na obrázku 5 je úkolem praktické části práce. Je nutno zdůraznit, že na rozdíl od zde uvedeného jednoho stromu pro dělení dat, algoritmus IF vytváří velké množství stromů, tj. les (*iForest*) a vyhodnocení anomálií probíhá na základě průměrných dosažených hodnot ze všech stromů. V reálném světě je také analyzovaných dat velké množství a mají mnoho veličin. Zatímco znázornění jako na obrázcích 1 až 4 v případě mnoha veličin není možné (jedná se o multidimenzionální problém), vizualizace stromu je velmi podobná, jako na obrázku 5. Každý uzel může obsahovat informaci o veličině a její hodnotě, podle které se data dělí, bez ohledu na množství veličin. Na obrázku 5 jsou v uzlech z důvodu lepší demonstrace dělení dat také uvedené body, kterých se dělení týká, což v reálném příkladu není možné z důvodu velkého počtu ani to není nutné pro analýzu.



Obrázek 5 Izolace dat – binární rozhodovací strom IF

2.2 Fáze učení

Ve fázi učení dochází k tvorbě sady rozhodovacích binárních stromů, tedy k vytvoření lesa *iForest*. Vstupními parametry pro vytvoření lesa jsou počet stromů, velikost vzorku dat a samotná data, která mají být analyzována. Systém vypočítá limit pro maximální hloubku stromu. Tato hodnota je volena jako průměrná hloubka na základě velikosti vzorku dat. Postupně se vytvoří žádaný počet binárních stromů. Pro každý strom je použito jiného náhodného vzorku dat. Pro vstupní parametry jsou doporučovány hodnoty 100 stromů a vzorek dat o velikosti 256. [7]

Samotný strom *iTree* přijímá 3 vstupní parametry, kterými jsou vzorek dat, aktuální hloubka stromu (vkládá se počáteční hloubka – hodnota 0) a maximální hloubka stromu (limit). Jedná se o rekurzivní algoritmus, kdy ukončení rekurze nastane, pokud je velikost dat v uzlu menší nebo rovna jedné, nebo pokud je dosaženo limitu maximální hloubky stromu. V takovém případě je navrácen externí uzel s hodnotou velikosti dat v uzlu. Za zmínku stojí, že pokud je velikost dat 1 došlo ke skutečné izolaci prvku v externím uzlu, ale pokud je velikost dat

větší než 1, došlo k ukončení rekurze dosažením maximální hloubky. Pokud počáteční podmínky pro ukončení rekurze nejsou splněny, funkce vybere náhodně veličinu, podle které se budou data dělit. Po výběru veličiny dojde k náhodnému výběru jedné její hodnoty mezi její minimální a maximální hodnotou. Data jsou rozdělena na dvě části a funkce rekurzivně volá sama sebe pro levého a pravého potomka. Levý potomek obdrží data, která mají u zvolené veličiny menší hodnotu a pravý potomek obdrží data, která mají větší nebo stejnou hodnotu, než je náhodně zvolená hodnota pro dělení dat. Rekurzivnímu volání je přiřazena příslušná část dat, aktuální hloubka stromu zvýšená o 1 a maximální hloubka stromu, která se nemění oproti původnímu stromu. Funkce vrací interní uzel s informací o dělící veličině a její hodnotě a s přiřazeným levým a pravým potomkem. [7]

Výstupem fáze učení je sada binárních stromů, která je připravená pro použití ve vyhodnocovací fázi.

2.3 Fáze vyhodnocení

V této fázi je postupováno tak, že všechna původní data jsou postupně vložena do stromů IF a je počítána hodnota jejich hloubky ve stromu *h(x)*. Prvek dat postupuje stromem a je řazen do dalších uzlů (levých nebo pravých potomků) na základě srovnání svých hodnot s testem v daném interním uzlu až eventuelně skončí v externím uzlu. Ke hloubce externího uzlu se se na základě velikosti dat v něm přičte hodnota funkce c(ψ):

$c(\psi) = 2H(\psi-1) - 2(\psi-1)/n$ pro $\psi > 2$

$c(\psi) = 1$ pro $\psi = 2$

$c(\psi) = 0$ pro jiné hodnoty ψ

kde funkci *H(i)* lze aproximovat $H(i) \cong ln(i) + 0.5772156649$, ψ je velikost dat v uzlu a *n* je celková velikost dat. [8]

Pro každý prvek dat jsou hodnoty *h(x)* upravené hodnou funkce *c(ψ)* použity pro výpočet jeho průměrné hloubky *E(h(x))* ze všech stromů IF. Z této hodnoty E(h(x)) se poté vypočítá skóre anomálie *s*:

$$s(x, \psi) = 2^{-\frac{E(h(x))}{c(\psi)}}$$

kde *x* je prvek dat, ψ je velikost vzorku dat.

Skóre anomálie je v intervalu $0 < s \leq 1$, data s nejvyšším skóre *s* jsou nejvíce anomální.

2.4 Isolation Forest s úplnou izolací

Jak již bylo zmíněno, existuje i novější varianta IF, která nepočítá s předčasným ukončením rekurze ve fázi učení. Původní autoři IF představili upravené algoritmy v odborném článku [8]. Při tvorbě binárních stromů IF se zde rekurze ukončuje pouze v momentě, kdy už jsou data dále nedělitelná, dochází k úplné izolaci vzorků dat. Tato metoda odpovídá demonstraci v kapitole 2.1.2 této práce. Ve vyhodnocovací fázi je potom využito limitní hloubky stromu, jehož zvolená hodnota může ovlivnit kvalitu výsledků při detekci shluků anomálií s různou hustotou. Pro běžné použití je zde doporučeno použití maximální možné hodnoty: $\psi - 1$, kde ψ je velikost vzorku dat. Pro implementaci IF a vizualizaci binárních stromů v praktické části práce bude využito původních algoritmů pro *iTree* a *iForest*, tedy těch, které vytvoří částečný model do vypočítané maximální hloubky. Tento limit je volen jako průměrná hodnota hloubky stromu určité velikosti, na základě velikosti vzorku dat zadané uživatelem.

3 POUŽITÉ TECHNOLOGIE A PRACOVNÍ PROSTŘEDÍ

Pro implementaci algoritmu IF byl zvolen programovací jazyk JavaScript a pro vizualizaci binárních stromů software *Graphviz*. K testování bude použit framework Jest. Celá knihovna pro implementaci IF a vizualizaci stromů (dále jen knihovna) bude vytvořena jako balíček *npm*. V následujících podkapitolách jsou všechny zmíněné technologie stručně popsány.

3.1 Javascript

Javascript vznikl v roce 1995 jako jazyk pro web a původně sloužil hlavně pro kontrolu formulářů na straně uživatele bez nutností komunikace se serverem. Během let se z něho stal plnohodnotný jazyk, který je schopen komplexních výpočtů, rekurze i objektového programování. [9][10] V rámci webu tvoří společně s CSS a HTML základní stavební kameny internetových stránek, kde zajišťuje interaktivitu a vylepšuje uživatelský zážitek. [11] V dnešní době však JavaScript není vázán pouze na klientskou stranu webu, ale lze s ním tvořit i serverové a desktopové aplikace.

Jedná se o interpretovaný jazyk, na rozdíl od mnoha jiných jazyků není kompilovaný a je překládán „za běhu“. [9] Každý moderní internetový prohlížeč umí JavaScript interpretovat, což ho předurčuje k širokému použití od začátečníků až po profesionální programátory. K začátku programování stačí jakýkoliv textový editor a prohlížeč. [12] V prohlížeči lze také otevřít prostředí pro vývojáře (např. *DevTools* v Google Chrome), které umožňuje analyzovat a ladit kód, zobrazit chyby, pracovat s konzolí atd. [9]

V poslední době se díky *Node.js* neomezuje využití JavaScriptu pouze na prohlížeče a webové aplikace. [13] Zatímco i velmi jednoduchý kód JavaScriptu potřebuje pro spuštění v prohlížeči být propojen se souborem v jazyku HTML, *Node.js* poskytuje prostředí umožňující běh JavaScriptu samostatně. Po nainstalování *Node.js* pomocí instalačního balíčku z webu *nodejs.org* je možno přímo v terminálu operačního systému spustit JavaScriptový soubor. [9]

```
node index.js
```

Výše uvedený příkaz například spustí soubor „index.js“ v pracovním adresáři. Výstupy pomocí příkazu `console.log()` v Javascriptu, které se při použití ve webové aplikaci v prohlížeči zobrazují v konzoli nástrojů pro vývojáře, se zde zobrazí přímo v konzoli operačního systému.

V této práci bude jazyka JavaScript využito pro vytvoření zdrojových kódů knihovny a propojení s *npm* balíčkem *Graphviz* pro vizualizaci.

3.2 Technologie npm

Npm package manager (běžně je používána pouze zkratka *npm*) je softwarový registr využívaný open source vývojáři pro sdílení a vyhledávání balíčků a také společnostmi pro privátní správu projektů. [14] Pro ovládání *npm* se většinou využívá příkazový řádek, web *npm* slouží pro vytvoření uživatelského účtu a vyhledávání v databázi balíčků. Pro účely praktické části této práce je *npm* technologie využitá dvěma způsoby. Jedním z nich je využití *npm* balíčků, např. *Graphviz*, *lodash* a *jest*, druhým je vytvoření nového *npm* balíčku v podobě knihovny pro vizualizaci IF. Balíček *npm* může být sdílen veřejně (*public*) nebo neveřejně (*private*). První z možností je zdarma a postačí pro ni mít vytvořený účet na webu *npm*. Takto bude sdílena i knihovna vytvořená v praktické části práce.

Využívat *npm* lze přímo po instalaci *Node.js*, protože je *npm* je její součástí. Po výběru balíčku v registru jej lze v adresáři projektu snadno nainstalovat.

```
npm install <name>
```

V souboru s JavaScript kódem je pak nutno pomocí příkazu `require`, tento balíček naimportovat.

```
const <name> = require('<name>');
```

V uvedených příkladech `<name>` zastupuje jméno balíčku. Po provedení popsaných kroků lze využívat funkcionalitu zvoleného balíčku.

3.3 Software Graphviz

Opensource software *Graphviz* slouží pro vizualizaci grafů, což je způsob, jak zobrazit strukturovanou informaci jako schéma. Využití nachází např. v softwarovém inženýrství, databázích, web designu a strojovém učení. *Graphviz* přebírá popis grafu v jednoduchém textovém formátu, který převádí na schémata v užitečných formátech jako *png*, *jpeg*, *svg* a *pdf*. Přitom může uživatel zvolit, jak se graf bude zobrazovat, tvary, barvy, styly čar, aj. [15] Instalace *Graphviz* se provádí pomocí instalačního balíčku na webu *graphviz.org* dle příslušného operačního systému.

3.3.1 Jazyk DOT

Jazyk DOT obsahuje gramatiku pro popis grafů, které jsou následně vygenerovány pomocí softwaru *Graphviz*. Graf musí být označen jako *graph* nebo *digraph*, kde *graph* obsahuje hrany bez určení směru, zatímco *digraph* obsahuje hrany s určeným směrem závislosti mezi uzly.

Hrany bez určení směru:

```
A – B
```

Hrany s určením směru:

```
A -> B
```

Jednoduchý příklad grafu v podobě binárního stromu je uveden níže. Jedná se o graf *digraph* s určením směru mezi uzly, uzly jsou pojmenovány pomocí velkých písmen a vztah mezi nimi je vyjádřen šipkami. Středníky na konci řádku nejsou vyžadovány, ale zvyšují čitelnost. [16] Celý kód v jazyku DOT je uložen do souboru „input.dot“.

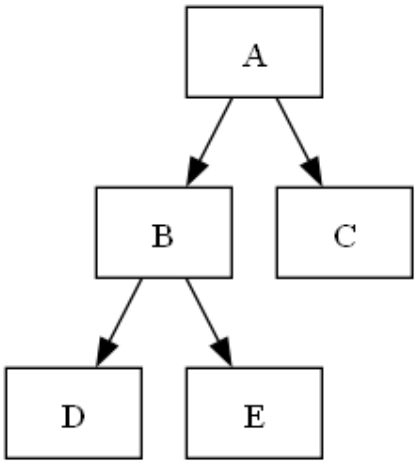
Soubor „input.dot“:

```
digraph
{
    A -> B;
    A -> C;
    B -> D;
    B -> E;
}
```

Pro vygenerování grafu byl použit příkaz uvedený níže, předpokladem je že soubor „input.dot“ se nachází v příslušném adresáři. Příkaz specifikuje výstupní formát *png*, název souboru v DOT jazyku „input.dot“ a název výstupního souboru „graph.png“. Dále je určen tvar uzlů grafu jako obdélník. [16][17] Graf se uloží do pracovního adresáře, výsledek je na obrázku 6.

Příkaz pro uložení grafu do zvoleného formátu:

```
dot -Tpng input.dot -o graph.png -Nshape=rect
```



Obrázek 6 Graf vygenerovaný do souboru „graph.png“

Výše uvedený způsob tvorby grafů je vhodný pro demonstraci, popř. pro malý počet grafů. Běžně se využívá možnosti vytvoření řetězce v podobě DOT jazyka pomocí kódu. Tato metoda bude použita i v praktické části práce.

3.3.2 Balíček npm Graphviz

Pro zajištění exportu vizualizovaných grafů bude použito softwaru *Graphviz* v podobě *npm* balíčku. Jedná se o populární balíček, který dosahuje přibližně 300 tisíc stažení týdně. Základní funkcionalita je zřejmá z ukázky kódu v popisu balíčku na serveru *npmjs.com* [18], export je funkční do všech běžně používaných formátů. Podmínkou pro použití je instalace samotného softwaru *Graphviz*. Instalace *npm Graphviz* se v adresáři projektu provede příkazem:

```
npm install graphviz
```

Níže je uveden příklad *JavaScript* kódu, jehož výsledkem bude stejný graf (binární strom) jako na obrázku 6. Nejprve je importována funkcionalita *Graphviz* a deklarován nový graf *g*. Pomocí metod `addNode()` a `addEdge()` jsou vytvořeny uzly a hrany mezi nimi. Uzlům je nadefinován tvar obdélníku. Graf je vyexportován do formátu *png* metodou `output()`.

```
let graphviz = require('graphviz');

var g = graphviz.digraph("G");

g.addNode("A", {"shape" : "rect"});
g.addNode("B", {"shape" : "rect"});
g.addNode("C", {"shape" : "rect"});
g.addNode("D", {"shape" : "rect"});
g.addNode("E", {"shape" : "rect"});
```

```
g.addEdge("A", "B");
g.addEdge("A", "C");
g.addEdge("B", "D");
g.addEdge("B", "E");

g.output("png", "graph.png");
```

3.4 Testovací framework Jest

Dynamické jazyky podporují testovací frameworky, které výrazně usnadňují práci při psaní testů. [13] V praktické části práce bude pro testování použit *Jest*, což je framework vhodný pro Javascript, který se zaměřuje na jednoduchost instalace i použití. [19]

Pro instalaci *Jest* je možno využít příkaz níže, využívající *npm*.

```
npm install --save-dev jest
```

Volba `--save-dev` znamená, že se *Jest* instaluje pouze pro vývojové účely (*dev dependency*). [13]

Dále je nutno v souboru „package.json“ doplnit sekci kódu znázorněnou níže červeně, čímž je *Jest* připraven k využití. [20]

Soubor „package.json“:

```
{
  "devDependencies": {
    "jest": "^29.7.0"
  },
  "scripts": {
    "test": "jest"
  }
}
```

3.4.1 Unit testy u funkcí

Unit testy testují základní funkcionalitu jednotlivých funkcí, bez toho, aby zjišťovali, jak funkce kooperují mezi sebou a kód funguje jako celek. [21] Využití Jest pro unit testy je v této kapitole demonstrováno na jednoduchém kódu se dvěma funkcemi v souboru „testedCode.js“. Kód obsahuje funkci `multiply()` pro násobení a `subtract()` pro odčítání. Obě funkce jsou na konci kódu vyexportovány.

Soubor „testedCode.js“:

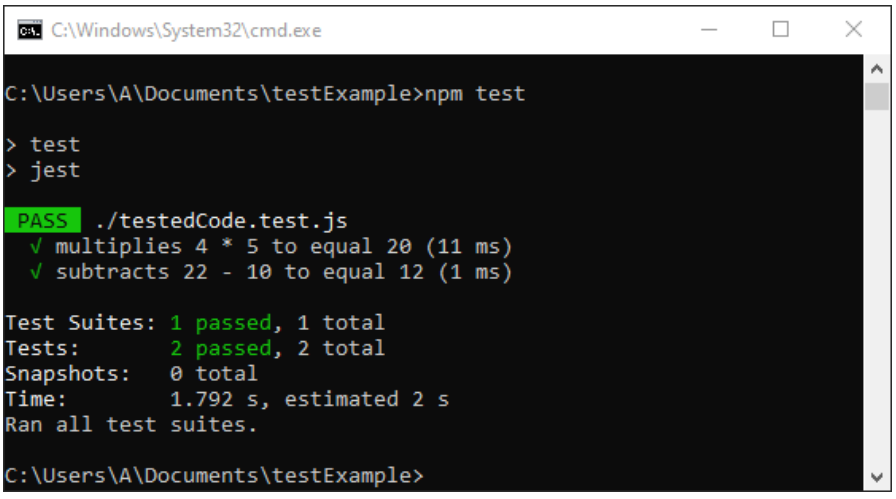
```
function multiply(a, b) {
    return a * b;
}
function subtract(x, y){
    return x - y;
}
module.exports = {multiply, subtract};
```

Na každou funkci je napsán jeden test a kód je uložen do vlastního JavaScript souboru nazvaného „testedCode.test.js“. Na začátku jsou importovány funkce z příslušného souboru s testovaným kódem. Samotný test vždy obsahuje text s popisem jeho funkce, v části `expect` je volání funkce s příslušnými vstupy a v části `toBe` očekávaný výstup. [20]

Soubor „testedCode.test.js“:

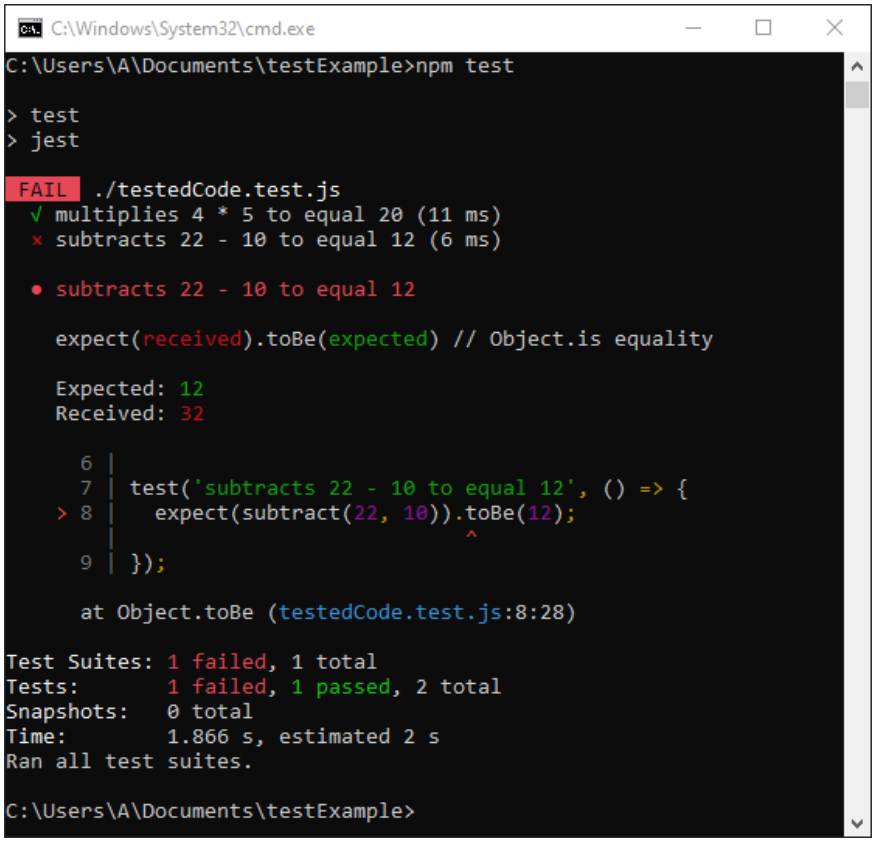
```
const {multiply, subtract} = require("./testedCode");

test("multiplies 4 * 5 to equal 20", () => {
  expect(multiply(4, 5)).toBe(20);
});

test("subtracts 22 - 10 to equal 12", () => {
  expect(subtract(22, 10)).toBe(12);
});
```

Test se provede příkazem `npm test` v příkazové řádce. Výsledek v případě úspěšných testů je na obrázku 7.



Obrázek 7 Výstup frameworku *Jest* po úspěšných testech

V případě, že některý test selže, podává *Jest* podrobnou zpětnou vazbu. Pokud je u funkce `subtract()` změněno znaménko v návratové hodnotě z mínus na plus, příslušný test selže. Výstup je na obrázku 8.



Obrázek 8 Výstup frameworku *Jest* po selhání testu

3.4.2 Testování tříd

Testování metod tříd pomocí unit testů probíhá podobně jako testování funkcí. Pro seskupení částí testovacího kódu lze využít bloků `describe`, což kód zpřehledňuje a také udržuje působnost pomocných funkcí v rámci těchto bloků. V rámci bloku `describe` je výhodné použít příkazy `beforeEach` a `beforeAll`, pro vykonání určitého kódu před každým testem, nebo jednou před všemi testy. Takto lze například vytvářet nové instance třídy nebo volat pomocné funkce. [21]

Příklad kódu pro testování třídy `Calculator` a jejích dvou metod `add()` a `subtract()`. Před každým testem je pomocí `beforeEach` vytvořen nový objekt:

```
const Calculator = require("./testedClass");

describe("Calculator", () => {
  let calc;
```

```
  beforeEach(() => {
    calc = new Calculator();
  });

  test("adds 1 + 2 to equal 3", () => {
    expect(calc.add(1, 2)).toBe(3);
  });

  test("subtracts 5 - 3 to equal 2", () => {
    expect(calc.subtract(5, 3)).toBe(2);
  });
});
```

3.5 Pracovní prostředí

Text práce byl napsán na operačním systému Windows 10 v aplikaci MS Office Word s využitím oficiální šablony pro závěrečné práce FAI UTB ve Zlíně. Pro tvorbu *npm* knihovny v praktické části práce bylo použito editoru *Visual Studio Code*, který je dostupný pro všechny běžně používané operační systémy. Editor se ukázal být dobrou volbou, zobrazení syntaxe JavaScriptu je přehledné a ovládání je intuitivní. Pro průběžné spouštění kódu při jeho tvorbě bylo využíváno příkazového řádku v konzoli Windows aplikace *cmd.exe* (CLI) s využitím technologie *Node.js*.

PRAKTICKÁ ČÁST

4 NÁVRH KNIHOVNY

Návrhem a implementací softwarových produktů se zabývá softwarové inženýrství. Jeho úkolem je stanovení jednoznačných požadavků a tvorba odpovídajícího softwaru. Požadavky mají specifikovat co je a není od softwaru očekáváno. Plánování projektu začíná sběrem uživatelských požadavků od zákazníka. Ty určují, co má systém dělat, ne však to, jak to má dělat. Jedná se o výroky formulované přirozeným jazykem, které mohou být doprovázeny neformálními diagramy. Práce s požadavky, tedy jejich zjišťování, analýza, ověřování, validace a správa, se nazývá inženýrství požadavků (*requirement engineering*). [22][23]

Na základě uživatelských požadavků se dále specifikují systémové požadavky, určující jaké vlastnosti má budoucí systém mít. Lze je rozdělit na funkční a nefunkční. [22]

Funkční požadavky popisují, jaké služby má systém poskytovat a jak má reagovat na zadané vstupy. Jedná se tedy o popis funkcionality systému. Dále také musí určovat co by systém provádět neměl. Funkční požadavky by měly být detailní, určovat vstupy, výstupy, výjimky apod.

Nefunkční požadavky charakterizují, jak se systém chová vzhledem k vlastnostem jako je bezpečnost, spolehlivost, dostupnost, znovupoužitelnost, údržba, výkon apod. Význam nefunkčních požadavků roste v okamžiku, kdy se na trhu nacházejí produkty se stejnou funkcionalitou, ale odlišují se jinými vlastnostmi. [23]

4.1 Analýza požadavků na knihovnu

Uživatelské požadavky na knihovnu byly určeny na základě zadání bakalářské práce a konzultací s vedoucím práce. Základním výstupem očekávaným od knihovny je grafická reprezentace jednotlivých stromů IF. Předpokládaným uživatelem je profesionál zabývající se strojovým učením a analýzou dat, který je již seznámen s teorií IF. Může se také ale jednat o profesionála, studenta, či zájemce z jiného oboru, který pomocí této knihovny lépe pochopí princip IF. Z uživatelských požadavků byly specifikovány systémové funkční a nefunkční požadavky.

Tabulka 1 Uživatelské požadavky

Ozn.	**Požadavek**
UP-1	Uživatel získá na základě vložených dat vizualizaci jednoho stromu nebo všech stromů IF do požadovaného výstupního formátu, vstupní data mohou mít libovolnou dimenzi.
UP-2	Uživatel si může nechat zobrazit základní informace o vyhodnocení IF vzhledem k detekci anomálií.
UP-3	Uživatel je informován o provedených akcích systému zpětnou vazbou
UP-4	Uživatel má možnost knihovnu využít na různých platformách.
UP-5	Uživatel má k dispozici přehlednou dokumentaci s příklady použití.
UP-6	Uživatel dostává v případě nutnosti zpětnou vazbu chybovými hláškami.

Tabulka 2 Funkční požadavky na systém

Ozn.	**Založeno na UP**	**Požadavek**
FP-1	UP-1	Systém na základě vstupu (data, počet stromů, velikost vzorku dat) vytvoří IF – pole stromů.
FP-2	UP-1	Systém přijímá data ve formě pole (*array*) libovolné dimenze.
FP-3	UP-1	Systém na základě vstupu (index stromu, výstupní formát, název souboru) vygeneruje soubor s vizualizací stromu.
FP-4	UP-1	Systém na základě vstupu od uživatele (výstupní formát, název souboru) vygeneruje vizualizace všech stromů v IF, soubory jsou očíslovány podle příslušných indexů stromů.
FP-5	UP-1	Systém podporuje různé výstupní formáty pro vizualizaci.
FP-6	UP-2	Systém zobrazí informaci o průměrné hloubce prvků dat v IF.
FP-7	UP-2	Systém zobrazí informaci o skóre anomálie prvků dat.
FP-8	UP-3	Systém informuje uživatele o vytvoření IF
FP-9	UP-3	Systém informuje uživatele o exportu stromu, popř. celého IF
FP-10	UP-4	Systém lze nainstalovat a používat na různých platformách.
FP-11	UP-5	K systému náleží přehledná a podrobná dokumentace včetně příkladů použití.
FP-12	UP-6	Systém na nesprávné vstupy či způsob použití reaguje chybovými hláškami s informativním popisem problému/chyby.

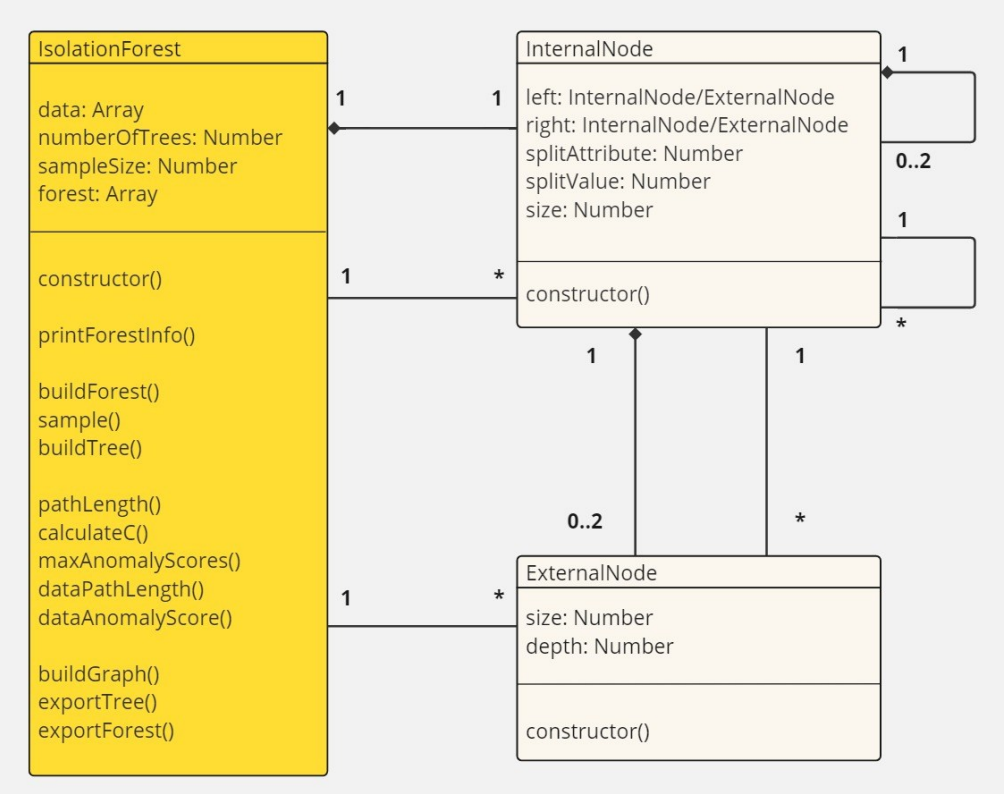
Tabulka 3 Nefunkční požadavky na systém

Ozn.	Požadavek
NP-1	Výkon: Systém zpracuje i velká data.
NP-2	Výkon: Systém spolehlivě vizualizuje strom do velikosti 256 prvků (typicky používaná velikost vzorku dat IF).
NP-3	Přenositelnost: Systém je otestován na různých operačních systémech
NP-4	Použitelnost: Systém je jednoduchý, přehledný a snadno použitelný.
NP-5	Spolehlivost: Systém je důkladně otestován.
NP-6	Údržba: Systém je přehledně zpracován pro zajištění snadné údržby

4.2 UML diagram tříd

Rozhodnutí vytvořit knihovnu jako *npm* balíček odpovídá požadavkům na přenositelnost. Kód v jazyce JavaScript je navržen s využitím principů OOP. Budou vytvořeny tři třídy: `ExternalNode`, `InternalNode` pro uzly stromů a `IsolationForest` využívající předchozích tříd a obsahující metody pro vytvoření IF, analýzu anomálií a vizualizaci. UML diagram tříd na obrázku 9 zobrazuje vztahy mezi třídami, jejich proměnné a metody.

Mezi třídou `IsolationForest` a `InternalNode` je vztah kompozice, kde `InternalNode` představuje kořenový uzel každého stromu IF. Tento uzel dále rekurzivně obsahuje ve svých proměnných `left` a `right` instance `InternalNode` nebo `ExternalNode`. Tyto vztahy jsou také kompozitní. Dále je v diagramu vyznačen vztah asociace mezi `IsolationForest` a `InternalNode` i `ExternalNode`, pro naznačení vztahu se všemi uzly stromu, které sice třída `IsolationForest` přímo nevlastní, ale využívá jejich existence pomocí svých metod pro vyhodnocení anomálií i vizualizaci. Obdobně je vyznačena asociace třídy `InternalNode` sama k sobě a ke třídě `ExternalNode`. Kardinality vyznačené v diagramu jsou uvažovány pro jeden strom. Každý vytvořený strom je uložen v `IsolationForest` do proměnné typu pole s názvem `forest`.



Obrázek 9 UML diagram tříd

4.3 Navržené metody na základě požadavků

V této kapitole jsou stručně popsány navržené metody třídy `IsolationForest` s informací, kterému funkčnímu požadavku daná metoda odpovídá.

`printForestInfo()` – výpis informace o vytvoření IF (FP-8)

`buildForest()` – vytvoření pole stromů `forest` (FP-1, FP-2)

`sample()` – navrací vzorek dat žádané velikosti

`buildTree()` – pomocná metoda pro vytvoření jednoho stromu IF (FP-1)

`pathLength()` – výpočet hloubky určitého prvku dat v jednom stromu

`calculateC()` – výpočet hodnoty funkce *c* pro detekci anomálií

`maxAnomalyScores()` – vyhodnocení a zobrazení anomálií

`dataPathLength()` – výpočet průměrné hloubky každého prvku dat v IF (FP-6)

`dataAnomalyScore()` – výpočet skóre anomálie pro každý prvek dat (FP-7)

`buildGraph()` – vytvoření grafické reprezentace stromu pomocí *npm Graphviz*

`exportTree()` – export jednoho stromu do požadovaného formátu (FP-3, FP-5, FP-9)

`exportForest()` – export všech stromů IF do požadovaného formátu (FP-4, FP-5, FP-9)

Tímto jsou pokryty funkční požadavky FP-1 až FP-9. Metody bez určení funkčních požadavků slouží k zajištění funkcionality vytvoření IF, analýzy anomálií a vizualizaci. Požadavek FP-10 na multiplatformní použitelnost je splněn pomocí technologie *npm*, požadavek FP-11 bude splněn tvorbou dokumentace a FP-12 při implementaci zavedením vhodných chybových hlášek.

K nefunkčním požadavkům bude přihlédnuto ve fázi implementace a testování.

5 IMPLEMENTACE

Implementace knihovny v jazyku JavaScript bude dosaženo pomocí tří tříd, z nichž jedna obsahuje metody pro veškerou funkcionalitu. Tato třída s názvem `IsolationForest` využívá pro vybudovaní jednotlivých stromů, tedy jejich interních a externích uzlů třídy `InternalNode` a `ExternalNode`. Využity jsou dva *npm* balíčky: *lodash* (např. metoda `_.sampleSize()`) a *Graphviz* pro export stromů. JavaScriptový kód v souboru „index.js“ je zkontrolován a upraven podle *JavaScript Standard Style* [24], který dohlédá na správná odsazení, pozici závorek, mezery a celkovou jednotnou úpravu kódu. Pro přehlednost však bylo zachováno použití středníků na konci řádků. Komentáře jsou provedeny ve stylu *JSDoc*. [25] Celá knihovna bude na závěr (viz kapitola 7) sama zveřejněna jako balíček *npm*.

5.1 Třídy pro uzly stromů

Základními stavebními kameny binárních stromů IF jsou interní a externí uzly tvořené třídami `InternalNode` a `ExternalNode`.

5.1.1 Třída InternalNode

Instance třídy `InternalNode` je kořenem každého stromu a obsahuje ve svých proměnných `left` a `right` další interní a externí uzly, čímž je rekurzivně vytvořena struktura stromu. Dalšími proměnnými této třídy jsou `splitAttribute`, `splitValue` a `size`, obsahující informace o náhodně zvolené veličině (atributu), dle které jsou data v uzlu dělena, její náhodně zvolené hodnotě a velikosti dat v daném uzlu. Tyto informace jsou využity pro tvorbu stromů, vyhodnocování anomálií i vizualizaci. Jedná se o jednoduchou třídu, jejíž proměnné jsou inicializovány konstruktorem, kromě kterého třída neobsahuje žádné další metody.

Implementace třídy `InternalNode`:

```
class InternalNode {
  constructor (left, right, splitAttribute, splitValue, size) {
    this.left = left;
    this.right = right;
    this.splitAttribute = splitAttribute;
    this.splitValue = splitValue;
    this.size = size;
  }
}
```

5.1.2 Třída ExternalNode

Třída určená pro externí uzly stromů (listy) obsahuje pouze proměnné pro velikost dat (`size`) a hloubku ve stromu (`depth`), které jsou využity při analýze anomálií a zobrazeny při vizualizaci. Třída obsahuje pouze konstruktor pro inicializaci proměnných.

Implementace třídy `ExternalNode`:

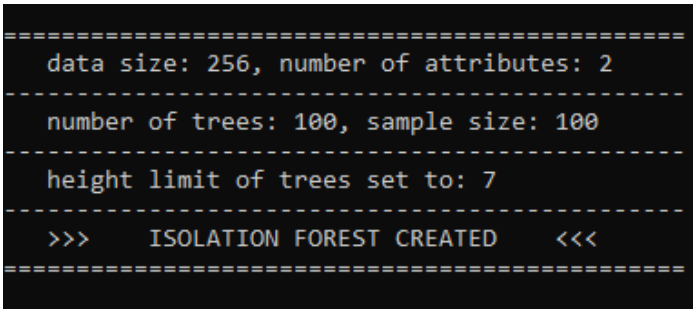
```
class ExternalNode {
  constructor (size, depth) {
    this.size = size;
    this.depth = depth;
  }
}
```

5.2 Třída IsolationForest

Tato třída obsahuje ve svých metodách funkcionalitu knihovny, při čemž využívá pro tvorbu stromů dříve zmíněných dvou tříd. Vstupy pro inicializaci třídy vycházejí z pseudokódu algoritmů pro IF [7], jedná se tedy o analyzovaná data, počet stromů a požadovanou velikost vzorku dat.

`constructor(data, numberOfTrees, sampleSize)`

Ze vstupů jsou vytvořeny stejnojmenné proměnné, další proměnná třídy pro ukládání stromů `forest` je inicializována v konstruktoru jako prázdné pole. Dále je vytvořena proměnná `heightLimit` a je jí přiřazena vypočtená hodnota na základě zvolené velikosti vzorku dat. Tato hodnota slouží pro ukončení rekurze při tvorbě stromů. Konstruktor volá vlastní metodu třídy `buildForest()` pro vytvoření IF a metodu `printForestInfo()` pro zobrazení základních informací uživateli, tedy údajů o vstupních datech, zvolené velikosti vzorku, počtu stromů a vypočítaném limitu maximální hloubky stromů. Více o metodě `buildForest()` v následující podkapitole, metoda `printForestInfo()` zde nebude dále popisována, výstup z ní je na obrázku 10.



Obrázek 10 Výstup do konzole o vytvoření IF z metody printForestInfo()

Konstruktor třídy obsahuje také ošetření výjimek na vstupu od uživatele. Vstupní data musí být ve formě neprázdného pole polí, kde všechny prvky dat (vnitřní pole) musí mít stejnou velikost. Dále musí být data unikátní, v rámci sady dat se tedy nesmí žádný prvek opakovat. Pro zjištění velikosti a unikátnosti dat jsou využity i metody `_.some()` a `_.uniqueWith()` z knihovny *lodash*. Počet stromů musí být celé kladné číslo, stejně jako velikost vzorku, která navíc nesmí být větší než velikost dat. Pokud uživatel zadá nesprávný vstup, je mu vrácena informativní chybová hláška.

Metody třídy `IsolationForest` lze podle jejich funkcionality logicky dělit do tří skupin, takto jsou i řazeny v samotném kódu a popsány v následujících podkapitolách s případným uvedením ukázek kódu.

5.2.1 Tvorba IF – fáze učení

Tato část kódu třídy `IsolationForest` obsahuje tři metody pro tvorbu stromů IF. Jedná se o metodu `buildForest()`, volanou konstruktorem třídy a dále metody `sample()` a `buildTree()`. Metoda `buildForest()` využívá další dvě zmíněné metody k vybudování stromů. Následuje stručný popis funkcionality těchto tří metod.

`buildForest()`

Obsahuje smyčku, která dle zadaného počtu stromů (`numberOfTrees`) pro každý strom zavolá vzorkovací funkci `sample()` ke zredukování dat na požadovanou velikost (`sampleSize`). S tímto vzorkem dat pak volá metodu `buildTree()` a navrácený nově vytvořený strom uloží do pole `forest`.

`sample(dataToSample, sizeOfSample)`

Navrací na základě vstupních dat náhodný vzorek o požadované velikosti. K tomuto účelu metoda používá funkci `_.sampleSize()` z knihovny *lodash*.

`buildTree(treeData, currentDepth = 0)`

Metoda vybuduje na základě vzorku dat strom pomocí vytváření instancí dříve zmíněných tříd `InternalNode` a `ExternalNode`. Tyto jsou ukládány do proměnných `left` a `right` instance třídy `InternalNode` na vyšší úrovni, čímž je tvořena struktura stromu. Tato metoda využívá principu IF popsaného v teoretické části práce. Na počátku je vybrán náhodný parametr dat, pomocí metod `_.minBy()` a `_.maxBy()` z knihovny *lodash* se zjistí jeho minimální a maximální hodnota v datech uzlu. Dále se zvolí v tomto intervalu náhodná hodnota, podle které se data rozdělí do levého a pravého potomka daného uzlu. Metoda volá rekurzivně

samu sebe, dokud nenastane situace, že jsou data dále nedělitelná (mají velikost 1) nebo je dosaženo limitu maximální hloubky `heightLimit`. V této části stromu tak vznikne externí uzel `ExternalNode`, do jehož proměnných jsou uloženy informace o velikosti dat a aktuální hloubce ve stromu. Metoda `buildTree()` obsahuje dva vstupní parametry, první pro vzorek dat a druhý pro aktuální hloubku stromu, která je defaultně nastavena na nulovou hodnotu. První volání metody z metody `buildForest()` obsahuje pouze jeden argument, zatímco další volání (rekurzivní) jsou provedena s aktualizovanou hodnotou hloubky ve stromu. Tímto je udržována hodnota hloubky a může být vložena při ukončení rekurze do externího uzlu stromu `ExternalNode`. Metoda také obsahuje `do-while` smyčku pro ošetření situace, kdy by všechna data v uzlu měla stejnou hodnotu náhodně zvolené veličiny. V takovém případě by minimální i maximální hodnota této veličiny byla stejná a data by nešla dělit, přestože podle jiné veličiny jsou data dále dělitelná. Pokud tato situace nastane, volí se ve smyčce `do-while` nová náhodná veličina, čímž eventuelně dojde k dalšímu rozdělení dat.

5.2.2 Detekce anomálií – fáze vyhodnocení

Kód pro fázi vyhodnocení se skládá z pěti metod, z nichž metody `pathLength()`, `calculateC()` a `maxAnomalySocres()` slouží pro pomocné výpočty, vyhodnocení a zobrazení informací. Jsou volány metodami, které uživatel knihovny využije, pokud chce zobrazit informaci o průměrné hloubce prvků dat ve stromech IF (`dataPathLength()`) a hodnotě skóre anomálie (`dataAnomalyScore()`).

```
pathLength(dataMember, iTree, currentPathLength = 0)
```

Vypočítá hloubku určitého prvku dat `dataMember` v jednom požadovaném stromu IF. Přestože se počítá hloubka ve stromu, byl použit anglický název pro délku cesty (*pathLength*), což odpovídá názvosloví typickému pro IF. [7] Metoda funguje rekurzivně a prochází podle zadaných dat dříve vytvořený binární strom na základě porovnávání hodnot `dataMember` s hodnotami `splitAttribute` a `splitValue` vnitřních uzlů stromu. Přitom udržuje informaci o aktuální hloubce ve stromu `currentPathLength` obdobným způsobem jako u metody `buildTree()`, tedy původní volání funkce je provedeno bez defaultně zadaného parametru, rekurzivní volání je provedeno z aktualizovanou hodnotou hloubky. Rekurze je ukončena, pokud je dle hodnoty `dataMember` dosaženo externího uzlu. Metoda vrací aktuální hloubku ve stromu povýšenou o hodnotu funkce $c(\psi)$ vypočtenou pro velikost dat aktuálního uzlu (volání metody `calculateC`). Hodnota funkce $c(\psi)$ je různá od nuly,

pokud velikost dat v uzlu je větší než 1 (na základě limitu `heightLimit` došlo k předčasnému ukončení rekurze při tvorbě stromu).

Ukončení rekurze v metodě `pathLength()` při dosažení externího uzlu:

```
if (iTree instanceof ExternalNode) {
      return currentPathLength + this.calculateC(iTree.size);
}
```

`calculateC(size)`

Metoda je volána výše popsanou metodou `pathLength()` s velikostí dat v uzlu na vstupu (`size`). Obsahuje jednoduchý `if-else` výraz. Pokud je velikost dat uzlu větší než 2 vrací hodnotu dopočítanou dle příslušného vztahu [7], pokud je velikost dat uzlu přesně 2, vrací 1 a jinak vrací nulu. Hodnota 1 odpovídá logice toho, že pokud byla rekurze v metodě `pathLength()` zastavena v momentě, kdy je velikost dat v uzlu 2, skutečná hloubka prvku by byla o 1 větší (došlo by ještě k jednomu poslednímu dělení dat). Pokud je velikost dat 1, došlo k izolaci dat, a proto se hloubka dat neupravuje (tato metoda vrací nulu). V případě větší velikosti dat je hodnota hloubky ve stromu upravena příslušnou hodnotou funkce *c(ψ)*.

`dataPathLength()`

Vypočítá pro každý prvek dat průměrnou hloubku ze všech stromů IF. Tato metoda nemá žádné vstupní parametry, pracuje přímo s proměnnou `data` třídy `IsolationForest`. Obsahuje vnořenou smyčku, díky které pro každý prvek dat (vnější smyčka) a pro každý strom (vnitřní smyčka) spočítá pomocí volání metody `pathLength()` průměrnou hloubku ve stromu. Vypočtené hodnoty jsou ukládány do pole `averagePathLengths`, které má délku odpovídající vstupním datům a na příslušných indexech obsahuje odpovídající hodnotu. Toto pole je metodou vraceno.

`dataAnomalyScore(numberOfAnomalies = 0)`

Vypočítá skóre anomálie pro všechna data. Za použití metody `dataPathLength()` je vytvořeno pole s průměrnými hloubkami ve stromech IF. Smyčka, která iteruje přes hodnoty v tomto poli dopočítává dle příslušného vztahu hodnotu skóre anomálie [7] pro každý prvek dat a ukládá ji do pole `dataAnomalyScores`, které potom metoda navrací. Je zde také využito metody `calculateC()`, protože funkce *c(ψ)* se ve zmíněném vztahu vyskytuje. Metoda má volitelný parametr s defaultní hodnotou nula. Pokud je tento parametr použit s hodnotou větší než nula, volá se metoda `maxAnomalyScore()`, která vypíše informace o požadovaném počtu anomálií.

`maxAnomalyScores(dataScores, dataLengths, numberOfMaxValues)`

Pomocná metoda volaná volitelně metodou `dataAnomalyScores()`. Vypíše do konzole informaci o požadovaném počtu prvků dat seřazených podle nejvyšší hodnoty skóre anomálie, tedy takových, u kterých je nejpravděpodobnější, že se jedná o anomálie. Na vstupu jsou dvě pole obsahující skóre anomálie a průměrnou hloubku ve stromu pro všechna data. Třetím vstupem je počet prvků dat, které se mají zobrazit. Metoda obsahuje smyčku `do-while`, která vyhledává pomocí metody `Math.max()` nejvyšší skóre anomálie v příslušném poli a ukládá jejich indexy do pole `maxValuesIndexes`, dokud toto pole nedosáhne požadované délky. Na obrázku 11 je výstup do konzole obsahující informace o pěti prvcích dat s nejvyšším skóre anomálie. Výstup obsahuje index v původním poli dat, skóre anomálie, průměrnou hloubku ve stromu a hodnotu dat. Prvek dat [10,10], který byl vyhodnocen jako nejvíce anomální, je volen pro demonstraci na indexu 0 v poli vstupních dat.

```
>>>            EVALUATION OF DATA            <<<
5 values with highest Anomaly Score are:
| Index of data: 0     | anomaly Score: 0.8557073711753017    | path length: 2.29    | data value: [10,10]
| Index of data: 226   | anomaly Score: 0.6625347940708648    | path length: 6.05    | data value: [1.01,1]
| Index of data: 121   | anomaly Score: 0.6197941232294966    | path length: 7.03    | data value: [1.25,3.94]
| Index of data: 18    | anomaly Score: 0.5925744847639874    | path length: 7.69    | data value: [1.14,3.8]
| Index of data: 17    | anomaly Score: 0.5865568421843318    | path length: 7.84    | data value: [3.2,1.01]
```

Obrázek 11 Výstup do konzole z metody maxAnomalyScores()

5.2.3 Vizualizace

Kód pro vizualizaci obsahuje tři metody, z nichž jedna je pomocná pro vybudování grafu v DOT jazyku, další dvě metody ji využívají.

`buildGraph(graph, treeNode, nodeIdCounter = 0)`

Jedná se o metodu, která projde zadaný strom a vybuduje kód grafu v jazyku DOT pomocí *npm* balíčku *Graphviz* a jeho metod `addNode()` – nový uzel, `addEdge()` – nová hrana a `set()` – úprava vzhledu uzlů. Vstupem je předem inicializovaný prázdný graf a kořen stromu pro vizualizaci. Třetí parametr (defaultně nastaven na nulu) slouží pro inicializaci proměnné pro číslování uzlů. Metoda obsahuje pomocnou rekurzivní funkci pro zajištění unikátního očíslování jednotlivých uzlů stromu. Při průchodu stromem jsou do grafu přidávány odpovídající uzly s příslušnými informacemi pro vizualizaci. Interním uzlům je přidělena

informace o parametru a jeho hodnotě pro dělení dat, externím hloubka v grafu, oběma pak velikost dat v uzlu. Správného přiřazení hran mezi uzly je docíleno průběžným číslováním uzlů pomocí proměnné `nodeIdCounter`, tak aby rodičovský uzel byl vždy správně hranami propojen se svými potomky. Všem uzlům je přiřazen tvar *box*, tedy obdélník. U externích uzlů je nastaveno orámování širší čarou pro lepší vizuální rozlišení od interních uzlů, zejména u větších grafů.

`exportTree(treeToExport, exportFormat, fileName, exportInfo = true)`

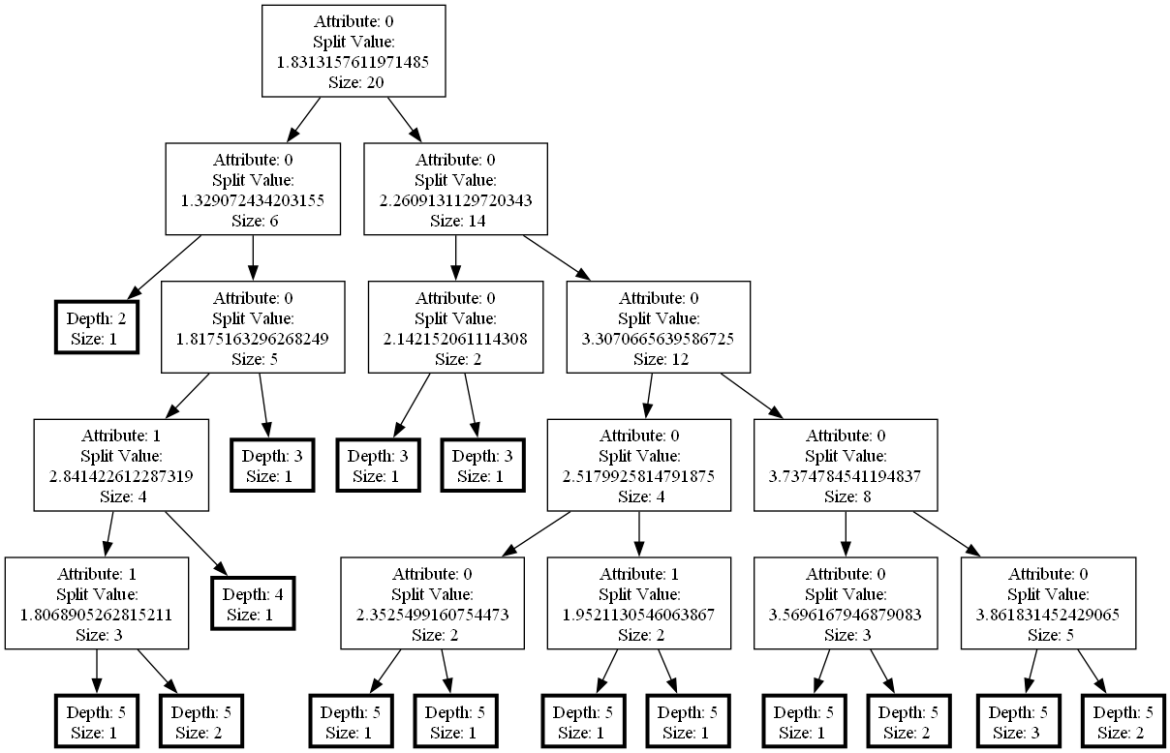
Vyexportuje strom do souboru požadovaného jména a formátu. Metoda vytvoří prázdný graf *Graphviz* do něhož je pomocí `buildGraph()` vložena reprezentace grafu v DOT jazyku. Ta je potom metodou `output` z balíčku *Graphviz* vyexportována do souboru. Vstupem je strom z pole `forest`, formát exportu (řetězec) a název souboru (řetězec). Čtvrtým parametrem je boolean o zobrazení informace o exportu. Defaultní `true` vypíše jednoduché info do konzole s názvem souboru, volání s `false` je využito při exportu celého stromu, viz níže popis metody `exportForest()`. Jako výstupní formáty lze používat formáty podporované softwarem *Graphviz*, např. *npm, bmp, jpeg, tif, svg*, nebo též *dot*, který uloží do souboru reprezentaci stromu v textovém formátu jazyka DOT. Ošetření výjimek na vstupu od uživatele ověřuje, zda strom k exportu je skutečně strom (kořenový uzel je `InternalNode`), dále zda formát a jméno souboru k exportu jsou řetězce. Ověřuje se také volitelný čtvrtý parametr jako boolean. Na obrázku 12 je příklad vizualizace stromu pro vzorek dat velikosti 20 s dopočítaným limitem `heightLimit = 5`, při kterém byla ukončena tvorba stromu. V nejhlubší vrstvě se tedy nacházejí uzly s různou velikostí dat (zde maximálně 3).

Volání metody, generující soubor „treeExport.png“ pro strom na indexu 0 z pole `forest`:

```
exportTree(myForest.forest[0], “png“, “treeExport“);
```

`exportForest(forestExportFormat, fileName)`

Vyexportuje všechny stromy dané instance třídy `IsolationForest`, tak že prochází ve smyčce pole `forest` a volá na jednotlivé stromy metodu `exportTree()`. Při těchto voláních je čtvrtý argument zadán jako `false`, čímž je zrušen mnohonásobný výpis informace o exportu jednotlivých stromů. Namísto toho sama metoda `exportForest()` informuje uživatele v konzoli o exportu celého lesa. Vstupy jsou požadovaný formát a název souboru (řetězce). Ke jménu souboru je přidán příslušný index stromu. Ošetření výjimek na vstupu je řešeno obdobně jako u `exportTree()`.



Obrázek 12 Vizualizace stromu použitím metody exportTree()

6 TESTOVÁNÍ

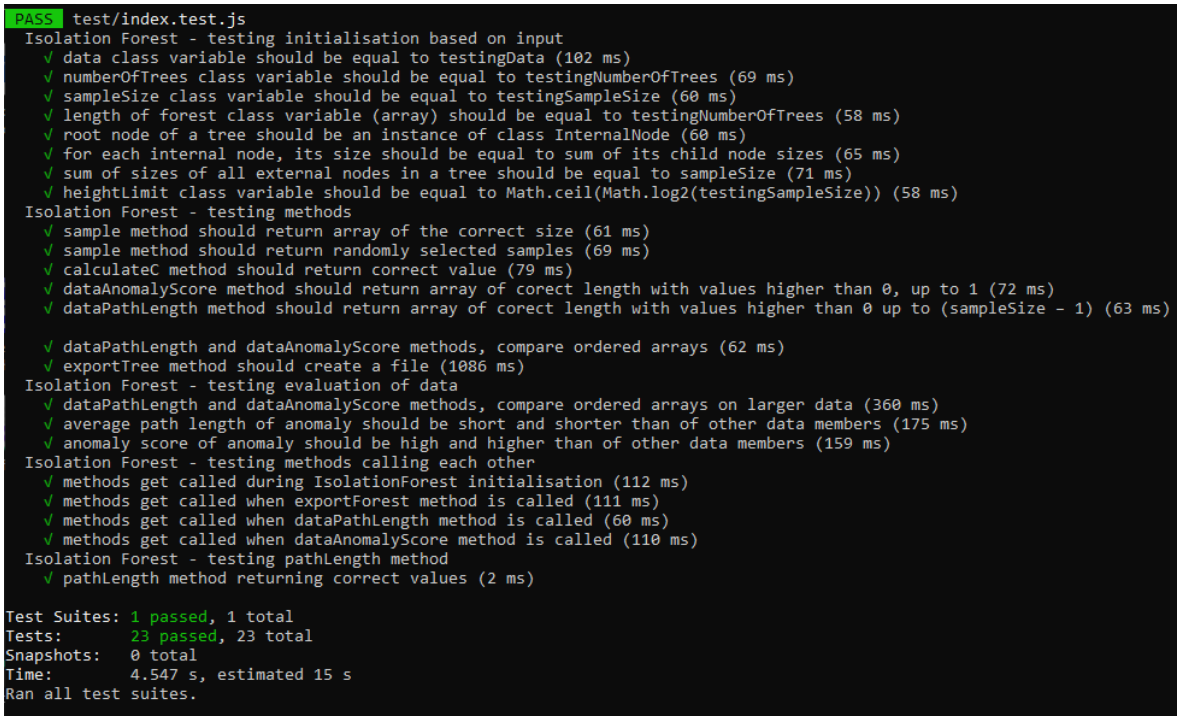
Pro automatizované testování knihovny bylo využito testovacího frameworku *Jest*, zmíněného v kapitole 3.4. Části knihovny, které nebylo možné testovat automaticky (zejména vizuální výstupy) byly testovány manuálně. Testování proběhlo na systému Windows 10 a Linux Debian 12.

6.1 Testování pomocí Jest

Samostatný soubor s testy s názvem „index.test.js“ se nachází v adresáři „test“. Tento test se automaticky spustí v konzoli pomocí příkazu:

```
npm test
```

Celkem bylo navrženo 23 testů, každý z nich má informativní popis, výsledek běhu testů je na obrázku 13.



Obrázek 13 Výsledek testů pomocí knihovny Jest

Kód obsahuje pět sekcí `describe`, které dělí testy do níže popsaných skupin.

Inicializace třídy IsolationForest:

Tato skupina obsahuje 8 testů souvisejících s vytvořením instance třídy `IsolationForest`. Část testů je zaměřena na kontrolu proměnných třídy, zda po inicializaci obsahují správné hodnoty. Další testy jsou věnovány kontrole vytvořených stromů. Každý strom musí jako

kořenový uzel mít instanci třídy `InternalNode`. Dále musí každý vnitřní uzel stromu mít velikost dat odpovídající součtu velikostí dat jeho dvou potomků. Na závěr je testováno, zda součet velikostí dat v externích uzlech odpovídá velikosti vzorku dat (tedy velikosti dat v kořenovém uzlu stromu).

Testování metod:

Zde je testována metoda `sample()`, zda navrací náhodné pole správné velikosti a `calculateC()`, zda navrací správnou hodnotu na základě různých vstupů. Metoda `dataAnomalyScore()` musí navracet pole správné velikosti s hodnotami větší než 0 a maximálně 1. Metoda `dataPathLength()` musí navracet pole správné velikosti s hodnotami většími než 0 až po hodnoty velikosti vzorku dat ponížené o 1. V dalším testu jsou vytvořeny dvě stejná pole s hodnotami z obou vyhodnocovacích metod. Jedno z těchto polí je seřazeno od nejvyššího skóre anomálie, druhé od nejkratší průměrné hloubky. Testuje se, zda jsou pole seřazeny ve stejném pořadí, a tedy zda dochází k odpovídajícímu přepočtu hloubky ve stromech na skóre anomálie pro všechna data. Na konci této sekce je testována metoda `exportTree()` na export souboru.

Testování fáze vyhodnocení:

Na větším vzorku dat je zde znovu použit test seřazení hodnot v polích popsaný výše. Dále je testováno, zda anomální bod je vyhodnocen s vysokým skóre anomálie a krátkou průměrnou hloubkou ve stromech ve srovnání s normálními daty.

Volání metod mezi sebou:

Nejprve se zde testuje, zda byly volány příslušné metody při vytvoření instance třídy `IsolationForest` a také obsah informace vytištěné do konzole metodou `printForestInfo()`. Následující testy kontrolují správné volání metod mezi sebou, například při volání metody `exportForest()` musí být volána metoda `exportTree()` tolikrát, kolik je stromů v poli `forest` a se správnými argumenty.

Test metody pathLength():

Tato část testovacího kódu obsahuje pouze jeden test, pro který byl „manuálně“ vytvořen testovací strom. Nebyl tedy vygenerován metodou `buildTree()`, ale postupně vytvářením instancí tříd `InternalNode` a `ExternalNode`. Tento strom byl potom použit při volání a testování metody `pathLength()` pro různá data. Díky tomuto postupu bylo předvídatelné, jakou hloubku ve stromu má metoda vracet. Dále je zde také tento strom vyexportován

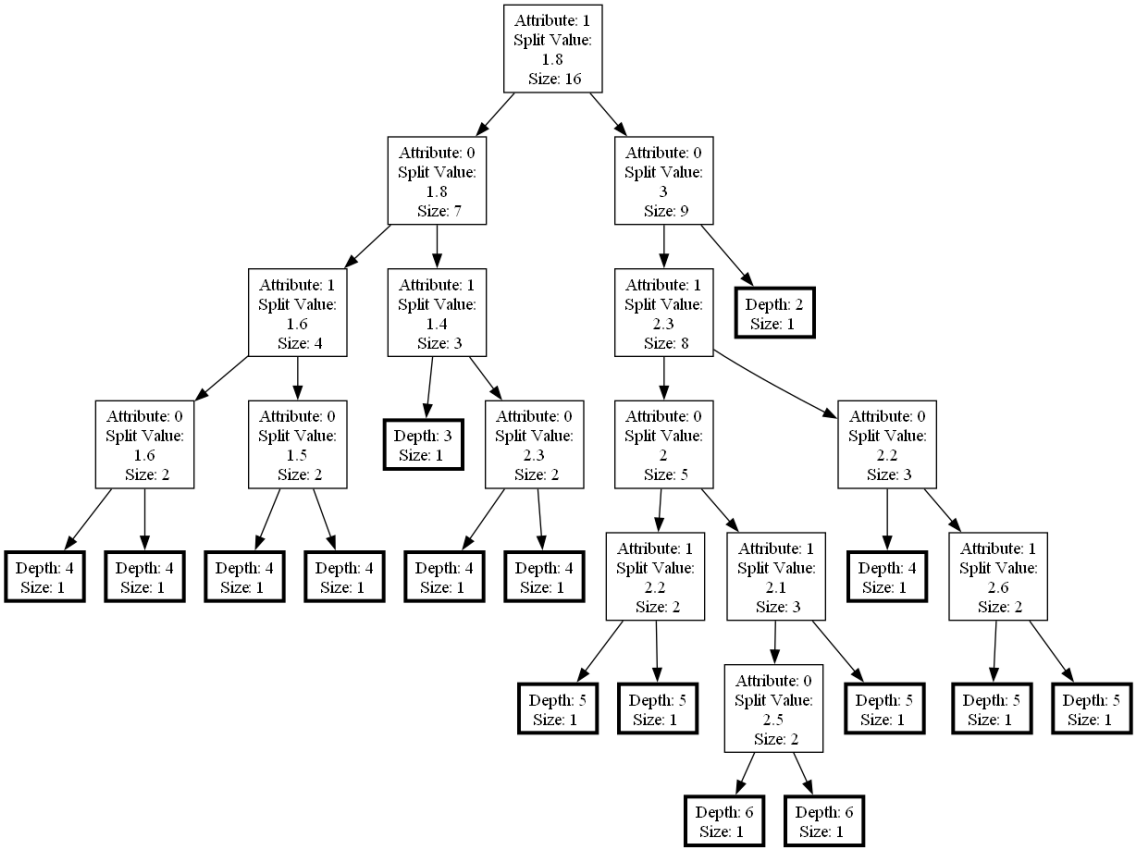
metodou `exportTree()`, aby mohl být použit pro kontrolu exportu do souboru v následující podkapitole.

6.2 Manuální testování

Zde je popsáno manuální testování knihovny. Jedná se o kontrolu správnosti vizualizace stromů a export do různých formátů metodami `exportTree()` a `exportForest()`.

6.2.1 Testování vizualizace

Pro kontrolu správného vytvoření vizualizace byl použit strom se stejnými hodnotami jako na obrázku 5 v kapitole 2.1.2. Nyní byla struktura stromu vytvořena pomocí tříd `InternalNode` a `ExternalNode`. Tento strom byl ve výše zmíněném testu vyexportován metodou `exportTree()` do formátu *png* a lze jej vidět na obrázku 14. Porovnáním obrázků došlo k ověření správného přiřazování informací o atributech a hodnotách pro dělení dat, stejně jako samotného tvaru stromu – propojení uzlů mezi sebou.



Obrázek 14 Strom pro kontrolu správné vizualizace, obsahově totožný s obrázkem 5

6.2.2 Testování exportu do souborů

Pro testování exportu do různých formátů bylo využito souboru „experiment5.js“ ve složce „experiments“. Tento soubor obsahuje volání metody `exportTree()` a `exportForest()` do formátů *png, bmp, dot, gif, jpeg, jpg, pdf, svg, tif*. Všechny vyexportované soubory měly očekávaný obsah.

7 ZVEŘEJNĚNÍ

Celá knihovna má být zveřejněna jako balíček *npm*, což je podmíněno tím, že kód musí být nejprve zveřejněn na serveru *GitHub*. Obě činnosti jsou popsány v následujících podkapitolách.

7.1 Zveřejnění kódu na serveru GitHub

Pro vytvoření vzdáleného repozitáře na serveru *GitHub*, který slouží jako webová nadstavba pro verzovací systém *Git*, je nutno mít účet na serveru a dále na počítači nainstalovaný *Git*. Vytvoření nového repozitáře na serveru *GitHub* je velmi přímočaré. Po kliknutí na tlačítko *New* je nutno zadat několik základních informací jako je volba mezi veřejným či soukromým repozitářem a jeho název, dále je také možnost přidat soubor „README.md“. Na další obrazovce je již k dispozici odkaz na nově vytvořený repozitář, který se využije při propojení se systémem *Git* na lokálním počítači pro nastavení vzdáleného úložiště.

Pro inicializaci repozitáře a jeho propojení s *GitHubem* je nutno v terminálu v pracovním adresáři projektu zadat několik příkazů, které jsou níže stručně popsány.

Inicializace repozitáře:

```
git init
```

Přidání adresářů a souborů lze provézt několika způsoby, zde jsou přidány všechny:

```
git add –all
```

Příkaz `commit` s typickým komentářem pro prvotní *commit* (revizi) kódu:

```
git commit -m "Initial commit"
```

Tímto je vytvořen lokální repozitář, který je však dostupný pouze na příslušném počítači. Pro jeho přiřazení ke vzdálenému repozitáři na serveru *GitHub* slouží následující příkaz, kde se využije výše zmíněný odkaz. Zde je uveden již odkaz na konkrétní repozitář s názvem „isolation-forest-visualization“ vytvořený pro knihovnu.

```
git remote add origin https://github.com/AlenaStrnadova/isolation-forest-visualization.git
```

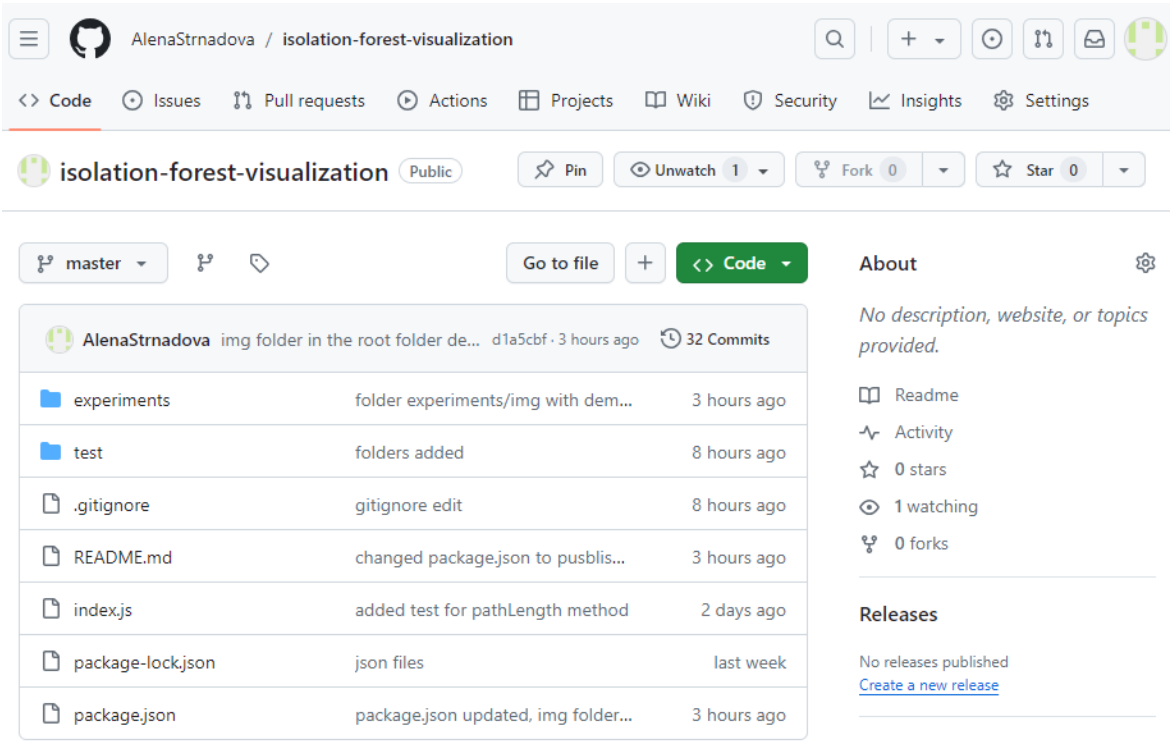
Následujícím příkazem `push` s uvedenými volbami se provede uložení kódu do hlavní větve (*master branch*) repozitáře:

```
git push --set-upstream origin master
```

Při změnách v kódu pak lze opět využít stejný příkaz pro přidání souborů i pro *commit*, uložení kódu příkazem `push` už při dalším využití probíhá bez výše uvedených voleb.

Tímto byl vytvořen veřejný repozitář s kódem knihovny (viz obrázek 15), obsahující příslušné složky a soubory. Dostupný je na odkazu:

https://github.com/AlenaStrnadova/isolation-forest-visualization



Obrázek 15 Veřejný repozitář knihovny na serveru *GitHub*

7.2 Zveřejnění knihovny jako balíčku npm

Po vytvoření *GitHub* repozitáře je samotné dokončení zveřejnění knihovny jako balíčku *npm* celkem jednoduché. [26] Opět se pracuje pomocí příkazů v konzoli, pokud není uvedeno jinak v pracovním adresáři projektu. Podmínkou je samozřejmě instalace *Node.js* s jehož součástí je i možnost pracovat *npm* balíčky. Dále je nutno mít vytvořený účet na serveru *npmjs.com*. Následuje výčet použitých příkazů a jejich stručný popis.

Inicializace vyzve uživatele k postupnému zadání informací o balíčku, včetně názvu, popisu, licence, klíčových slov, vše viz obrázek 16. V závorkách jsou uvedeny varianty odpovědí přednastavené systémem, které stačí pouze odsouhlasit. Je nutné dbát na unikátnost názvu

balíčku, tedy aby stejný název již na serveru nefiguroval. Také se zde zadává odkaz na *GitHub* repozitář. Zadané informace se uloží do souboru „package.json“ a dají se v něm později editovat. Inicializace se provede pomocí příkazu:

```
npm init
```

```
C:\Users\A\Documents\isolation-forest-visualization>npm init
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sensible defaults.

See `npm help init` for definitive documentation on these fields
and exactly what they do.

Use `npm install <pkg>` afterwards to install a package and
save it as a dependency in the package.json file.

Press ^C at any time to quit.
package name: (isolation-forest-visualization)
version: (1.0.0)
description: Exports visual representation of trees in Isolation Forest into different formats using Graphviz. Evaluates
 the anomaly scores and average path lengths for input data.
entry point: (index.js)
git repository: https://github.com/AlenaStrnadova/isolation-forest-visualization.git
keywords: isolation forest visualization graphviz anomaly detection export image binary tree
author: Alena Strnadova
license: (ISC) MIT
About to write to C:\Users\A\Documents\isolation-forest-visualization\package.json:
```

Obrázek 16 Inicializace *npm* balíčku

V tomto momentě lze již funkcionalitu balíčku vyzkoušet lokálně před jeho zveřejněním. Nejprve se balíček nastaví jako globálně dostupný:

```
npm link
```

Kdekoliv v počítači pak lze následujícím příkazem balíček přidat:

```
npm link isolation-forest-visualization
```

A je zde možno vytvořit soubor se zkušebním kódem, ve kterém se balíček naimportuje:

```
const IsolationForest = require('isolation-forest-visualization');
```

Pokud zkouška proběhla v pořádku, následuje zveřejnění balíčku. Příkaz `login` vyzve k přihlášení do předem vytvořeného účtu na serveru *npmjs.com*, příkaz `publish` poté balíček zveřejní:

```
npm login
```

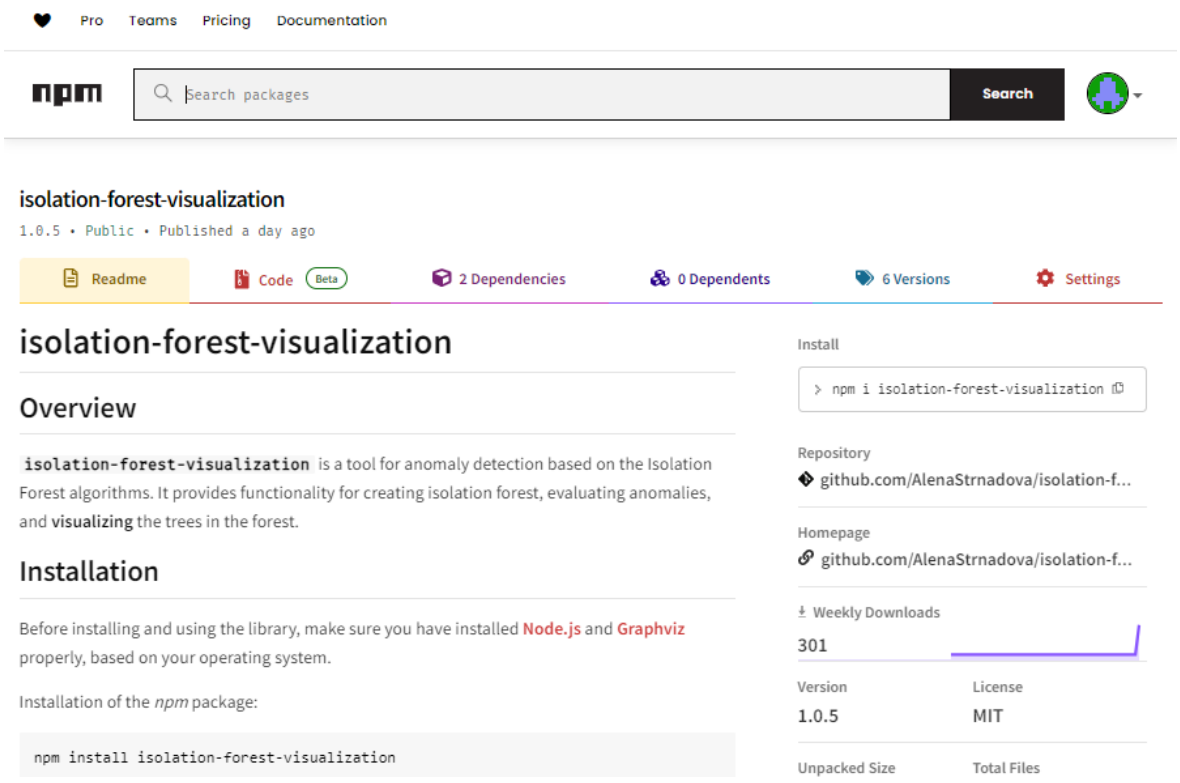
```
npm publish
```

Stejným příkazem `publish` se provádí i zveřejnění aktualizace kódu. V takovém případě je nutné nejdříve v souboru „package.json“ změnit verzi software na vyšší např. při malé změně z verze 1.0.0 na verzi 1.0.1.

Po úspěšném dokončení předchozího postupu obdrží uživatel potvrzující email se základními informacemi. Tímto je knihovna zveřejněna jako nový *npm* balíček a je dostupná na odkazu:

https://www.npmjs.com/package/isolation-forest-visualization

Na obrázku 17 je vidět nová knihovna s názvem „isolation–forest–visualization“ na serveru *npmjs.com*. Stránka obsahuje informace ze souboru „README.md“, tedy dokumentaci softwaru (viz kapitola 8), klíčová slova, odkaz na *GitHub*, licenci, počet stažení atd.



Obrázek 17 Zveřejněná knihovna isolation-forest-visualization

8 DOKUMENTACE A PŘÍKLADY POUŽITÍ

Tato kapitola, uzavírající celou práci, popisuje knihovnu z pohledu uživatele. Nejprve je zde vysvětlena instalace a způsob práce s knihovnou pomocí jejích metod. Tato část bude také přiložena v anglické variantě k balíčku *npm*. Následně jsou uvedeny příklady použití pomocí souborů ve složce „experiment“ dodaných s balíčkem. Tyto příklady slouží jak pro předvedení práce s knihovnou, tak pro demonstraci vlastností IF. Zejména první z příkladů („experiment1.js“) je podrobněji popsán a může posloužit pro hlubší pochopení principů popsaných v teoretické části.

8.1 Dokumentace knihovny isolation-forest-visualization

Úvod:

Algoritmus Isolation Forest sloužící pro detekci anomálií tvoří na základě vstupních dat sadu binárních stromů. Balíček „isolation–forest–visualization“ umožňuje vizualizaci těchto stromů a jejich export do souborů. Dále umožňuje vyhodnocení dat z hlediska anomálií a zobrazení jejich základních vlastností.

Instalace:

Podmínkou pro instalaci *npm* balíčku je instalace *Node.js* a instalace softwaru *Graphviz*.

Pokud se balíček instaluje v adresáři, kde ještě nebyl vytvořen soubor „package.json“, je nutno nejprve použít příkaz:

```
npm init -y
```

Instalace balíčků se provádí příkazem:

```
npm install isolation-forest-visualization
```

V souboru se zdrojovým kódem v jazyku JavaScript je nutno balíček naimportovat pomocí následujícího řádku kódu:

```
const {IsolationForest} = require('isolation-forest-visualization');
```

Vytvoření instance třídy IsolationForest:

Naimportovaná třída `IsolationForest` obsahuje metody pro fázi učení, fázi vyhodnocení, i pro vizualizaci.

Vytvoření instance třídy `IsolationForest` se vstupy `data` (data k analýze), `trees` (počet stromů) a `sample` (velikost vzorku dat):

```
const data = [[2, 3], [4, 3], [5, 5], [4, 6]];
const trees = 5;
const sample = 2;
const myForest = new IsolationForest(data, trees, sample);
```

Data musí být ve formě pole polí, kdy vnitřní pole musí mít všechna stejnou délku. Data se nesmějí opakovat (musí být unikátní).

Metoda dataPathLength() – výpočet průměrné hloubky ve stromech:

Navrací pole s průměrnou hloubkou ve stromech pro každý prvek dat:

```
const lenghts = myForest.dataPathLength();
```

Metoda dataAnomalyScore() – výpočet skóre anomálie:

Navrací pole se skóre anomálie pro každý prvek dat:

```
const scores = myForest.dataAnomalyScore();
```

Metoda dataAnomalyScore() – vyhodnocení nejpravděpodobnějších anomálií:

Pokud je tato metoda volána s volitelným argumentem v podobě celého čísla, zobrazí v konzoli informaci o požadovaném počtu nejpravděpodobnějších anomálií, přičemž stále může vracet pole se všemi hodnotami skóre anomálie:

```
myForest.dataAnomalyScore(5);
```

Metoda exportTree() – export jednoho stromu do souboru:

Stromy jsou ve třídě `IsolationForest` ukládány do proměnné třídy v podobě pole s názvem `forest`. Pomocí metody `exportTree()` je možno vyexportovat vizualizaci požadovaného stromu z pole `forest` do souboru zadaného formátu a názvu. Formát může být jakýkoliv formát podporovaný softwarem *Graphviz*, např. *png, jpg pdf, svg, dot*. Název může obsahovat i název existujícího adresáře (např. „img/tree“), jinak se soubor uloží do hlavního adresáře projektu. Formát a název souboru jsou řetězce (*string*):

```
const myTree = myForest.forest[0];
const exportFormat = 'png';
const fileName = 'tree';
myForest.exportTree(myTree, exportFormat, fileName);
```

Metoda exportForest() – export všech stromů do souborů:

Vyexportuje všechny stromy z pole `forest` do souborů zadaného formátu a názvu. K názvu souborů je vždy přidán index z pole `forest`. Pro název souboru a formát platí stejná pravidla jako u metody `exportTree()`. Uvedené volání metody vyexportuje soubory s názvy „forestExport0.png“, „forestExport1.png“ atd. podle velikosti pole `forest` (podle počtu stromů):

```
const exportFormat = 'png';
const fileName = 'forestExport';
myForest.exportForest(exportFormat, fileName);
```

8.2 Příklady použití

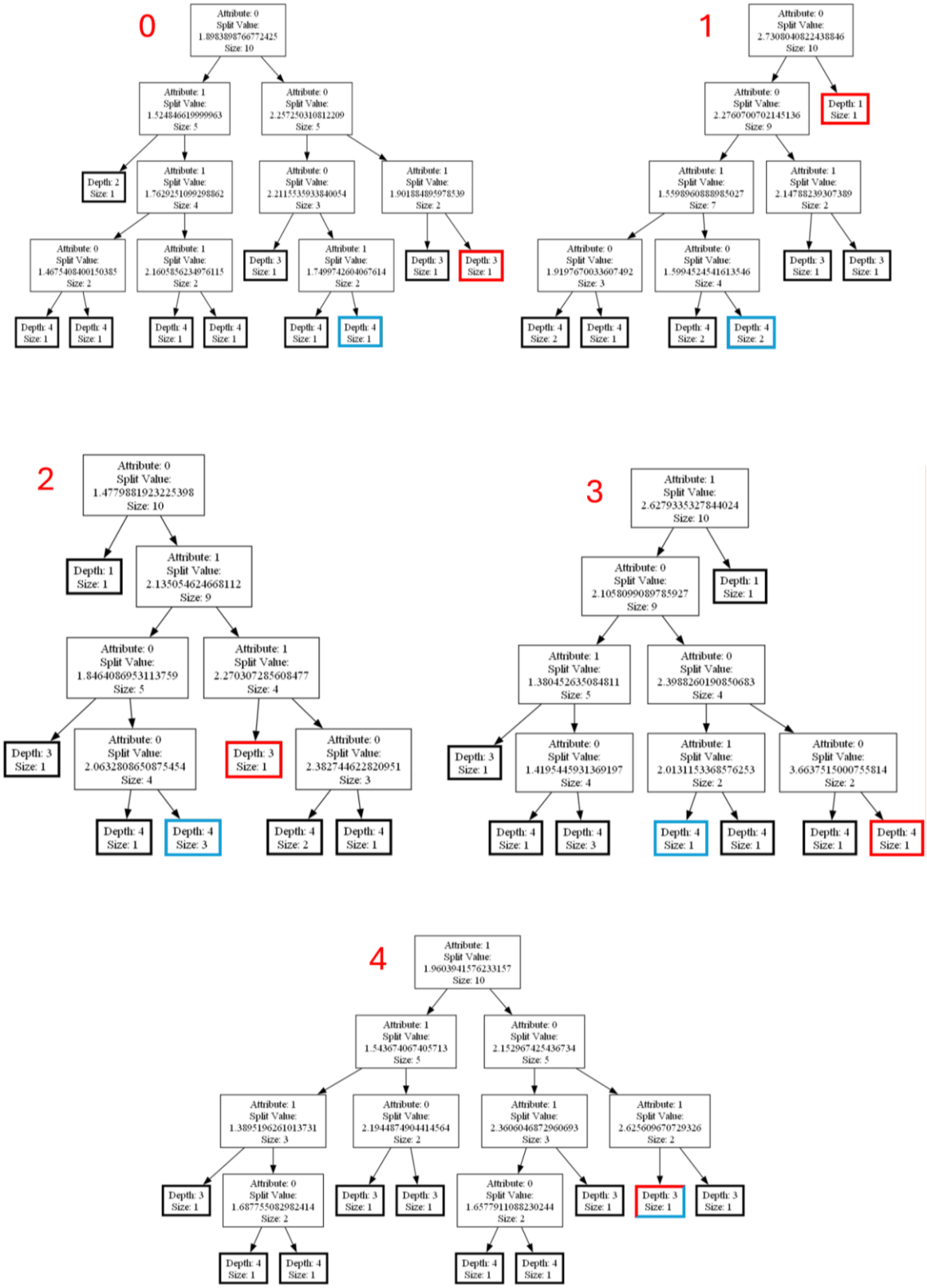
Knihovna obsahuje složku „experiments“ s pěti soubory „experiment1.js“ až „experiment5.js“, které jsou dále popsány. Tyto soubory obsahují příklady kódu představující funkcionalitu knihovny na různých typech dat, příklady exportů do souborů i vyhodnocení anomálií.

8.2.1 Experiment1 – export do souborů

Soubor demonstruje použití knihovny na datech velikosti 16, počet stromů 5, velikost vzorku 10. Ukazuje použití metody `exportTree()` na strom z pole `forest` na indexu 0 a metody `exportForest()` pro export všech stromů. Celkový počet stromů je 5, do složky „img“ bylo tedy vyexportováno celkem šest souborů (jeden metodou `exportTree()` a pět metodou `exportForest()`). Nastala situace, kdy soubor „treeExperiment1.png“ vizuálně odpovídá souboru „forestExperiment0.png“, což je logické, protože se jedná o tentýž strom, pouze vyexportovaný různými metodami. Popsané volání obou metod:

```
myForest.exportTree(myForest.forest[0], 'png', 'img/treeExperiment1');
myForest.exportForest('png', 'img/forestExperiment');
```

Na obrázku 18 jsou všechny stromy vyexportované metodou `exportForest()`, označené příslušnými indexy z pole `forest`. Data pro tento experiment byla zvolena stejně, jako v kapitole 2.1.2, kde na nich byla vysvětlována úplná izolace prvků – externí uzly měly velikost dat 1. Zde je možno vidět, že dělení dat probíhalo u všech stromů maximálně do dopočítaného limitu `heightLimit = 4`. Na této úrovni mají tudíž některé uzly velikost dat větší.

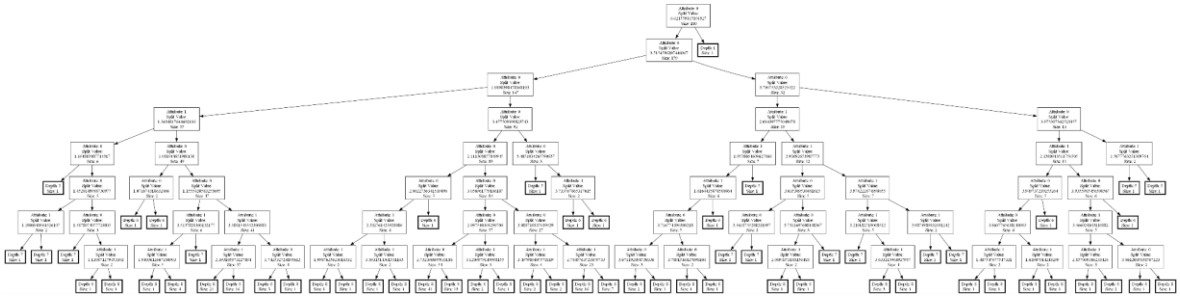


Obrázek 18 Sada stromů vyexportovaná metodou exportForest()

Pro demonstraci chování IF byl na obrázku 18 v každém stromu červeně zvýrazněn uzel, do kterého zapadne anomální bod [4.5, 2.25]. Modře byl potom zvýrazněn uzel pro normální bod [2.25, 2.0]. Přestože se jedná jen o malý počet stromů, má anomálie většinou krátkou cestu stromem, v průměru je hodnota 2,2. Naproti tomu druhý ze zkoumaných bodů má průměrnou hloubku 4,46. Tento bod dokonce čtyřikrát dosáhl limitní hloubky 4, z toho ve dvou případech (strom 1 a 3) bylo nutno upravit hloubku pomocí hodnoty *c(ψ)*, protože velikost dat v uzlu byla větší než 1. Zajímavostí je také strom 4, ve kterém oba zkoumané body zapadli do stejného uzlu o velikosti dat 1. To působí zdánlivě nelogicky, je to ale způsobeno tím, že stromy byly na rozdíl od ukázky v kapitole 2.1.2, tvořeny pouze na vzorku dat (velikost 10). Přestože oba zkoumané body patří do původní sady dat, nemusí mít každý z nich ve stromu „vlastní“ izolovaný uzel.

8.2.2 Experiment2 – export a vyhodnocení dat

V této ukázce jsou jako data použity dvoudimenzionální datové prvky, velikost dat je 256. Velikost vzorku dat je zvolena jako 180, počet stromů 100. Tímto zadáním se experiment nejvíce blíží reálnému použití, kdy autoři algoritmu doporučují pro velké množství dat vzorek dat 256 a počet stromů 100. [7] Po inicializaci IF dochází k exportu stromu na indexu 0 do formátu *png*. Na obrázku 19 je náhled zmíněného vyexportovaného stromu. Je vidět, že tento strom vytvořený pro vzorek dat 180, je již hodně rozsáhlý, zejména roste do velké šířky. Vertikální velikost omezuje limit hloubky vypočítaný zde na 8. Dále je volána metoda `dataAnomalyScore(5)`, která na základě využití volitelného argumentu zobrazí 5 nejpravděpodobnějších anomálií. Výstup do konzole je vidět na obrázku 20. Uživatel je informován o vytvoření IF a jeho parametrech, dále o exportu stromu do souboru, kdy je i zobrazeno jeho jméno. Nakonec je zobrazena informace o pěti nejpravděpodobnějších anomáliích, jejich index v původním poli dat, skóre anomálie, průměrná hloubka ve stromech IF a hodnota dat.



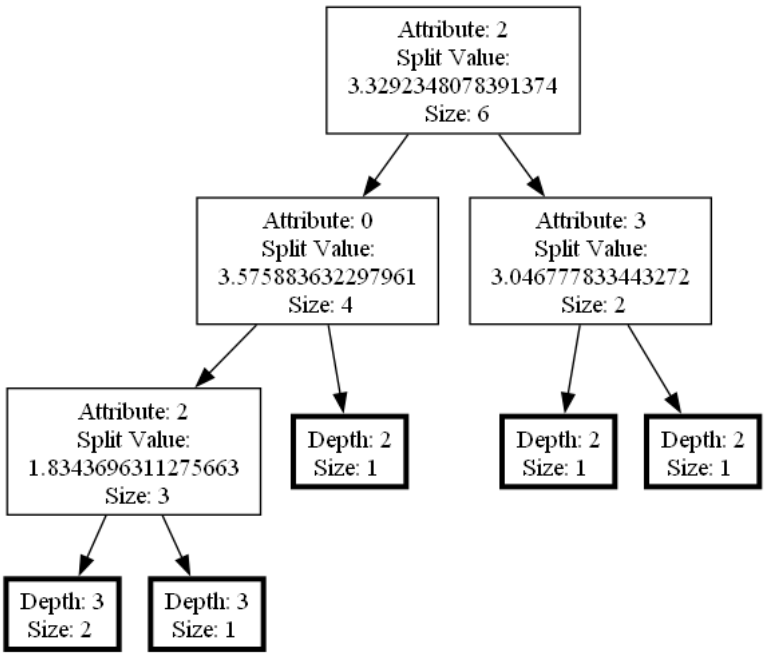
Obrázek 19 Náhled rozsáhlejšího stromu – velikost vzorku 180

```
================================================
  data size: 256, number of attributes: 2
------------------------------------------------
  number of trees: 100, sample size: 180
------------------------------------------------
  height limit of trees set to: 8
------------------------------------------------
  >>>    ISOLATION FOREST CREATED    <<<
================================================


  Exported a tree into img/treeExperiment2.png file.


=============================================================================================================
                          >>>            EVALUATION OF DATA              <<<
-------------------------------------------------------------------------------------------------------------
 5 values with highest Anomaly Score are:
-------------------------------------------------------------------------------------------------------------
| Index of data: 0      | anomaly Score: 0.8202833725817237     | path length: 2.8954306102549263       | data value: [10,10]
| Index of data: 226    | anomaly Score: 0.6671995793221299     | path length: 5.914439292642005        | data value: [1.01,1]
| Index of data: 121    | anomaly Score: 0.5954410422385947     | path length: 7.577502838337654        | data value: [1.25,3.94]
| Index of data: 24     | anomaly Score: 0.5931282761103741     | path length: 7.6343821780161685       | data value: [3.64,4]
| Index of data: 67     | anomaly Score: 0.582606038687657      | path length: 7.8959939565345705       | data value: [3.9,1.05]
=============================================================================================================
```

Obrázek 20 Výstup do konzole obsahující vyhodnocení dat

8.2.3 Experiment3 – vícedimenzionální data

V tomto příkladu byla použita vícedimenzionální data (4 veličiny). Dochází k exportu stromu na indexu 0 a vyhodnocení všech dat (velikost 10). Na obrázku 21 je ukázka vyexportovaného stromu s velikostí vzorku 6. Za povšimnutí stojí, že strom vypadá stejně jako u dat se dvěma veličinami, pouze se u vnitřních uzlů střídá více atributů pro dělení dat. Toto by platilo u dat s libovolným počtem veličin.



Obrázek 21 Příklad stromu s vícedimenzionálními daty

8.2.4 Experiment4 – vyhodnocení dat

V tomto experimentu je uveden příklad výstupu vyhodnocení dat pomocí hodnot navrácených metodami `dataAnomalyScore()` a `dataPathLength()`, kdy si uživatel sám zvolil, jak se vyhodnocení zobrazí. Na obrázku 22 je výstup do konzole, po informaci o vytvoření IF následuje uživatelem upravená informace o vyhodnocení dat. V této ukázce je situace demonstrována na malých datech o velikosti 4 a informace jsou zobrazeny pro všechna data. Takto by však bylo možno zobrazit informace o všech prvcích dat i pro velká data, nebo naopak jen pro několik prvků na určitých indexech v původním poli dat.

```
================================================
   data size: 4, number of attributes: 2
------------------------------------------------
   number of trees: 30, sample size: 4
------------------------------------------------
   height limit of trees set to: 2
------------------------------------------------
   >>>    ISOLATION FOREST CREATED    <<<
================================================

data member on index 0: value: [3,1], anomaly score: 0.4280557096578676, average path length: 2.2666666666666666

data member on index 1: value: [5,2], anomaly score: 0.43887265398526404, average path length: 2.2

data member on index 2: value: [4,4], anomaly score: 0.45561274149673014, average path length: 2.1

data member on index 3: value: [2,3], anomaly score: 0.43887265398526404, average path length: 2.2
```

Obrázek 22 Uživatelem upravená informace o vyhodnocení dat

8.2.5 Experiment5 – export do různých formátů

Demonstrace exportu pomocí metod `exportTree()` a `exportForest()` do nejběžněji používaných grafických formátů (*png, bmp, giv, jpeg, jpg, svg, tif*), formátu *pdf* a *dot*. Tento soubor byl také použit pro manuální testování v kapitole 6.2.2.

ZÁVĚR

Cílem této práce bylo navrhnout a implementovat knihovnu pro vizualizaci algoritmu IF, čehož bylo dosaženo vytvořením *npm* balíčku. Knihovna byla zveřejněna a může být využita odborníky z praxe i studenty, kteří se chtějí seznámit s technologií IF.

V teoretické části jsou nejprve stručně popsány principy detekce anomálií, včetně zmínění různých metod strojového učení. Ve druhé kapitole následuje popis samotného algoritmu IF, pozornost je věnována zejména principu izolace dat a tvorbě binárních stromů. Popsány jsou potřebné algoritmy pro fázi učení i fázi vyhodnocení. Zmíněna je i varianta IF s úplnou izolací prvků, přestože v této práci není použita. Jedná se však o zajímavou variantu a srovnání obou variant pomáhá i k pochopení algoritmů zde použitých.

Další kapitola práce se věnuje popisu použitých technologií včetně jazyka JavaScript, balíčkového systému *npm* a softwaru *Graphviz*. Zde jsou také vysvětleny základní principy jazyka DOT, který *Graphviz* využívá pro textovou reprezentaci grafů. V této části práce je také stručně popsán framework *Jest* použitý v praktické části pro testování knihovny a použité pracovní prostředí.

V praktické části práce je nejprve knihovna navržena. Pomocí analýzy požadavků byl vytvořen UML diagram tříd a následně jsou zde vypsány navržené metody.

V kapitole o implementaci jsou použité třídy popsány podrobněji. Třída `IsolationForest` slouží pro ukládání vytvořených stromů a obsahuje metody pro veškerou funkcionalitu fáze učení, fáze vyhodnocení i vizualizaci. Přestože hlavním cílem práce bylo vytvořit vizualizaci binárních stromů, knihovna dokáže na základě požadavku uživatele i vyhodnotit anomálie, a to v podobě výčtu nejpravděpodobnějších anomálií, i vyhodnocením skóre anomálie pro všechna data.

Práce pokračuje popisem testování knihovny, které bylo provedeno automaticky i manuálně pro různé funkcionality. Dále je demonstrováno zveřejnění kódu na repozitáři *GitHub* i zveřejnění samotné knihovny jako *npm* balíčku na serveru *npmjs.com*.

Závěr práce je věnován dokumentaci z pohledu uživatele, která obsahuje návod, jak pracovat s knihovnou a také praktické příklady použití. Obdobná dokumentace v anglickém jazyce byla připojena i ke zveřejněnému balíčku *npm*.

SEZNAM POUŽITÉ LITERATURY

[1] HENDL, Jan. *Big data: věda o datech - základy a aplikace*. Průvodce (Grada). Praha: Grada Publishing, 2021. ISBN 978-80-271-3031-3.

[2] DUNNING, Ted a FRIEDMAN, Ellen. *Practical Machine Learning: A New Look at Anomaly Detection*. O'Reilly Media, 2014. ISBN 978-1-491-90408-4.

[3] KISHAN, Mehrotra G.; CHILUKURI, Mohan K. a HUANG, HuaMing. *Anomaly Detection: Principles and Algorithms*. Springer International Publishing, 2017. ISBN 978-3-319-67524-4.

[4] ADARI, Suman Kalyan a ALLA, Sridhar. *Beginning Anomaly Detection Using Python-Based Deep Learning*. Second Edition. Apress, 2024. ISBN 979-8-8688-0007-8.

[5] FEASEL, Kevin. *Finding Ghost in Your Data: Anomaly Detection Techniques with Examples in Python*. Apress, 2022. ISBN 978-1-4842-8869-6.

[6] HOSSAIN, Eklas. *Machine Learning Crash Course for Engineers*. Springer, 2024. ISBN 978-3-031-46989-3.

[7] LIU, Fei Tony; TING, Kai Ming a ZHOU, Zhi-Hua. Isolation Forest. Online. *2008 Eighth IEEE International Conference on Data Mining*. 2008, s. 413-422. ISBN 978-0-7695-3502-9. Dostupné z: https://doi.org/10.1109/ICDM.2008.17. [cit. 2024-02-23].

[8] LIU, Fei Tony; TING, Kai Ming a ZHOU, Zhi-Hua. Isolation-Based Anomaly Detection. Online. *ACM Transactions on Knowledge Discovery from Data*. 2012, roč. 6, č. 1, s. 1-39. ISSN 1556-4681. Dostupné z: https://doi.org/10.1145/2133360.2133363. [cit. 2024-02-27].

[9] LABRECQUE, Josef; LOVE, Jahred a ROSENBAUM, Daniel. *The JavaScript Workshop*. Packt Publishing, 2019. ISBN 978-1-83864-191-7.

[10] FRISBIE, Matt. PROFESSIONAL *JavaScript for Web Developers*. Fifth Edition. Wiley, 2024. ISBN 9781394193219.

[11] ACKERMANN, Philip. *JavaScript: The Comprehensive Guide*. Rheinwerk Publishing, 2022. ISBN 978-1-4932-2286-5.

[12] FUCHS, Paul. *JavaScript Programmieren für Einsteiger*. BMU Media, 2019. ISBN 9783966450195.

[13] FLANAGAN, David. *JavaScript: Das Handbuch für die Praxis*. 7. Auflage. O'REILLY, 2021. ISBN 978-3-96009-157-8.

[14] *About npm*. Online. Npm. 2023. Dostupné z: https://docs.npmjs.com/about-npm. [cit. 2024-03-14].

[15] *About*. Online. Graphviz. 2023. Dostupné z: https://graphviz.org/about/. [cit. 2024-03-16].

[16] *DOT Language*. Online. Graphviz. 2022. Dostupné z: https://graphviz.org/doc/info/lang.html. [cit. 2024-03-16].

[17] *Command Line*. Online. Graphviz. 2023. Dostupné z: https://graphviz.org/doc/info/command.html. [cit. 2024-03-16].

[18] *Node.js Graphviz Module*. Online. Npm. (C) 2010-2019. Dostupné z: https://www.npmjs.com/package/graphviz. [cit. 2024-03-16].

[19] *Jest: Delightful JavaScript Testing.* Online. Dostupné z: https://jestjs.io/. [cit. 2024-03-19].

[20] *Getting Started*. Online. Jest: Delightful JavaScript Testing. Dostupné z: https://jestjs.io/docs/getting-started. [cit. 2024-03-19].

[21] DA COSTA, Lucas. *Testing JavaScript Applications*. Manning Publications Co., 2021. ISBN 9781617297915.

[22] O'REGAN, Gerard. *Concise Guide to Software Engineering*. Second Edition. Springer, 2022. ISBN 978-3-031-07816-3.

[23] LAPLANTE, Phillip A. a KASSAB, Mohamad H. *Requirements Engineering for Software and Systems*. Fourth Edition. CRC Press, 2022. ISBN 978-1-003-12950-9.

[24] *JavaScript Standard Style.* Online. Dostupné z: https://standardjs.com/. [cit. 2024-04-22].

[25] *@use JSDoc*. Online. 2011. Dostupné z: https://jsdoc.app/. [cit. 2024-04-22].

[26] FREECODECAMP. *How to Create and Publish an NPM Package – a Step-by-Step Guide.* Online. SEMAH, Benjamin. FreeCodeCamp. 2023. Dostupné z: https://www.freecodecamp.org/news/how-to-create-and-publish-your-first-npm-package/. [cit. 2024-05-03].

SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK

IF	Isolation Forest
npm	node package manager
MS	Microsoft
FAI	Fakulta aplikované informatiky
UTB	Univerzita Tomáše Bati
CLI	Command line interface
UP	Uživatelský požadavek
FP	Funkční požadavek
NP	Nefunkční požadavek
OOP	Objektově orientované programování

SEZNAM OBRÁZKŮ

SEZNAM TABULEK

SEZNAM PŘÍLOH

PŘÍLOHA P I: PŘILOŽENÉ CD – OBSAH

- Bakalářská práce ve formátu PDF
- Zdrojové kódy knihovny „isolation-forest-visualization“
- Ukázky vizualizace stromů IF