FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

ZADÁNÍ DIPLOMOVÉ PRÁCE

Název:	Systém pro správu neuronových sítí
Student:	Bc. Michal Stejskal
Vedoucí:	Ing. Josef Vogel, CSc.
Studijní program:	Informatika
Studijní obor:	Webové a softwarové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	Do konce zimního semestru 2019/20

Pokyny pro vypracování

Pomocí metod softwarového inženýrství navrhněte a implementujte systém pro vytvoření neuronové sítě pro klasifikaci obrázků, logů či vstupního textu pro účely chatbotů.

Uživatel vybere ze seznamu problémů jaký problém chce řešit, předá vstupní data a definuje způsob zpracování odpovědi.

Systém vytvoří neuronovou síť a spustí trénování nebo vybere již předtrénovanou síť.

Uživatel definuje způsob zpracování odpovědi sítě. Pro každou třídu klasifikace vybere, zda se má odpovědět přímo, nebo zda má být odpověď sítě zpracována specifikovaným uživatelským kódem.

1. Seznamte se s uživatelskými požadavky.
2. Na základě analýzy vyberte technologie implementace.
3. Pomocí metod softwarového inženýrství analyzujte a navrhněte aplikaci.
4. Implementujte takový systém.
5. Vytvořte webového klienta pro systém.
6. Aplikaci otestujte a nasaďte.

Seznam odborné literatury

Dodá vedoucí práce.

Ing. Michal Valenta, Ph.D.
vedoucí katedry

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

V Praze dne 10. července 2018

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Diplomová práce

Systém pro správu neuronových sítí

Bc. Michal Stejskal

Katedra softwarového inženýrství
Vedoucí práce: Ing. Josef Vogel, CSc.

6. května 2019

Poděkování

Tímto bych rád upřímně poděkoval svému vedoucímu, panu Ing. Josefu Vogelovi, CSc. za rady a konzultace, které mi věnoval při psaní této diplomové práce. Děkuji také své rodině za podporu během celých mých studií.

Prohlášení

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

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

V Praze dne 6. května 2019 .....

Odkaz na tuto práci

Stejskal, Michal. *Systém pro správu neuronových sítí*. Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2019.

Abstrakt

Tato diplomová práce se zabývá návrhem a vývojem systému pro automatizované nasazení neuronových sítí. Cílem této práce je vytvořit systém, který na základě uživatelem předaných dat naučí model neuronové sítě, automatizovaně jej nasadí do Kubernetes klastru a vystaví API pro komunikaci s uživatelem. Aby měl uživatel možnost přidat další funkcionalitu k již vytvořeným modelům neuronových sítí, nabízí systém možnost vytvořit uživatelské moduly, jejichž kód zadá uživatel v rámci klientské aplikace. Tyto moduly jsou propojeny s modelem neuronové sítě skrze třídy klasifikace.

První část práce je zaměřena na popis neuronových sítí, jejich proces učení a vybraných architektur.

V druhé části je popsána architektura tohoto systému, příkladové případy užití, požadavky na tento systém, popis uživatelů systému a zvolené metody strojového učení.

Na základě výsledků druhé části je ve třetí části popsán proces implementace jednotlivých komponent systému, rozhraní jejich komunikace a popis implementace a měření zvolených neuronových sítí. Vyvinutými komponentami jsou REST Kotlin API, které zpracovává požadavky klientské aplikace a komunikuje s Kubernetes klastrem. Dále tato komponenta obsluhuje rovněž vyvinuté REST Kotlin API, které vytváří Docker obrazy z uživatelem zadaného kódu. Dále byla vyvinuta klientská aplikace sloužící k interakci s uživatelem a zpracování výsledků vrácených REST API a tři modely neuronových sítí pro zpracování obrázků, klasifikaci log záznamů a zpracování psaného textu pro potřeby informačních chatbotů. Poslední vyvinutou komponentou je šablona v jazyce Python, která je spojena s uživatelským zdrojovým kódem.

Celý systém byl otestován a popis průběhu testování společně se zvolenými testovacími metodami jsou popsány ve čtvrté kapitole.

Proces nasazení systému do produkčního či testovacího prostředí je popsán v páté kapitole. V této kapitole je popsána kontejnerizace jednotlivých komponent systému nástrojem Docker a propojení jejich komunikace.

V závěru je diskutována budoucnost systému a další možná rozšíření.

V příloze této diplomové práce se nachází diskuze nad technologiemi automatického strojového učení a jejich následná implementace do vzniklého systému.

Klíčová slova neuronové sítě, Kubernetes, Kotlin, REST, Angular, automatické nasazení, správa neuronových sítí

Abstract

This master thesis deals with the design and development of a system for automated deployment of neural networks. The aim of this work is to create a system that train a neural network model based on user-submitted data, deploys it automatically to the Kubernetes cluster and build an API for communication with the user. In order to allow the user to add additional functionality to the neural network models which already created, the system offers the possibility to create user modules whose source code is entered by the user within the front end application. Those modules are linked to the neural network model through classification classes.

The first part is focused on the description of neural networks, their learning and architecture.

The second part describes the architecture of this system, examples of use, requirements for this system, description of system users and selected methods of machine learning.

Based on the results from the second part, the third part describes the process of implementation of the individual components of the system, the interface of their communication and the description of the implementation of measurement of selected neural networks. The developed components are REST Kotlin API, which handles client application requests, communicates with the Kubernetes cluster and also handles the developed REST Kotlin API, which creates Docker images from the user-specified code. In addition, a client application was developed to interact with the user and process the returned REST API results, and three models of neural networks for image processing, log record classification, and written test processing for information chatbots. The

latest developed component is the Python template, which is associated with the user's source code.

The entire system has been tested and a description of the testing process and the selected test method, both automatic and manual testing, are described in Chapter Four.

The process of deploying the system to the production or test environment is described in Chapter 5. This chapter describes the containerization of individual system components using the Docker tool and their communication links.

In the end, the future of the system and other possible extensions are discussed.

In the appendix of this thesis there is a discussion about the technologies of automatic machine learning and their implementation into the system.

Keywords neural network, Kubernetes, Kotlin, REST, Angular, automatic deployment, neural network management

Obsah

Seznam obrázků

Úvod

Motivace práce

Metody strojového učení dostávají v dnešní době stále více prostoru při řešení nejrůznějších problémů. Díky velkému výkonu dnešních počítačů se velmi rozšířily možnosti aplikace těchto modelů na reálné problémy. V dnešní době mohou modely strojového učení běžet i na mobilních zařízeních s omezeným výkonem. Nevýhodou implementace těchto modelů je často uživatelova neznalost problémů zpracování dat a strojového učení obecně. Cílem této práce je navrhnout a implementovat systém, který poskytne uživateli jednoduché rozhraní jak přinést metody strojového učení do svého života. Nově vzniklý systém Chobot umožní uživateli vytvořit model neuronové sítě tak, že definuje jaký typ problému chce pomocí metod strojového učení řešit a předá vstupní data. Následně mu bude vytvořena samostatná instance naučeného modelu, který bude mít vystavené rozhraní pro komunikaci. Jelikož samotná klasifikace vstupních dat občas nemusí uživateli stačit, bude tento systém také nabízet vytvoření uživatelských modulů ze zadaného zdrojového kódu. Uživatelský modul je samostatná aplikace nasazená v rámci systému Chobot, která v sobě obsahuje uživatelem zadaný zdrojový kód určující jeho chování. Moduly budou navázány na určitou třídu klasifikace jedné instance klasifikátoru a v okamžiku, kdy uživatel odešle požadavek na daný klasifikátor, a tento požadavek bude oklasifikován zmíněnou třídou klasifikace, bude uživatelský modul notifikován o zpracování odpovědí sítě. Výsledná data budou předána zpět uživateli.

Využití popsaného systému je mnoho. Příkladem mohou být obchody využívající klasifikátory obrázků pro kategorizaci produktů společně s chatbotem informující zákazníka o stavu jeho objednávky. Veškeré procesy obchodu mohou být prohledávány analyzátorem log záznamů a administrátor obchodu je informován o jakémkoliv abnormálním chování aplikace obchodu.

Cíle práce

Cílem této práce je navrhnout, implementovat a otestovat systém, který umožní uživatelům automatizovaně vytvářet a nasazovat modely strojového učení, konkrétně modely neuronových sítí. Předpokládaným vstupem od uživatele jsou pouze vstupní data, která následně poslouží pro trénování modelů neuronových sítí – klasifikátorů. Systém bude podporovat tři možnosti klasifikace vstupních dat a těmi jsou:

- Klasifikace log záznamů pro účely odhalení abnormálních událostí ve sledované aplikaci.
- Klasifikace obrázků pro možnost zjistit, jaké objekty se na obrázcích nacházejí.
- Klasifikace vstupního textu pro účely jednoduchých informačních chatbotů.

Pro vytvoření systému je potřeba zmapovat vhodné technologie strojového učení pro potřeby klasifikátorů, provést analýzu vhodného prostředí pro jejich automatické nasazení a životní cyklus. Dále je plánováno umožnit uživateli přidat funkcionalitu navázanou na jednotlivé třídy klasifikace pomocí modulů, které budou vykonávat aditivní kód zadaný uživatelem. Tyto moduly budou notifikovány z jednotlivých instancí klasifikátorů a společně se vstupními daty dostanou právě třídu klasifikace od modelu strojového učení.

Celý takto vzniklý systém je následně potřeba otestovat, nasadit a určit jakým směrem se vydá jeho budoucí vývoj.

Struktura práce

Tato diplomová práce je rozdělena do šesti různých kapitol. Každá kapitola popisuje určitou fázi vývoje celého systému a jsou v ní popsány využité postupy a techniky. Dále je v příloze Automatické strojové učení diskutováno automatizované strojové učení a jeho implementace do systému pro větší personalizaci výsledků.

Kapitola Úvod do neuronových sítí popisuje teoretický základ neuronových sítí, jejich proces učení a vybrané architektury.

Kapitola Analýza a návrh se zabývá jednotlivými požadavky na tento systém, které vznikly jako výsledek studia systémů automatizovaného nasazení kontejnerizovaných aplikací a neuronových sítí. Dále se zabývá architekturou celého systému, funkcionalitou jeho komponent a jejich zabezpečením. V této kapitole jsou také popsány procesy nově vzniklého systému a případy užití.

Kapitola Realizace popisuje proces vývoje systému Chobot. Jednotlivé podkapitoly popisují použité techniky a ukazují důležité části zdrojového kódu použité při vývoji komponent tohoto systému. Dále jsou v této kapitole uvedeny postupy vývoje klasifikačních neuronových sítí a měření jejich výsledků.

Kapitola Testování systému shrnuje testovací techniky použité při testování jednotlivých komponent systému. Nejdříve je diskutováno automatické testování a následně i manuální testování. Další část této kapitoly se zabývá statickou analýzou zdrojových kódů a jejími výsledky. V závěru jsou shrnuty výsledky tohoto testování.

V kapitole Nasazení je popsán postup nasazení systému v testovacím nebo produkčním prostředí.

V kapitole Budoucnost systému a další rozvoj je popsán plánovaný rozvoj systému, jeho rozšíření a přidání nových technik klasifikačních neuronových sítí a jejich personalizace vůči uživatelským datovým sadám.

Příloha Automatické strojové učení se zabývá technikami automatizovaného strojového učení, jejich popisem a vysvětluje proces zapojení těchto technik do systému Chobot.

Kapitola 1

Úvod do neuronových sítí

1.1 Neuronové sítě

Koncept neuronových sítí vznikl na základě inspirace lidským mozkem. Standardní algoritmický přístup definuje chování aplikace na základě předem naprogramovaných instrukcí a rozhodování na základě podmínek. Pokud tedy přijde vstup, na který algoritmus není naprogramován vrátí většinou předem danou výchozí hodnotu. Naproti tomu neuronové sítě fungují na podobném principu jako lidský mozek, kdy jsou jednotlivé neurony sítě učeny na základě vstupních dat a učí se abstrahovat dané vstupy podobně jako je tomu právě u lidského mozku. Neuronová síť obsahuje velký počet neuronů, které jsou spolu propojené do určité topologie a každý neuron reaguje na určitý vstup jinak.

1.1.1 Neuron

Neuron[1] je základní stavební jednotka sítě. Na základě jeho aktivace prochází vstupní data do dalších vrstev sítě k další neuronům. Každý neuron se skládá z několika částí. Konkrétně se jedná o:

- Vstupní hrany neuronu x_k, na kterých přijímá vstup od ostatních neuronů. Vstup je většinou reprezentován v podobě číselných hodnot. Pokud je neuron uvnitř vstupní vrstvy, přijímá na vstupních hranách x_k trénovací data.

- Synaptické váhy vstupních hran w_k, které násobí hodnoty vstupních hran x_k a tím zesilují vstup z dané hrany. Váhy mohou nabývat jak kladných hodnot, pro vstupy které je potřeba posílit, tak i záporných hodnot pro vstupy, které je potřeba určitým způsobem omezit. Čím vyšší je hodnota váhy dané vstupní hrany, tím je tento vstup důležitější. Jednotlivé váhy reprezentují informace, které se neuron naučil.

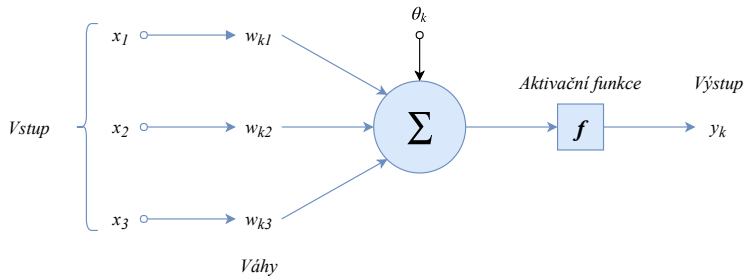
- Agregační funkce, která sečte násobky hodnot všech příchozích hran x_k s jejich synaptickými váhami w_k

$$\sum_{k=1}^{n} w_k \cdot x_k$$

  kde n je počet příchozích hran neuronu.

- Prahová hodnota θ, která je porovnávána s výsledkem agregační funkce. Pokud výsledek agregační funkce vyšší než prahová hodnota, dojde k aktivaci neuronu.

- Aktivační přenosová funkce f, která v případě aktivace neuronu transformuje vnitřní potenciál neuronu společně s prahovou hodnotou a tím získá výstupní hodnotu neuronu y_k.



Obrázek 1.1: Struktura neuronu

Tento popis neuronu není jediný používaný koncept. Jiné typy neuronových sítí využívají jiné koncepty neuronů. Například RBF sítě, kde je neuron použit k výpočtu vzdálenosti mezi vstupním vektorem a vektorem naučených vah.

1.1.2 Aktivační funkce neuronu

Aktivační funkce neuronu slouží k určení jeho výstupní hodnoty[2]. Aktivační funkce se nejčastěji používají u dopředných neuronových sítí, kde transformují vnitřní potenciál neuronu na reálnou hodnotu. Obecně lze říci, že aktivační funkce provádí zobrazení $R^n \rightarrow R$.

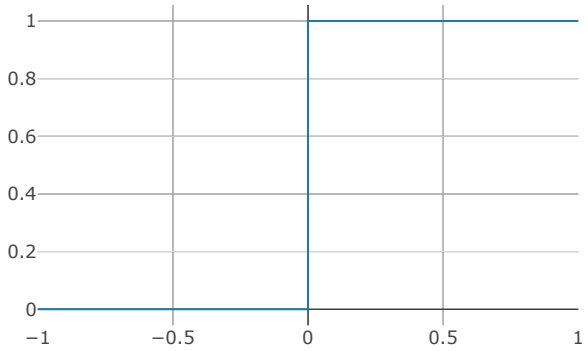
1.1.2.1 Skoková aktivační funkce

Skoková aktivační funkce pracuje na jednoduchém základu. Tato funkce má nastavenou mez a porovnává hodnotu vnitřního potenciálu neuronu s touto mezí. Pokud je mez vyšší než hodnota potenciálu, funkce vrátí nulu. Pokud je

mez nižší, funkce vrátí jedna. Vyjádřit skokovou funkci f s nastavenou mezí h lze následujícím vztahem:

$$f(x) = \begin{cases} 1, \text{ pro x} \geq \text{h} \\ 0, \text{ pro x} < \text{h} \end{cases}$$

Graf skokové funkce je na obrázku 1.2.



Obrázek 1.2: Graf skokové aktivační funkce

1.1.2.2 Sigmoidní aktivační funkce

Sigmoidní aktivační funkce je speciálním druhem logistické funkce. Její vzorec je vyjádřen ve tvaru

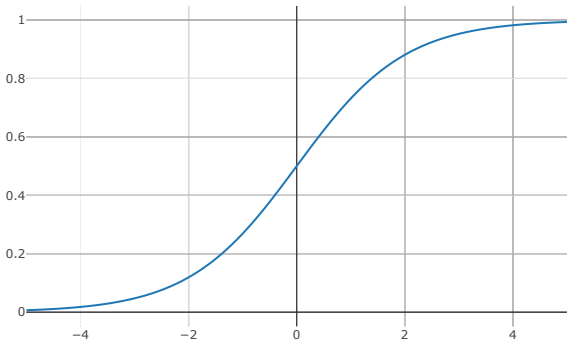
$$f(x) = \frac{1}{1 + e^{-x}}$$

Výhodou tohoto druhu aktivační funkce je, že v každém jejím bodě existuje její první spojitá derivace. Tato funkce se přibližuje v $-\infty$ k 0 a v ∞ k 1. Hodnota funkce v 0 je $0,5$. Tato funkce se hodí například při klasifikaci dvou tříd.

Graf sigmoidní aktivační funkce je na obrázku 1.3.

1.1.2.3 Softmax aktivační funkce

Dalším typem aktivační funkce je Softmax. Tato funkce transformuje vstupní vektor x dimenze N na výstupní vektor y dimenze N, jehož hodnoty jsou na intervalu $(0, 1)$. Součet prvků tohoto vektoru je roven 1 a hodnoty tohoto vektoru přestavují pravděpodobnosti, že určitý vstup patří do určité klasifikační třídy. Vzorec této aktivační funkce je vyjádřen ve tvaru



Obrázek 1.3: Graf sigmoidní aktivační funkce

$$f(x)_k = \frac{e^{x_k}}{\sum_{j=1}^{K} e^{x_j}}$$

kde K reprezentuje počet klasifikačních tříd.

1.1.2.4 Aktivační funkce hyperbolického tangentu

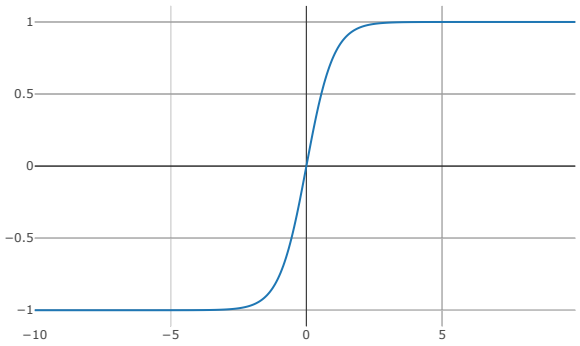
Hyperbolický tangens je hyperbolická funkce definovaná pomocí hyperbolických funkcí *sinh* a *cosh*. Hodnota této funkce se v $-\infty$ blíží k -1 a v ∞ k 1. V 0 tato funkce nabývá 0. Tato funkce lze zapsat pomocí tvaru

$$f(x) = \frac{1}{1 + e^{-kx}}$$

a její graf je na obrázku 1.4.

1.1.3 Učení neuronové sítě

Během učení neuronové sítě dochází k nastavení parametrů jednotlivých neuronů tak, aby daný neuron prováděl požadovanou transformaci a síť jako celek podávala co nejpřesnější výsledky. Během učení dochází zpravidla k aktualizacím hodnot jednotlivých vah vstupních hran neuronu a k úpravě jeho prahové hodnoty. Častou učící metodou je algoritmus zpětného šíření chyby (back propagation) ve spojení s klesáním podle gradientu (gradient descent). Učící algoritmus probíhá iterativně a většinou je učení rozděleno do několika částí, epoch. Rozdělení učícího procesu na epochy umožňuje ukládání záchytných bodů, ke kterým se lze vrátit v případě chyby během učení, nebo



Obrázek 1.4: Graf aktivační funkce hyperbolického tangentu

lze na konci procesu vybrat ten záchytný bod, ve kterém dávala síť nejlepší výsledky. Každá epocha obsahuje několik iterací, ve kterých jsou vstupní data předkládána síti a ta se z nich učí. Velikost množiny vstupních dat záleží na povaze řešeného problému, v určitých případech se síti v každé epoše předloží všechna vstupní data, v jiném případě jejich podmnožina či pouze jeden záznam.

1.1.3.1 Ztrátová funkce

Ztrátová funkce vyjadřuje rozdíl mezi výsledkem vráceným neuronovou sítí a skutečným výsledkem. Cílem učícího algoritmu je minimalizovat tento rozdíl. Mezi nejvíce používané ztrátové funkce se řadí funkce střední kvadratické chyby (Mean squared error), která je vyjádřena pomocí vzorce

$$E = \frac{1}{n} \cdot \sum_{y=1}^{n} (\widehat{y}_i - y_i)^2$$

kde y reprezentuje reálný výstupní vektor a $\widehat{y}$ reprezentuje výstupní vektor sítě pro n vstupů[3].

1.1.3.2 Algoritmus zpětného šíření chyby

Algoritmus zpětného šíření chyby slouží k propagaci chyb výstupní vrstvy k neuronům, které jsou v nižších vrstvách a úpravám hodnot jejich vah. Algoritmus nejdříve dopředně šíří vstupní data skrze jednotlivé vrstvy sítě[1]. Následně dojde k porovnání reálného výsledku s výsledkem, který spočítala síť. Na základě rozdílu těchto dvou výsledků dojde k úpravě vah výstupních

neuronů a jejich předání neuronům v nižší vrstvě. Neurony v této nižší vrstvě opět spočítají svoji chybu a předají ji neuronům v další nižší vrstvě. Tento proces je opakován až po vstupní vrstvu. Následně jsou upraveny váhy vstupů jednotlivých neuronů, kdy je ke vstupním datům a jejich váhám přičten určitý k násobek chyby. Pseudokód algoritmu je následující.

- Inicializace vah jednotlivých neuronů náhodnými čísly.
- Opakování následujících kroků až do splnění podmínky zastavení učení modelu (např. počet iterací, či konvergence ztrátové funkce).
- Pro každé $[x, y]$ ze vstupních trénovacích dat:
  1. Výpočet hodnoty out_u pro každý neuron u v síti.
  2. Pro každý neuron v ve výstupní vrstvě je spočítána modifikační chyba pomocí vzorce
  $$\Delta_v = out_v \cdot (1 - out_v) \cdot (y - out_v).$$
  3. Pro každý neuron s ve skrytých vrstvách je spočítána chyba pomocí vzorce
  $$\Delta_s = out_s \cdot (1 - out_s) \cdot \sum_{v \in vystup} (w_{s,v} \cdot \Delta_v).$$
  4. Každé vazbě propojující neuron j a neuron k je aktualizována váha
  $$w_{j,k} = w_{j,k} \cdot \Delta w_{j,k} \quad \text{kde } \Delta w_{j,k} = \eta \cdot \Delta_k \cdot out_j$$
  kde η značí tzv. míru učení, pomocí které se určuje rychlost učení.

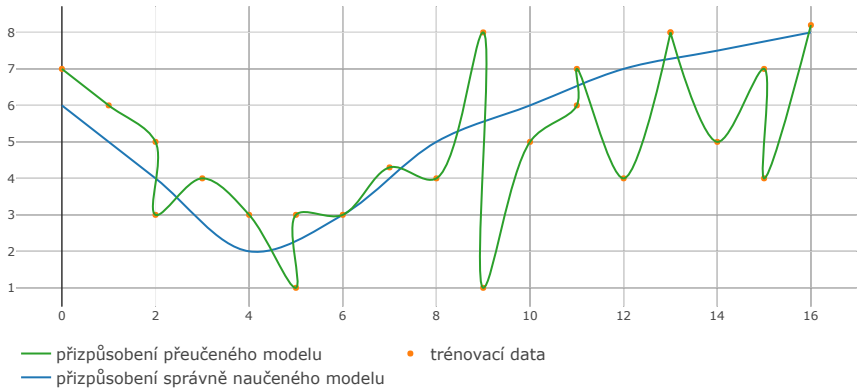
1.1.3.3 Klesání podle gradientu

Během učení sítě je potřeba upravovat hodnoty vah neuronů tak, aby síť vracela co nejpřesnější výsledky a hodnota ztrátové funkce byla co možná nejnižší. Optimalizační algoritmus klesání podle gradientu (gradient descent)[4] hledá směr nejvyššího růstu ztrátové funkce a poté směřuje přesně opačným směrem. Tím se dostane do lokálního minima ztrátové funkce. Při použití tohoto algoritmu je potřeba určit tzv. *krok učení*. Krok učení určuje jak velký má být posun v opačném směru gradientu. Pokud je krok učení příliš velký, může nastat situace, kdy se ztrátová funkce nedostane do svého lokálního minima, protože je toto minimum přeskočeno. Pokud je naopak krok učení příliš malý, proces učení trvá velmi dlouho. Nalezení lokálního minima ztrátové funkce však není požadovaný výsledek optimalizačního procesu. Aby nedocházelo k uváznutí v lokálním minimu, využívá se algoritmu stochastického klesání podle gradientu (stochastic gradient descent). Ten k úpravám vah nevyužívá všechna vstupní data, ale pouze aktuální trénovací vstup. Dalším variantou tohoto optimalizačního algoritmu je algoritmus klesání gradientu v podmnožině dat (mimi batch gradient descent), který k úpravám vah využívá podmnožinu vstupních dat.

1.2 Typy učení neuronových sítí

První kategorií učení neuronové sítě je učení s učitelem, kdy síť dostane jako vstup dvojici *[vstup, požadovaný výstup]*. Cílem učícího algoritmu je nastavit váhy jednotlivých neuronů na základě chyby spočítané pomocí algoritmu zpětného šíření chyby tak, aby byla chyba minimální a zároveň byla síť pořád schopna generalizace. Pokud je trénovacích dat málo a neuronová síť se během učení perfektně přizpůsobí těmto datům, dochází obvykle k jejímu přeučení. Při přeučení dokazuje model téměř dokonalé výsledky na množině trénovacích dat, ale pokud dostane data, která nebyla součástí trénovacích dat, jsou jeho výsledky velmi chybové.

Přeučení lze zabránit několika způsoby. Tím prvním je zvětšení trénovací množiny dat. Druhým způsobem je využití metody mazání přebytečných neuronů (dropout), která náhodně odebírá neurony z architektury neuronové sítě. Další možným způsobem je využití testovací (validační) množiny dat, na které se během trénovaní ověřuje chybovost modelu.



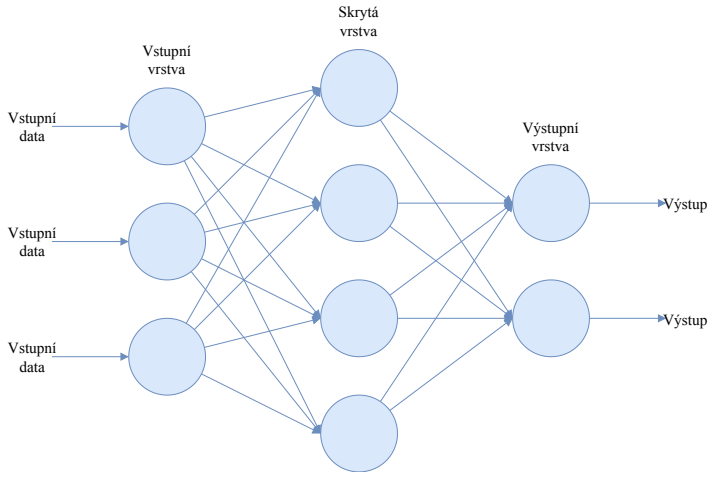
Obrázek 1.5: Přeučení modelu

Druhou kategorii představuje učení bez učitele, kdy neuronová síť dostane pouze vstupní data a jejím úkolem je nalézt nové informace a propojení v těchto datech. Učení bez učitele se často využívá při řešení problému shlukování, kdy síť hledá podobnosti ve vstupních datech a třídí je do kategorií.

1.3 Architektura neuronových sítí

1.3.1 Dopředné neuronové sítě

Dopředné neuronové[5] sítě se skládají z několika vrstev, ve kterých jsou neurony nižších vrstev propojeny s neurony z vyšších vrstev. Propojení mezi neurony je orientované pouze do vyšších vrstev, nikdy ne jinak. Takováto síť se skládá minimálně ze dvou vrstev, kdy první vrstva slouží k přijetí vstupních dat a druhá vrstva slouží jako vrstva výstupní. Každá další vrstva přidaná mezi tyto dvě vrstvy tvoří skrytou vrstvu. Čím více síť obsahuje skrytých vrstev, tím složitější problémy dokáže řešit.



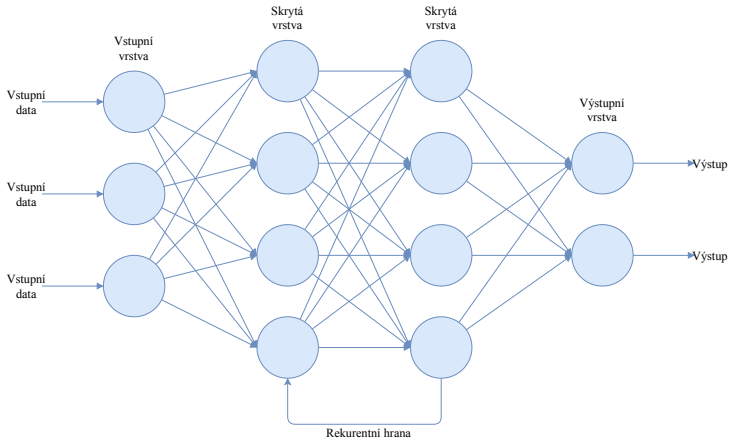
Obrázek 1.6: Architektura dopředné sítě

1.3.2 Rekurentní neuronové sítě

Rekurentní sítě[6] obsahují navíc na rozdíl od dopředných sítí alespoň jednu hranu, která vede zpětně z vyšší vrstvy do nižší. Síť tedy obsahuje smyčku. Díky těmto zpětným hranám je síť schopná uchovávat informace z předchozích vstupů a tím držet kontext. Díky kontextu dokáží rekurentní sítě řešit problémy závislé na čase jako je například překlad textu. Architektura rekurentní neuronové sítě je uvedena na obrázku 1.7.

1.3.3 Konvoluční neuronové sítě

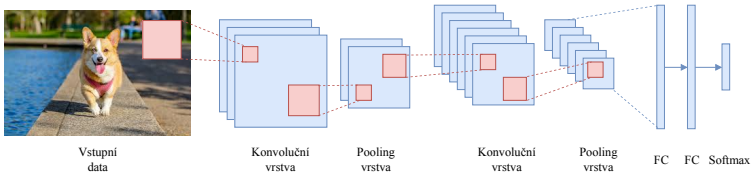
Neurony dopředné neuronové sítě se během procesu učení učí samostatně, každý zvlášť. To je dostačující pro řešení mnoha druhů problému ale při klasifikaci obrázků lze velmi jednoduše model s dopřednou architekturou zmást. Obecně stačí obrázek pootočit či posunout a model jej nemusí klasifikovat



Obrázek 1.7: Architektura rekurentní sítě

správně. Tento problém se dá řešit tím způsobem, že během učení jsou modelu předkládány i posunuté či otočené obrázky. Tento způsob sice zvyšuje robustnost modelu, ale také se tímto způsobem zvyšuje počet neuronů v každé vrstvě sítě[4].

Konvoluční neuronové sítě jsou dopředné sítě, kde se během učení jednotlivé neurony ovlivňují a sdílejí své váhy[7]. Tyto sítě bývají obecně hluboké a mají desítky až stovky vrstev. Stejně jako standardní dopředné architektury sítí mají konvoluční neuronové sítě vstupní, výstupní a skryté vrstvy. K těmto vrstvám se ale přidávají konvoluční a podvzorkovací (subsampling pooling) vrstvy. Konvoluční vrstvy jsou napojené buďto přímo na vstupní vrstvu, nebo právě na podvzorkovací vrstvu, kdy je podvzorkovací vrstva vložena mezi dvě konvoluční vrstvy. Úkolem těchto vrstev je získání důležitých atributů z obrázku a slouží poté jako extraktor příznaků (feature extractor). Tyto příznaky jsou následně předány standardním skrytým vrstvám, které mají za úkol samotnou klasifikaci.

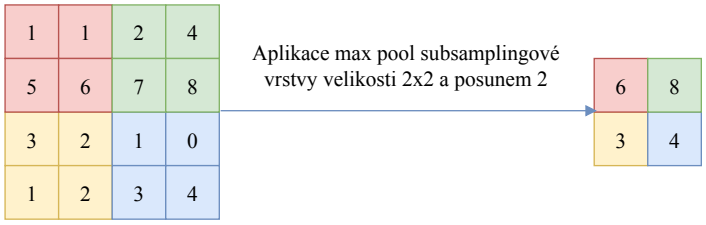


Obrázek 1.8: Konvoluční síť

1.3.3.1 Podvzorkovací vrstva

Podvzorkovací vrstvy (subsampling layer) jsou přímo napojeny na konvoluční vrstvy a berou jejich výstupní data jako vstupní. Tyto vrstvy pracují velmi

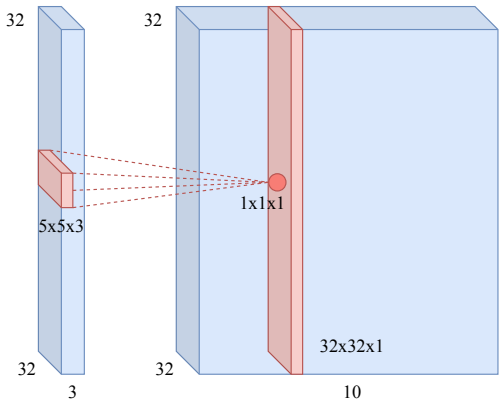
rychle, protože ve většině případů provádějí algoritmicky nenáročné operace. Příkladem mohou být velmi často používané tzv. *max pooling*, kdy je na vstupní data velikosti $N \times N$ aplikován filtr velikosti $M \times M$, kde $N > M$. Tento filtr začne na pozici $[0, 0]$ a vybere maximální hodnotu obsaženou v submatici velikosti filtru. Následně dojde k posunu o k pozic (stride). Po posunu je opět vybrána nejvyšší hodnota a filtr se opět posune. Tímto způsobem je dimenze vstupních dat výrazně redukována.



Obrázek 1.9: Max pooling podvzorkovací vrstva

1.3.3.2 Konvoluční vrstva a konvoluční filtry

Konvoluční vrstvy slouží k detekci objektů v rámci obrázku. Nejnižší konvoluční vrstva v rámci architektury sítě může sloužit například k detekci hran, hlubší konvoluční vrstvy mohou posloužit k detekci složitějšího vzorce (například kruhů). Čím je konvoluční vrstva hlouběji, tím složitější vzorce dokáže detekovat[8]. Konvoluční vrstvy procházejí vstupní data a aplikují na ně konvoluční filtry. Během aplikace filtru je vypočítána konvoluce mezi vstupními daty a daným konvolučním filtrem a výsledek je uložen do příznakové mapy. Jednotlivé aplikace filtrů se částečně překrývají a vrstva se během trénování učí jaký konvoluční filtr použít na určitý typ vstupních dat.



Obrázek 1.10: Konvoluční vrstva

Kapitola 2

Analýza a návrh

Tato kapitola popisuje požadavky, které vznikly během studia systému pro nasazení a běh neuronových sítí a dalších klasifikačních modelů. Požadavky popsané v této kapitole jsou důležité pro správný návrh celého systému, jeho architektury a samostatných komponent. Každá komponenta systému Chobot má svoji jedinečnou funkcionalitu, proto jsou i seznamy požadavků rozděleny na podkapitoly, právě podle komponent systému. Jednotlivé komponenty systému popsané v této kapitole mají své specifické požadavky na použité technologie. Proto jsou v této kapitole také popsány vybrané technologie zvolené během studia požadavků a proveditelnosti implementace vzniklého systému.

Dále jsou v této kapitole popsány zvolené modely klasifikátorů a jejich architektury, princip jejich učení a zabezpečení celého systému. Na konci kapitoly se nachází popis procesů systému a vybrané případy užití.

2.1 Analýza požadavků pro systém

2.1.1 Klientská aplikace

Klientská aplikace představuje hlavní rozhraní pro uživatele. Uživatel v rámci této komponenty vytváří nové instance klasifikátorů a definuje funkcionalitu modulů.

Funkční požadavky

- Klientská aplikace umožní vytvoření nové instance klasifikátoru.
- Klientská aplikace umožní předat vstupní trénovací a další data klasifikátoru, pokud jsou tato data potřeba pro trénování.
- Klientská aplikace umožní vytvoření nového modulu ze zdrojového kódu zadaného do online editoru.
- Klientská aplikace umožní vytvoření nového modulu z Git repozitáře.

- Klientská aplikace umožní propojení modulu s instancí klasifikátoru na základě třídy klasifikace.
- Klientská aplikace umožní nasadit či případně zrušit nasazení instance klasifikátoru a modulů do systému pro automatizované nasazení.
- Klientská aplikace umožní zobrazení log záznamů, přístupových údajů a stavových informací o nasazených klasifikátorech a modulech.

Nefunkční požadavky

- Klientská aplikace bude mít jednoduché a rychlé rozhraní.
- Klientskou aplikaci bude možné samostatně nasadit.

2.1.2 Aplikace pro vytváření nových instancí klasifikátorů a modulů – kontejnerové API

Kontejnerové API je hlavní aplikace celého systému. Slouží pro propojení všech dalších komponent systému a jejich administraci. Dále obsluhuje klientskou aplikaci, systém pro automatizované nasazení a obrazové API.

Funkční požadavky

- Rozhraní umožní vytvářet nové instance klasifikátorů. Typ klasifikátoru bude definován v rámci přijatého požadavku na vytvoření.
- Rozhraní bude přijímat trénovací data a další potřebné parametry pro trénování klasifikátoru a jeho běh.
- Rozhraní umožní nasadit vytvořené klasifikátory do systému pro automatizované nasazení.
- Rozhraní umožní zrušení nasazení klasifikátoru ze systému pro automatizované nasazení.
- Rozhraní předá data o nasazených klasifikátorech jako jsou přístupové údaje pro volání jejich API, URL kontrolerů nebo aplikační log záznamy klientské aplikaci.
- Rozhraní umožní vytvářet nové uživatelské moduly.
- Rozhraní bude přijímat typ modulu.
- Rozhraní umožní propojení modulu s klasifikátorem při vytvoření modulu.
- Rozhraní umožní nasadit vytvořené moduly do systému pro automatizované nasazení.

- Rozhraní umožní zrušení nasazení modulu ze systému pro automatizované nasazení.
- Rozhraní předá data o nasazených modulech jako jsou přístupové údaje pro volání jejich API, URL kontrolerů nebo aplikační log záznamy klientské aplikaci.
- Rozhraní bude přijímat a zpracovávat požadavky přijaté z klientské aplikace.
- Rozhraní bude komunikovat se systémem pro automatizované nasazení a pomocí programového API ho obsluhovat.
- Rozhraní bude komunikovat s API aplikace pro vytváření Docker obrazů a zadávat mu požadavky na vytvoření nových uživatelských modulů.

Nefunkční požadavky

- Rozhraní půjde jednoduše škálovat.
- Rozhraní půjde v případě potřeby rozšiřovat o další funkcionalitu.

2.1.3 Aplikace pro sestavení Docker obrazů uživatelských modulů – obrazové API

Obrazové API slouží k vytvoření nového Docker obrazu ze zdrojových kódů zadaných uživatelem. Nově vytvořený Docker obraz je následně využit při nasazení uživatelského modulu.

Funkční požadavky

- Aplikace bude přijímat požadavky od kontejnerového API.
- Aplikace vytvoří nový Docker obraz modulu spojením předpřipravené šablony a uživatelského zdrojového kódu.
- Obrazové API vytvoří nový Docker obraz modulu z Git repozitáře.
- Obrazové API nahraje vytvořený Docker obraz do privátního nebo veřejného Docker repozitáře.

Nefunkční požadavky

- Rozhraní půjde jednoduše škálovat.
- Rozhraní půjde v případě potřeby rozšiřovat o další funkcionalitu.

2.1.4 Klasifikátory – společná funkcionalita

Jednotlivé klasifikátory slouží ke klasifikaci požadavků přijatých od uživatele. Jedná se o samostatně automatizovaně nasazené aplikace.

Funkční požadavky

- Klasifikátor bude přijímat požadavky na klasifikaci obrazů, log záznamů nebo zprávy pro chatboty.
- Klasifikátor bude vracet informace o stavu na specifikované adrese.
- Klasifikátor bude zasílat požadavky na připojené moduly a vracet jejich odpověď uživateli, či jiné volající aplikaci.

Nefunkční požadavky

- S klasifikátorem půjde komunikovat skrze REST API.
- Klasifikátor půjde jednoduše škálovat.
- Klasifikátor půjde nasadit do systému pro automatizované nasazení.
- Klasifikátor bude podporovat zabezpečené připojení.

2.1.5 Moduly

Uživatelské moduly slouží pro přidání další funkcionality navázané na konkrétní třídu klasifikace určitého klasifikátoru. Chování modulů definuje uživatel zadaným zdrojovým kódem. Jedná se o samostatně automatizovaně nasazené aplikace.

Funkční požadavky

- Modul bude přijímat požadavky od klasifikátoru.
- Modul bude vykonávat uživatelem zadaný kód a jeho odpověď vrátí klasifikátoru.
- Modul bude vracet informace o stavu na specifikované adrese.

Nefunkční požadavky

- Modul půjde jednoduše škálovat.
- Modul půjde nasadit do systému pro automatizované nasazení.
- Modul bude podporovat zabezpečené připojení.
- Zdrojový kód modulu bude v jazyce Python.

2.1.6 Systém pro automatizované nasazení

Systém pro automatizované nasazení slouží jako místo, kde jsou nasazeny jednotlivé instance klasifikátorů a uživatelských modulů.

Funkční požadavky

- Systém umožní nasazovat aplikace a konfigurovat jejich parametry.
- Systém umožní směrování požadavků na nasazené aplikace.

Nefunkční požadavky

- Systém půjde flexibilně škálovat podle počtu nasazených aplikací.
- Systém bude mít aplikační rozhraní podporující jazyk Java nebo Python.
- Systém bude pod svobodnou licencí.
- Systém umožní škálovat nasazené aplikace.

2.2 Analýza požadavků pro klasifikátory

Klasifikátory jsou velice důležitými komponentami v rámci celého systému Chobot. Hlavní myšlenkou tohoto systému je umožnit uživateli vytvořit klasifikátor s modelem, který mu pomůže řešit určitý problém, s co nejmenším úsilím a potřebnými znalostmi. Jediné co musí uživatel modelu předat jsou vstupní data, na kterých se model posléze začne učit. V rámci terminologie distribučních modelů, lze systém Chobot zařadit do kategorie Software jako služba, kde nabízeným softwarem je právě klasifikátor. Jednotlivé klasifikátory mají společnou funkcionalitu, která slouží pro komunikaci s koncovým uživatelem a pro komunikaci s ostatními komponentami systému. Dále klasifikátory obsahují specifickou funkcionalitu, model neuronové sítě, který slouží právě k řešení daného problému.

2.2.1 Analýza společné funkcionality

Každý klasifikátor má vystavené REST rozhraní, na kterém přijímá požadavky a komunikuje s ostatními komponentami systému. Rozhraní má tři hlavní kontrolery.

- Kontroler stavu (health controller), který slouží jak pro uživatele, tak pro ostatní komponenty systému. Tento kontroler vrací příznak, zdali běží instance klasifikátoru správně. Dále slouží jako kontrola, zdali již instance klasifikátoru nastartovala. Tento kontroler není nijak zabezpečený a je přístupný z internetu.

- Kontroler predikce na kterém klasifikátor přijímá vstupní data, která mají být ohodnocena či jinak zpracována. Tato data předá modelu a čeká na jeho odpověď. Po přijetí odpovědi ve formě třídy klasifikace prohledá všechny připojené moduly a pokud najde modul, který je spárovaný s danou třídou klasifikace, zašle požadavek právě tomuto modulu. Tento kontroler přijímá spolu s požadavkem JWT token, který slouží k autorizaci oprávnění.
- Dále každý klasifikátor podporuje kontroler pro vystavení Swagger dokumentace. Tento kontroler je využíván klientskou aplikací.

Všechny klasifikátory mají přístup k databázi, která obsahuje informace o trénovacích datech nebo jiných důležitých parametrech. Funkční a nefunkční požadavky na společnou funkcionalitu klasifikátorů jsou popsány v sekci Analýza požadavků pro systém.

2.2.2 Analýza klasifikátoru pro klasifikaci log záznamů

Klasifikátor log záznamů slouží k detekci abnormálních log záznamů. Jelikož je množství log záznamů enormní, bylo by pro uživatele náročné klasifikovat log záznamy jednotlivě pro potřeby trénovacích dat. Dále by zde hrozila možnost, že uživatel nedokáže manuálně nalézt všechny abnormality. Je tedy nutné, aby klasifikátor log záznamů pracoval na principu učení bez učitele.

Klasifikátor získá při startu aplikace z databáze cestu k trénovací datové sadě. Tato data nejdříve převede do potřebné formy a poté spustí trénování modelu. Po natrénování modelu uvnitř klasifikátoru začne přijímat uživatelské požadavky.

Funkční a nefunkční požadavky získané během analýzy klasifikátorů log záznamů jsou následující.

Funkční požadavky

- Klasifikátor připraví trénovací i vstupní data do požadované (číselné) formy pro model.
- Model uvnitř klasifikátoru přijme vstupní data a provede klasifikaci.
- V rámci přípravy dat klasifikátor nahradí číselné záznamy zástupnými znaky nebo je úplně odstraní.

Nefunkční požadavky

- Model použitý v klasifikátoru bude pracovat na principu shlukování na základě podobnosti společně s procesem učení bez učitele.
- Trénovací data pro model budou v textovém souboru se standardním formátem log záznamů.

2.2.3 Analýza klasifikátoru pro klasifikaci obrázků

Klasifikátor obrázků bude existovat ve dvou verzích. První verze klasifikátoru bude mít předtrénovaný model, který nebude pro svůj běh vyžadovat žádná trénovací data. Druhým typem klasifikátoru je model, který bude trénován na základě uživatelem předaných dat. Pro klasifikaci obrázků pomocí modelů neuronových sítí obecně platí, že pokud je potřeba jejich učení provádět od začátku, je potřeba velká množina trénovacích dat. Toto omezení se dá obejít pomocí přenosového učení (transfer learning), kdy je použit již natrénovaný klasifikátor, ke kterému se přidá další vrstva. Nová vrstva je poté přetrénována a klasifikátor ji využívá společně původními vrstvami.

Funkční a nefunkční požadavky na klasifikátor obrázků jsou následující.

Funkční požadavky

- Klasifikátor připraví trénovací i vstupní data do požadované (číselné) formy pro model.
- Model uvnitř klasifikátoru přijme vstupní data a provede klasifikaci.

Nefunkční požadavky

- Proces učení modelu použitého v klasifikátoru bude pracovat na principu učení s učitelem.
- Trénovací data pro model neuronové sítě uvnitř klasifikátoru budou v adresářové struktuře, kdy každý adresář bude obsahovat obrázky pro jednu třídu klasifikace. Jméno daného adresáře pak určuje pojmenování klasifikační třídy.
- Učení klasifikátoru bude pracovat na principu přenosového učení.

2.2.4 Analýza klasifikátoru pro využití chatbotů

Pro tvorbu chatbotů existuje mnoho již existujících řešení, která ve většině případů pracují na principu hledání klíčových slov. Další možností jak klasifikovat vstupní text a vracet připravené odpovědi, je klasifikace textu pomocí neuronových sítí. Síť dostane na vstup připravená data a naučí se jednotlivé vzory pro každou třídu klasifikace. Model tedy dostane pro každou třídu klasifikace tři množiny vstupních dat.

- Třídu klasifikace.
- Vstupní množinu textových řetězců, ze kterých se klasifikátor učí vzory pro danou třídu. Může se jednat o celé věty nebo pouze klíčová slova.

- Výstupní řetězce, které klasifikátor vrací uživateli. Tyto řetězce model v rámci učení nepoužívá.

Po naučení modelu je klasifikátor připraven přijímat požadavky uživatele. Po přijetí požadavku předá vstupní řetězec modelu, ten jej oklasifikuje a vrátí třídu klasifikace. Klasifikátor následně podle třídy zvolí náhodnou odpověď z množiny výstupních řetězců pro danou třídu.

Funkční a nefunkční požadavky na chatbota jsou následující.

Funkční požadavky

- Klasifikátor připraví trénovací i vstupní data do požadované (číselné) formy pro model neuronové sítě.
- Model uvnitř klasifikátoru přijme vstupní data a provede klasifikaci.
- Klasifikátor přijme třídu klasifikace a náhodně vybere odpověď z připravené množiny odpovědí pro danou třídu.

Nefunkční požadavky

- Proces učení modelu použitého v klasifikátoru bude pracovat na principu učení s učitelem.
- Vstupní data pro model uvnitř klasifikátoru budou ve formátu JSON.

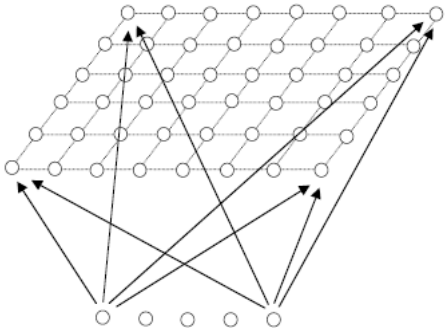
2.3 Zvolené modely klasifikátorů

2.3.1 Klasifikátor log záznamů – samoorganizující se mapy

Tyto neuronové sítě nepotřebují ideální vzor pro učení, jedná se tedy o učení bez učitele. Samoorganizující se mapy (SOM) fungují na principu shlukování objektů s podobnými vlastnostmi[9].

SOM je obvykle dvouvrstvá neuronová síť kde vstupní vrstva obsahuje neurony, které jsou úplně propojeny s kompetiční vrstvou. Kompetiční vrstva se skládá z uspořádané struktury neuronů buďto v podobě čtvercové či obdélníkové matice, nebo v podobě hexagonálního útvaru. Jednotlivé neurony ovlivňují učící proces svých sousedů.

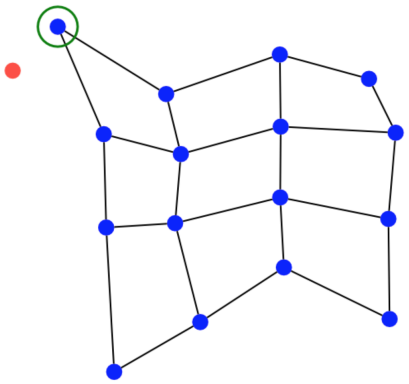
Topologie sítě je velmi propojena s procesem učení. Učení probíhá iterativně, kdy je v každém kroku náhodně vybrán jeden vzor ze vstupních dat. Na základě tohoto vzoru je vybrán konkrétní neuron, jehož vektor vah je nejvíce podobný (nebo stejný) vstupnímu vzoru. Takovému neuronu se říká *vítězný neuron*. Váhy vítězného neuronu jsou posunuty blíže k vybranému vzoru. Stejně tak je topologické okolí vítězného neuronu posunuto blíže ke vzoru. Čím více je neuron vzdálen od vítězného neuronu, tím méně je posouván (nejvíce je posunut vítězný neuron). Počet kroků učícího algoritmu je pevně daný.



Obrázek 2.1: Struktura samoorganizující se neuronové sítě

2.3.1.1 Konkrétní popis SOM

Samoorganizující se neuronová síť je tvořená konečným počtem neuronů a topologií. Jednotlivé neurony jsou indexovány čísly $1 \ldots m$ kde m je počet neuronů v kompetiční vrstvě. Váha každého neuronu je značena jako w_i. Na obrázku 2.2 je zobrazena kompetiční vrstva neuronové sítě 4×4. Její neurony jsou zobrazeny modrou barvou. Topologie neuronů je naznačena pomocí černých čar. Spojené jsou jen ty neurony, které jsou si nejblíže vzhledem k topologické vzdálenosti, naznačuje se tak uspořádání sítě. Dále je na obrázku vyznačen červený bod. Ten reprezentuje vstupní vzor. Pro tento vzor se zjišťuje, který neuron má nejblíže vektor vah právě tomuto vstupnímu vzoru. Ten neuron, který má dle metriky nejblíže je nazván vítězným neuronem. Na obrázku je označen zelenou barvou. Pro každý vstup tedy metoda **winner()** vrátí index vítězného neuronu.



Obrázek 2.2: Učící proces neuronové sítě

SOM jsou nejčastěji používány k nalezení podobností v datech, kdy jsou vstupní vektory, které se liší jenom v malém množství složek, označeny za podobné a existuje předpoklad, že se v síti budou vyskytovat na stejném

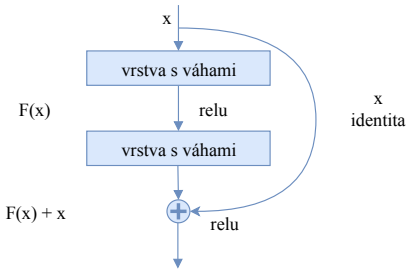
neuronu, nebo že vzdálenost mezi nimi bude velmi malá. Během trénování SOM je cílem rozmístit neurony tak, aby pokrývaly co nejvíce trénovacích dat a aby jednotlivé vstupní vektory byly co nejblíže k neuronům SOM. Pokud se neurony podaří správně rozmístit, budou pravděpodobně ve shlucích (centroidech). Každý neuron v jednom shluku zastupuje určitou kategorii ve vstupních datech.

2.3.2 Klasifikátor obrázku – konvoluční neuronové sítě

2.3.2.1 Předtrénovaný model

Pro vytvoření modelu, který bude klasifikovat obrovské množství obrázků, je potřeba hluboká neuronová síť obsahující mnohdy i stovky skrytých vrstev. Takto hluboké sítě je však obtížné natrénovat, a čím více je síť hluboká tím více se může v určitých případech zhoršovat její přesnost. Reziduální konvoluční sítě pracují na principu doučení se zbytkových funkcí místo učení se funkcí nových. Toho je dosáhnuto tím, že při učení jsou některé vrstvy přeskakovány (vstup do vrstvy n je napojen do vrstvy $(n + x)$ a tím tvoří určitý reziduální blok.

Během učení je výstup reziduálního bloku doplněn o původní vstupní data. Pokud se vrstvy reziduálního bloku nenatrénují na tato data, nebo se naopak přeučí, není transformace tímto blokem využita a tento blok je přeskočen.



Obrázek 2.3: Reziduální blok

Pro obecnou klasifikaci obrázků byl zvolen model ResNet. Tato moderní konvoluční neuronová síť využívá architekturu reziduálních bloků a jako váhy pro tento model byl zvolen připravený soubor s váhami z databáze ImageNet. Tato databáze obsahuje více než 20 000 kategorií obrázků a více než 1 000 000 obrázků má přiřazenou určitou kategorii.

2.3.2.2 Dotrénování předučeného modelu

Pro modely klasifikující specifickou, uživatelem zadanou množinu obrázků platí obdobná pravidla jako pro předtrénované modely. Architektura konvoluční sítě je ve většině případů stejná. Pokud by uživatel začal trénovat

model, který nemá vůbec nastavené váhy a žádná vrstva modelu tedy není natrénovaná, potřeboval by velkou množinu vstupních dat a samotný trénink sítě by trval velice dlouhou dobu[1]. Z toho důvodu se v těchto případech využívá techniky přenosového učení (transfer learning) společně s postupným dolaďováním vrstev modelu (fine tunning).

Přenosové učení je způsob doučení nové funkcionality modelu, který je již naučen na určitém problému. Základní model je naučen na velké množině obrázků. Poslední plně propojená vrstva, sloužící jako výstupní vrstva je z modelu odebrána a nový model je dotrénován na nové množině vstupních dat. Takto dojde k upravení vah modelu a původní model posléze slouží jako extraktor příznaků (feature extractor).

Dolaďování vrstev je metoda, která se provede po přenosovém učení. Vrchní klasifikační vrstva modelu již byla odebrána a nyní je nahrazena novým klasifikátorem. Nový klasifikátor může být pouze jedna vrstva, nebo několik propojených vrstev. Následně proběhne nové učení, kdy je model učen na vstupní datové sadě a váhy neuronů modelu jsou upravovány pomocí zpětné propagace chyb.

2.3.3 Klasifikace textu pro chatbota – dopředná neuronová síť

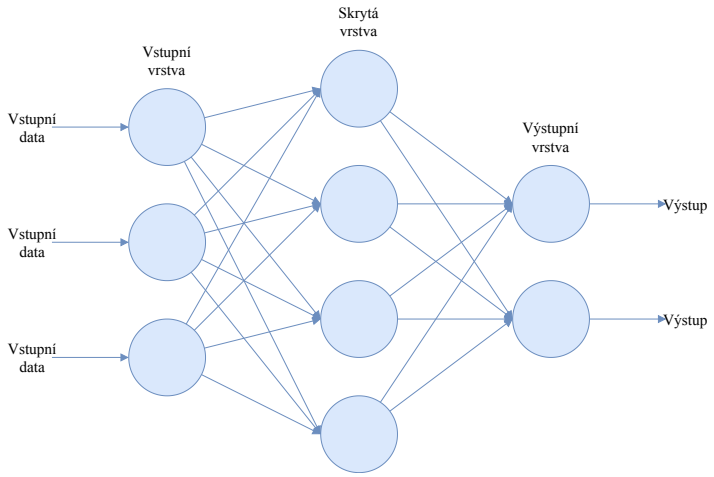
Při vytváření chatbotů pomocí neuronových sítí lze použít dva odlišné přístupy. Prvním přístupem je nechat síť vytvářet odpověď na základě uživatelova vstupu a naučených dat. Tento přístup je vhodné použít, pokud je úkolem vytvořit chatbota, který bude schopen vést konverzace o různých tématech.

Druhým přístupem, je nechat síť klasifikovat uživatelův záměr, tedy podstatu uživatelova textu a odpověď vybrat z množiny předpřipravených odpovědí pro daný záměr. Tento přístup je vhodný pro informační chatboty, kteří odpovídají na otázky uživatele. Například pokud se uživatel ptá na stav objednávky, nebo na otevírací hodiny obchodu, není potřeba, aby model vytvářel odpověď. Stačí pouze, aby správně určil, že se uživatel ptá právě na otevírací hodiny obchodu a následně mu vrátil připravenou odpověď.

Pro klasifikaci textového vstupu lze využít dopředné neuronové sítě. Architektura sítě je na obrázku 2.4.

Vstupní vrstva má jediný úkol, přijímá vstupní data v podobě číselného vektoru a distribuuje je na další skryté vrstvy modelu. Dimenze vstupní vrstvy je určena podle dimenze vstupního vektoru. Jednotlivé skryté vrstvy jsou plně propojeny jak mezi sebou tak i s výstupní vrstvou. Jednotlivé vrstvy využívají zpětnovazební učení, kdy pro každý vstup provedou nastavení vah. Následně je

[1]Například klasifikační model ImageNet ILSVRC byl trénován na 1.2 milionu obrázků a samotný trénink modelu bežící na několika GPU zabral mezi 2-3 týdny.



Obrázek 2.4: Model neuronové sítě chatbota

podle výstupní chyby proveden zpětný krok, kdy je spočítána chyba každému neuronu vnitřních vrstev a upravena váha. Jako aktivační funkce je použita funkce Softmax.

2.4 Výběr technologií pro implementaci klasifikátorů

Na trhu dnes existuje mnoho knihoven a projektů pro implementaci neuronových sítí a strojového učení obecně. V rámci akademické sféry jsou nejčastěji využívány jazyky R a Python. V rámci průmyslového prostředí je častěji volen jazyk Python a to hlavně díky jeho univerzálnosti. V tomto jazyce lze totiž pohodlně napsat jak model pro strojové učení, tak i následné podpůrné programy a infrastrukturu, kterou je potřeba vybudovat.

2.4.1 TensorFlow a Keras

Mezi nejznámější a nejpoužívanější nástroj pro vývoj neuronových sítí patří projekt TensorFlow[10]. Tento framework pro strojové učení je vyvíjen společností Google a patří mu třetí příčka mezi největšími projekty s otevřenou licencí na Github [2] a je nejoblíbenější framework v rámci komunity na stránkách stackoverlow [3].

TensorFlow společně s knihovnou Keras tvoří výkonný nástroj pro prototypování a vytváření modelů neuronových sítí. Keras nabízí **Model API**

[2]Data z dotazníkového výzkumu Octoverse 2018, kde je TensorFlow na třetí pozici s 9 300 přispěvateli

[3]Data z dotazníkového výzkumu Developer Survey 2018, kde je TensorFlow nejoblíbenějším framework s 73.5 %

a **Sequential API**, které umožňují implementaci architektury neuronové sítě po jednotlivých vrstvách, standardním způsobem[11]. Tento postup velmi urychluje tvorbu prototypů, ale i neuronových sítí schopných pracovat v produkčním prostředí. Keras slouží jako nadstavba nad TensorFlow, lze jej však použít i jako nadstavbu nad další framework pro strojové učení Theano. TensorFlow však nabízí více funkcionality, proto mohou nastat případy, kdy využití samotné knihovny Keras nestačí. Knihovna Keras je navíc v aktuální verzi začleněna přímo do distribuce nástroje TensorFlow.

```
from keras import Sequential
from keras.layers import Dense
import numpy as np

model = Sequential()
model.add(Dense(32, activation='relu', input_dim=100))
model.add(Dense(1, activation='sigmoid'))
model.compile(optimizer='rmsprop', loss='binary_crossentropy',
        metrics=['accuracy'])

data = np.random.random((1000, 100))
labels = np.random.randint(2, size=(1000, 1))
model.fit(data, labels, epochs=10, batch_size=32)
```

Ukázka zdrojového kódu 2.1: Zdrojový kód dopředné neuronové sítě vytvořený pomocí knihovny Keras

2.5 Platforma pro automatizované nasazování aplikací

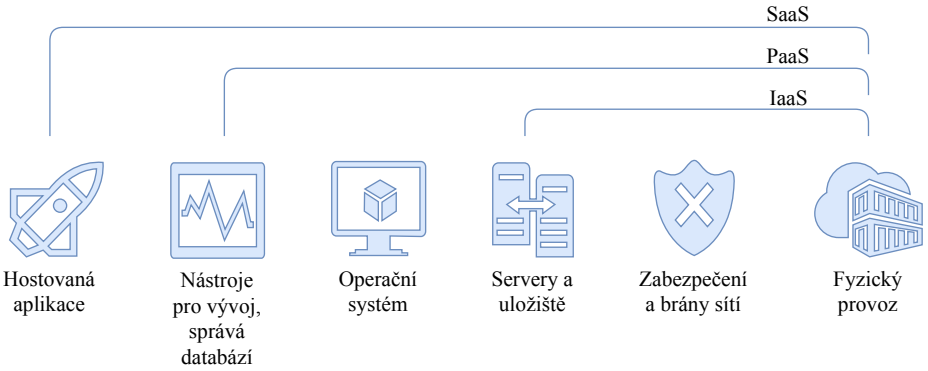
Automatizované nasazení a běh klasifikátorů a uživatelských modulů je stěžejní funkcionalita celého systému. Platforma pro jejich nasazení musí podporovat jak jednoduchou správu celého životního cyklu nasazených aplikací, tak i zabezpečení a případné škálovaní.

2.5.1 Distribuční modely

Cloudové technologie nabízejí uživateli možnost využívat infrastrukturu či aplikace bez nutnosti vlastnit vlastní hardware. Díky velkému množství poskytovatelů a konkurenčnímu prostředí má uživatel možnost vybrat si z několika možných modelů ten, který bude nejvíce vyhovovat jeho potřebám. Obecně lze nabídku distribučních cloudových modelů rozdělit na tři kategorie

- **IaaS** – Infrastruktura jako služba.
- **PaaS** – Platforma jako služba.
- **SaaS** – Software jako služba.

Každý z těchto modelů nabízí jiné vlastnosti a služby. Nejnižším modelem je model **Infrastruktura jako služba**, kde má uživatel nejvíce možností upravit prostředí, které mu je poskytnuto. Druhým modelem je **Platforma jako služba**, která nabízí uživateli připravené prostředí pro celý životní cyklus aplikace. Posledním modelem je **Software jako služba**, který nabízí určitou aplikaci uživateli, který jí následně využívá[12].



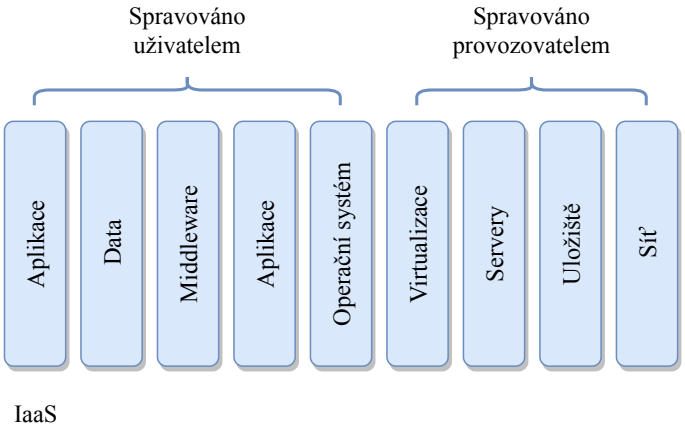
Obrázek 2.5: Hierarchie distribučních systémů

2.5.1.1 Infrastruktura jako služba

Distribuční cloudový model infrastruktura jako služba nabízí využití infrastruktury existující na vzdálených serverech jako službu pro uživatele, který na tuto infrastrukturu nasazuje vlastní aplikace. Provozovatel typicky poskytuje výpočetní prostředky, úložiště či síťovou infrastrukturu uživateli, který za jejich využití platí poplatky. Uživatel si může nakonfigurovat jaké prostředky chce využívat, kolik operační paměti potřebuje, či jak velké má být diskové úložiště. Provozovatel mu následně pomocí automatických nástrojů takovéto prostředí vytvoří a předá uživateli přístupové údaje. Uživatel si posléze sám instaluje potřebné prostředí pro běh jeho aplikací a má tedy velmi otevřené možnosti jaký programovací jazyk použít, či jaké aplikace nainstalovat. Většina poskytovatelů nabízí možnost škálovat infrastrukturu v případě potřeby a tím poskytují uživatelům možnost rychle reagovat na případné změny trhu bez nutnosti nakupovat hardware, který v případě menší poptávky po uživatelově službě není využit. Velkou výhodou modelu infrastruktura jako služba je, že se uživatel nemusí starat o monitoring serverů, dodržování zabezpečení na úrovni hardware ani o správu fyzických serverů.

Základním prvkem modelu infrastruktura jako služba je typicky virtuální server nebo klastr serverů, které jsou přiděleny uživateli. Uživatel si může často předem navolit, jaké aplikace mají být nainstalovány a nakonfigurovány. Příkladem může být připravená infrastruktura s nainstalovaným operačním

systémem. Mezi poskytovatele tohoto modelu lze započítat například Amazon Elastic Compute nebo Microsoft Azure.



Obrázek 2.6: Distribuční model Infrastruktura jako služba

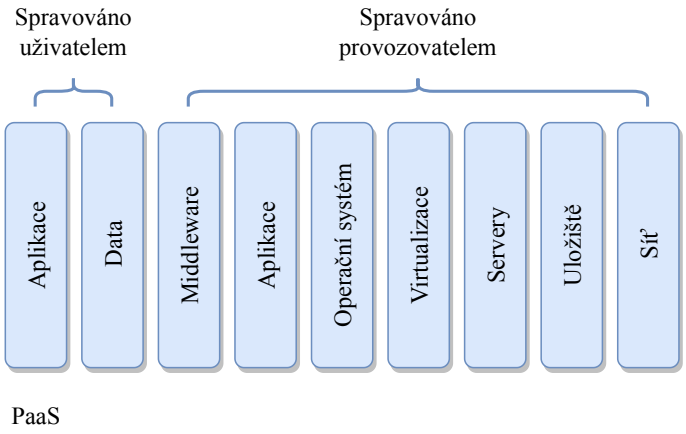
2.5.1.2 Platforma jako služba

Tento distribuční model nabízí uživateli připravené prostředí pro celý životní cyklus jeho aplikace. Uživatel tedy definuje, jaký typ prostředí pro svoji aplikaci potřebuje a poskytovatel mu pomocí automatických nástrojů toto prostředí vytvoří a předá přístupové údaje. Uživatel posléze toto prostředí může využívat jak pro vývoj aplikace, tak i pro její testování či produkční nasazení. Díky tomuto prostředí se uživatel nemusí starat o režii okolo běhu aplikace jako jsou konfigurace aplikačních serverů, či například příprava prostředí pro monitorování log záznamů. Platforma jako služba se obecně považuje za nadstavbu modelu Infrastruktura jako služba, jelikož uživatel s platformou získá i infrastrukturu. K infrastruktuře však uživatel nemá přímý přístup.

Nevýhodou tohoto přístupu je proprietární uzamčení u daného poskytovatele s daným typem prostředí. Pokud se uživatel rozhodne změnit poskytovatele, musí v některých případech aplikaci upravovat. Mezi poskytovatele tohoto modelu lze započítat například Google App Engine nebo Amazon Web Service.

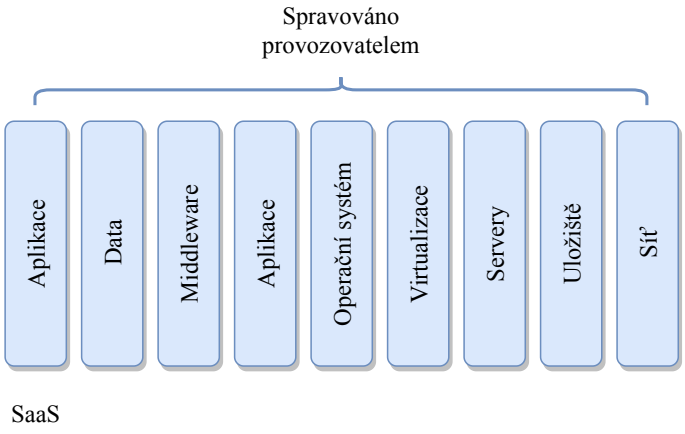
2.5.1.3 Software jako služba

Software jako služba nabízí uživateli využít aplikace třetí strany jako službu. Uživatel tedy nemusí aplikaci nikam instalovat ani ji konfigurovat. Poskytovatel nabízí uživateli aplikaci po dobu kdy platí předplatné. Po ukončení placení uživateli ve většině případů zaniká právo na využívání aplikace nebo se nabízená funkcionalita výrazně omezí.



Obrázek 2.7: Distribuční model Platforma jako služba

Uživatel se díky tomuto modelu nemusí zabývat instalací aplikace ani s její případnou aktualizací. Další výhodou je často možnost využití aplikace z vícero prostředí, nezávisle na uživatelově poloze. Častou nevýhodou je pak nutnost připojení k internetu v okamžiku, kdy je aplikace používána. Mezi poskytovatele tohoto modelu lze započítat například služby Microsoft Office 365, Google Docs nebo také Google Vision API, kdy Google nabízí své předtrénované modely a uživatel posílá dotazy a dostane v odpovědi například jaký objekt se nachází na obrázku.



Obrázek 2.8: Distribuční model Software jako služba

2.5.2 Kubernetes

Kubernetes je nástroj pro orchestraci kontejnerů, který je vyvíjen a distribuován pod svobodnou licencí. Tento nástroj slouží pro automatické nasazení, škálování a monitorování aplikací nasazených v kontejnerech uvnitř klastru běžícím na hostitelském stroji. Výhodou tohoto nástroje je velice dobrá dokumentace, obsahující mnoho příkladů, a výborná komunita. Kubernetes jakožto platforma nabízí přístup orientovaný primárně na kontejnery a jejich orchestraci a díky tomu jej lze považovat za hybridní model mezi modely Infrastruktura jako služba a Platforma jako služba.

Kubernetes využívá návrhový vzor **desired state**[13]. Tento návrhový vzor popisuje, jaký má být cílový stav operace, ale nepopisuje, jakým způsobem jej má být dosáhnuto. Jednoduchým příkladem může být to, že požadovaným stavem při jízdě automobilem je jet rychlostí 60 kilometrů za hodinu. Jakým způsobem se na tuto rychlost dostat, zdali akcelerací, brzděním či jinou metodou již není popsáno. Na stejném principu fungují konfigurační soubory pro Kubernetes. Konfigurace cílového stavu je zapsána do YAML souboru, ze kterého si Kubernetes klastr zaznamená, v jakém stavu se má nacházet, a neustále kontroluje, zdali se v tomto stavu nachází a pokud ne, například dojde-li k výpadku nasazeného kontejneru, okamžitě sjedná nápravu. V konfiguračním souboru, který obsahuje cílový stav, se nachází popis, jaké aplikace mají běžet, jaké obrazy k tomu použít, či v kolika instancích se mají vyskytovat. Tato konfigurace obsahuje i další informace, například jsou zde uvedena systémová omezení pro daný kontejner. Tedy například kolik paměti může využít, či na kolik procent vytěžovat procesor. Všechny konfigurace se zapisují do **ETCD databáze**, která popisuje cílový stav celého klastru. Z této databáze poté jednotlivé kontrolery čtou informace a případně provádějí operace tak, aby byl celý klastr v požadovaném stavu.

2.5.2.1 Kubernetes Java API

Kubernetes poskytuje několik implementací aplikačního rozhraní pro obsluhu a vytváření objektů. Mezi nejoblíbenější implementace se řadí Python a Java API[4], které nabízejí téměř veškerou funkcionalitu standardního aplikačního rozhraní. Zdrojový kód rozhraní pro jazyk Java je generován pomocí projektu swagger-codegen, kdy jsou jednotlivé funkcionality Kubernetes popsány pomocí OpenAPI specifikace. OpenAPI poskytuje sadu nástrojů s jejichž pomocí lze zdokumentovat libovolné API a následně, na základě této dokumentace, automatizovaně vytvořit knihovnu pro ovládání tohoto API v daném programovacím jazyce. OpenAPI dokumentace je psána ve formátu YAML nebo JSON.

[4] https://github.com/kubernetes-client

2.5.2.2 Hlavní uzel

Kubernetes klastr lze rozdělit na dvě logicky oddělené komponenty. První komponentou je hlavní uzel (Master Node), který má na starosti řízení celého klastru, rozhoduje kde poběží nové kontejnery, reaguje na události a provádí další akce. Tato komponenta obsahuje nástroj kube-apiserver, což je rozhraní celého klastru, se kterým komunikuje uživatel skrze REST API, přes konzolové aplikace (např. kubectl), či přes aplikační rozhraní. Toto rozhraní přijme požadavek, ověří jeho správnost a zavolá kontrolery odpovědné za dosažení požadovaného stavu. Skrze rozhraní také komunikuje hlavní uzel s jednotlivými pracovními uzly.

Nejdůležitější komponentou hlavního uzlu a celého klastru je **ETCD databáze**, která udržuje všechny konfigurace a popisuje cílový stav klastru. Veškeré konfigurace a nastavení jsou uloženy v této databázi. Data jsou zde uložena ve formátu **klíč-hodnota**. Tato databáze má své HTTP rozhraní, přes které s ní mohou jednotlivé uzly komunikovat.

Další důležitou částí hlavního uzlu je plánovač. Ten se stará o výběr pracovního uzlu, kde má být nový kontejner nasazen a udržuje informace o volných a obsazených zdrojích jednotlivých pracovní uzlů. Plánovač se v pravidelných intervalech dotazuje ETCD databáze, zdali obsahuje nějaké konfigurace Podů, které nemají přiřazený pracovní uzel. Pokud ano, přiřadí jim pracovní uzel a tuto informaci zapíše opět do této sdílené databáze.

Hlavní uzel se dále stará o kontrolery pro jednotlivé objekty Kubernetes klastru. Jedná se například o kontrolery replik, kontrolery pracovních uzlů a dalších. Každý z těchto kontrolerů se stará o určité prvky uvnitř pracovních uzlů a kontroluje zdali je dodržen popsaný stav. Pokud ne, provede nápravu. Například kontroler replik hlídá, zda replikační faktor daného Podu odpovídá požadovanému. Pokud některý z Podů havaroval, kontroler vytvoří nové. Naopak pokud je Podů více, kontroler některé vybere a ukončí je.

2.5.2.3 Pracovní uzly

Druhou komponentou klastru jsou pracovní uzly (Worker Node). V těchto uzlech běží jednotlivé Pody a v nich běží kontejnery. Každý pracovní uzel obsahuje běhové prostředí pro kontejnery, například Docker či runc. Hlavní uzel komunikuje s pracovními uzly pomocí komponenty **kubelet**, kterou má každý pracovní uzel samostatně. Tato komponenta řídí celý pracovní uzel, přijímá požadavky od hlavního uzlu a provádí je. Dále pracovní uzel nástroj obsahuje **kube-proxy**, který se stará o síťovou komunikaci a vyvažování zátěže.

2.5.2.4 Štítky a Selektor

Každému objektu v Kubernetes klastru lze přiřadit jeden či více štítků (label), které slouží k identifikaci daného objektu a také k propojení jednotlivých objektů. Jednotlivé objekty mají přiřazený **selektor**, který slouží k výběru

těchto objektů. Pokud je například potřeba spojit několik Podů do jedné služby, která bude navenek vystupovat jako jeden objekt, je těmto Podům přiřazen určitý stejný štítek a dále je vytvořena Kubernetes Service, která bude používat jako selektor tento štítek. Tento objekt třídy Service tedy bude seskupovat všechny Pody, které mají přiřazený určitý štítek. Jako selektor lze zvolit i více štítků. Štítek je ve tvaru klíč hodnota. Stejně tak bude probíhat i případné mazání objektů z klastru. Pokud bude například potřeba odebrat veškeré Pody na kterých běží určitá aplikace a těmto Podům byl při vytvoření přiřazen určitý štítek a také byla vytvořena Service s daným selektorem, stačí pouze odebrat tuto Service a nastavit odebrání všech potomků. Štítkování takto podporuje vzor desired state, kdy nezáleží na pořadí vytváření objektů a podporuje nižší provázanost objektů. Díky tomu pokud dojde například k odmazání všech Podů s daným štítkem a následně k jejich znovuvytvoření, jsou tyto Pody automaticky dále spravovány stejným objektem typu Service díky selektoru štítků[14].

2.5.2.5 Pod

Kubernetes Pod je základní jednotkou nasazení v Kubernetes klastru. Jedná se o nejmenší jednotku objektového modelu Kubernetes, ve které běží jeden nebo více kontejnerů, které spolu sdílejí zdroje, a které jsou silně propojené. Pokud v Podu běží několik kontejnerů, mohou spolu tyto kontejnery komunikovat na lokální úrovni, jako by běžely na stejném fyzickém stroji.

Při vytváření Podu v Kubernetes klastru, který běží na několika fyzických nebo virtuálních strojích, jsou kontejnery jednoho Podu seskupeny tak, aby běžely na stejném fyzickém či virtuálním stroji. Všechny kontejnery uvnitř Podu sdílejí jednu IP adresu, která je tomuto Podu přidělena. Sdílejí spolu také rozsah portů a připojená média. Pokud je do Podu připojené určité médium, například disk pro odkládání log záznamů, mohou mít všechny kontejnery uvnitř Podu připojené toto médium a zapisovat na něj či z něj číst.

Pod typicky představuje jednu instanci aplikace, která v něm běží. Pokud je potřeba aplikaci škálovat, vytváří se pomocí replik více Podů se stejnou aplikací běžící uvnitř kontejnerů. Při škálování se vždy vytváří nové Pody, se všemi kontejnery. Proto je potřeba, aby jednotlivé kontejnery v Podu byly skutečně silně provázané a nevznikala tak zbytečná režie kontejnerů, které nejsou tolik vytížené.

2.5.2.6 Sondy

Aby byla zajištěna obnova v případě selhání Podu, existují v Kubernetes sondy, které kontrolují, zda daný Pod běží v pořádku. Tyto sondy jsou dvou typů, **liveness** a **readiness**. Obě sondy fungují na principu HTTP/TCP dotazu. Na základě vráceného stavového kódu na tento dotaz je rozhodnuto, zdali má být daný Pod ukončen a případně nahrazen novým.

- Liveness sonda kontroluje, zdali Pod běží v pořádku. V případě, že tato sonda zjistí, že daný kontejner vrací chybový kód, dojde k restartu celého Podu. Kolikrát musí dojít k navrácení chybového kódu, či jak často má sonda kontrolovat daný Pod, lze nastavit v konfiguraci.

- Readiness sonda slouží ke zjištění, zdali sledovaná aplikace již korektně nastartovala a je připravena přijímat požadavky. Tato sonda je důležitá pro Liveness sondu. Pokud by okamžitě po vytvoření Podu začala Liveness sonda daný Pod kontrolovat, mohl by nastat případ, kdy Pod stále startuje, ale Liveness sonda již vysílá požadavek na jeho ukončení, jelikož nereaguje na její požadavky. Ve chvíli, kdy Readiness sonda dostane odpověď že je Pod již připravený, začne na něj připojená Service zasílat požadavky a Liveness sonda kontroluje jeho korektní běh.

Pokud hlavní proces uvnitř kontejneru skončí chybovým stavem, je tento kontejner automaticky restartován nezávisle na výsledcích vrácených sondami.

2.5.2.7 Připojená média

Pokud kontejnery uvnitř Podu potřebují přístup k souborům na lokálním disku, je vhodné, aby tyto soubory existovaly i v případě, že byl Pod ukončen a vytvořen nový. Dalším možným využitím je předání určitých vstupních dat, která jsou potřeba pro start aplikace uvnitř kontejneru, nebo například předání citlivých informací, či sdílení souborů mezi jednotlivými Pody.

K těmto účelům nabízí Kubernetes možnost připojení existujících médií (Volumes), která se při startu kontejneru uvnitř Podu zpřístupní. Do těchto médií mohou aplikace zapisovat a číst data a tím zajistit perzistentní úložiště pro odkládání dat nebo například pro jejich sdílení. Po ukončení existence Podů jsou tato média stále přístupná a nové aplikace k nim mohou přistupovat (záleží na typu připojeného média). Jednotlivé typy médií jsou abstrahovány pomocí konceptu perzistentních médií. Tento koncept nabízí aplikační rozhraní pro přístup k fyzickým médiím a jednotlivé implementace zajišťují správu dat. Pro jednotlivé typy lze při vytvoření definovat určitá omezení či přístupové módy. Kubernetes nabízí několik druhů médií, která lze připojit a každý z těchto druhů lze využít k jiným účelům.

- NFS neboli Network File Systém slouží pro připojení vzdálených médií pomocí počítačové sítě. Před samotným připojením je potřeba, aby existoval NFS server, na kterém budou média vytvářena. Po skončení existence Podu jsou média stále zachována, takže se k nim lze připojit i z jiného Podu, či připojit několik Podů zároveň.

- HostPath umožňuje připojení určitého souboru či složky z operačního systému, na kterém běží pracovní uzel. Připojený soubor či složka jsou

přístupné kontejnerům uvnitř Podu. Pokud aplikace potřebuje na připojené médium i zapisovat, je nutné, aby byl proces kontejneru puštěn s administrátorskými právy. Kubernetes ve výchozím nastavení spouští procesy uvnitř kontejnerů s těmito právy. To však z bezpečnostních důvodů není doporučený postup a je vhodné práva procesu upravit v konfiguračním souboru jednotlivých Podů.

- Local slouží k připojení lokálního disku či adresáře daného uzlu. Toto připojení není vhodné použít ve více uzlovém klastru, jelikož pokud bude Pod nasazen na jiný uzel, data nebudou přístupná. To, na jaký uzel bude Pod nasazen lze případně ovlivnit pomocí *nodeAffinity* v konfiguračním souboru. Pokud by však takovýchto Podů bylo mnoho, může nastat situace, kdy je většina pracovních uzlů nevytížena, zatímco jeden je zatížen téměř všemi běžícími Pody.
- Secret média slouží k připojení souborů obsahující citlivé údaje jako jsou například hesla či přístupové tokeny. Tyto údaje jsou uloženy v Kubernetes API a lze k nim přistupovat z kontejnerů těch Podů, které jsou k tomu nakonfigurovány.
- Další typy připojení fyzických médií závislé například na poskytovateli (např. Google Cloud Engine Persistent Disk).

2.5.2.8 Škálování uvnitř Kubernetes klastru

Při zvýšené zátěži a velkém množství požadavků na určitou službu, je potřeba škálovat aplikaci, aby byly všechny požadavky vyřízeny v co nejkratším čase. Při škálování lze využít dvou rozdílných přístupů.

- Prvním je horizontální škálování, kdy jsou přidávány další instance služby, na které jsou požadavky distribuovány. Tímto způsobem se zvýší dostupnost služby a dojde také k řešení výpadků některých instancí. Aplikace však musí být připravena na to, že bude distribuována v několika instancích. U bezstavových aplikací však tento problém odpadá.
- Druhým způsobem je vertikální škálování, kdy dojde ke zvýšení výpočetních prostředků stroje, na kterém daná služba běží. Takovéto škálování nevyžaduje úpravy aplikace, avšak jeho možnosti jsou omezené a poměrně nákladné.

Kubernetes využívá horizontální škálování, kdy jsou jednotlivé Pody replikovány a o jejich životnost se stará Replikační set a Deployment.

2.5.2.9 Replikační set a Deployment

Replikační set sleduje ETCD databázi a kontroluje, zda v klastru běží požadovaný počet Podů, který je uveden v konfiguraci určitého objektu Kubernetes

Deployment. Pokud je Podů méně, replikační set vytvoří nové, a naopak, pokud jich je více, některé z nich ukončí. To, které Pody budou ukončeny nelze předem stanovit, replikační set vybere náhodně jeden či více Podů a ty ukončí. Tímto způsobem probíhá škálování Podů v rámci klastru. Kubernetes Deployment zaštiťuje replikační set a rozšiřuje jeho chování a konfiguraci. Například umožňuje nastavit omezení, které říkají plánovači, na kterém pracovním uzlu mají být nové Pody vytvořeny.

2.5.2.10 Service

Kubernetes Service zajišťuje pojmenovanou abstrakci nad určitou množinou Podů a umožňuje k těmto Podům přistupovat jako k jednomu objektu. Objekt typu Service při svém vytvoření získá interní IP adresu a DNS jméno, na které jsou odesílány požadavky z klastru a Service poté distribuuje tyto požadavky na jednotlivé Pody. Pro výběr Podů, které mají být sloučeny pod tento objekt, je použit selektor štítků.

Typickým příkladem pro využití Kubernetes Service je propojení komunikace skupiny Podů obsahující webovou aplikaci se skupinou Podů obsahující REST API obsluhující tuto webovou aplikaci. Nejdříve dojde k vytvoření skupiny Podů s REST API, které jsou propojeny a zastřešeny pomocí selektoru štítků v konfiguraci Service. Tato Service má přiřazenou persistentní IP adresu a DNS jméno, pomocí kterého k ní lze přistupovat. Service následně rozesílá požadavky na jednotlivé Pody (např. s využitím *round robin* plánovacího algoritmu). Následně je vytvořena skupina Podů, které obsahují webovou aplikaci, a které jsou opět pomocí selektoru štítků spojeny v Service. Jednotlivé Pody s webovou aplikací jsou nakonfigurovány tak, aby hledaly DNS jméno Service zastřešující Pody s REST API, a odesílaly na ni požadavky. Pořadí vytváření jednotlivých Podů či objektů Service není nutné dodržovat právě díky selektoru štítků a s ním spojeného vyhledávání pomocí jména (service discovery).

Kubernetes Service může být několik typů.

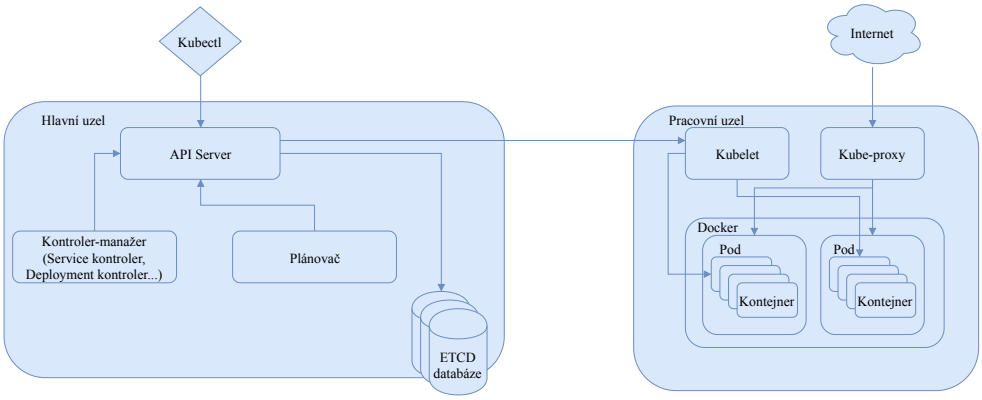
- External name – tento typ přiřazuje konkrétnímu objektu Service název (hostname), díky kterému k ní lze přistupovat (např example.com).
- ClusterIp – všechny objekty v rámci Kubernetes klastru mají dynamicky generovanou IP adresu. Pokud je Service typu ClusterIP, lze jí přidělit pevnou interní IP adresu v rámci klastru. Tento způsob lze využít například pro interní infrastrukturu v rámci klastru. Pokud je v klastru určitá Service, která slouží pro sběr log záznamů, lze jí přidělit pevnou IP adresu a sběrače log záznamů v rámci jednotlivých Podů nakonfigurovat tak, aby zasílaly zprávy právě na tuto adresu.
- NodePort – tento typ slouží k vystavení portu určité Service pro externí přístup ke Kubernetes klastru. Pokud je Service typu NodePort, dojde

k vystavení určitého portu a poté, pokud dojde k připojení na jakýkoliv stroj Kubernetes klastru, je tento port dostupný a skrze něj dojde k přístupu k dané Service. Z tohoto důvodu musejí být porty takto vystavené unikátní v rámci celého klastru.

- LoadBalancer – slouží jako rozšíření typu NodePort kdy přidává možnost distribuovat jednotlivé dotazy na určité Pody.

2.5.2.11 Jmenné prostory

Jmenné prostory (Namespaces) slouží k logickému oddělení Kubernetes objektů v rámci klastru. Pokud například na jednom klastru běží objekty jak pro testovací, tak pro produkční prostředí, lze tato prostředí oddělit do samostatných jmenných prostorů, kde jednotlivé objekty z testovacího jmenného prostoru nemají přístup k objektům z produkčního a naopak. Jednotlivým jmenným prostorům lze přiřazovat přístupová práva a tím například řídit kdo má přístup pouze k testovacímu prostředí a kdo i k prostředí produkčnímu.

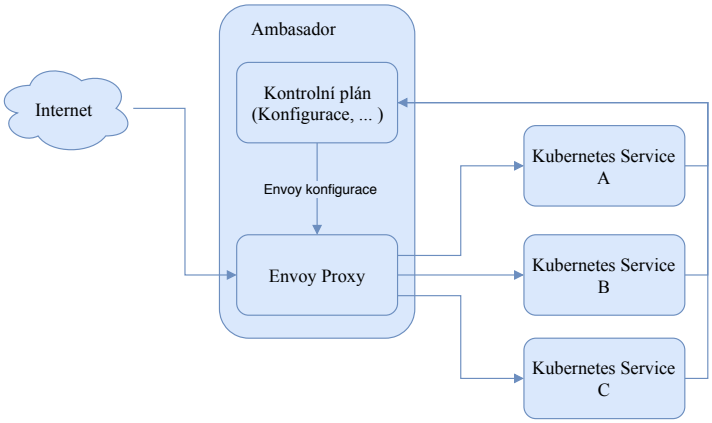


Obrázek 2.9: Struktura Kubernetes klastru

2.5.2.12 Kubernetes Ambasador

Kubernetes Ambasador je kontejner, který rozšiřuje funkcionalitu Kubernetes klastru. Ambasador pracuje jako návrhový vzor API Brána a slouží k vystavení kontrolerů Kubernetes Service koncovým uživatelům. Ambasador vystupuje vůči aplikacím a uživatelům přistupujícím z vnějšku klastru jako hlavní přístupový bod, který distribuuje požadavky na jednotlivé objekty typu Kubernetes Service na základě jejich přiřazené URI. Další funkcionalitou je možnost omezení směrování pro určité služby (například omezení počtu požadavků či maximální velikost požadavku)[15].

Ambasador je postaven na základech nástroje Envoy, což je proxy nástroj a komunikační sběrnice vytvořená pro účely rozsáhlých architektur složených z mnoha aplikací. Ambasador skenuje změny konfigurací jednotlivých objektů uvnitř Kubernetes klastru a pokud najde změny anotované Ambasador direktivou, propíše tyto změny do Envoy. Samotné směrování je tedy prováděno na straně Envoy, které je součástí nástroje Ambasador. Dále využívá API Kubernetes pro uložení konfigurace, díky tomu není potřeba další databázová úložiště a škálování lze provést pomocí replikačního setu[16].



Obrázek 2.10: Struktura komunikace pomocí nástroje Ambasador

Přidání nové Kubernetes Service, kterou bude Ambasador směrovat lze provést pomocí anotací uvnitř metadat. Při vytváření konfigurace lze přidat následující parametry pro vytvoření směrovacího pravidla[17].

```
metadata:
  name: python-api-stable
  annotations:
    getambassador.io/config: |
      ---
      apiVersion: ambassador/v0
      kind: Mapping
      name: python-api-canary-mapping
      prefix: /python-api
      weight: 10
      service: python-api-canary
```

Ukázka zdrojového kódu 2.2: Ukázka konfigurace nástroje Ambasador pomocí anotací

Ambasador podporuje několik typů konfigurace. Typ konfigurace se určuje hodnotou atributu *kind*. Možnosti konfigurací jsou následující:

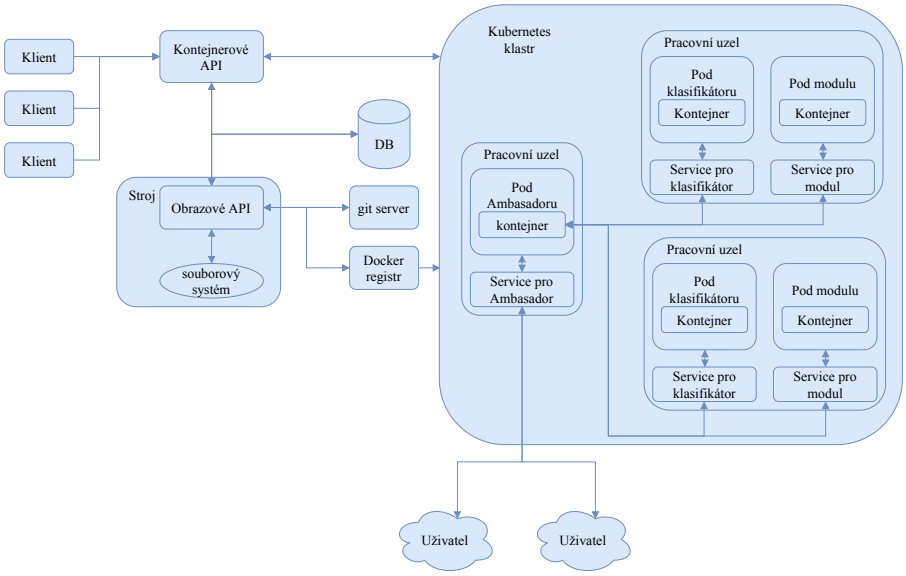
- Module manifests, který slouží ke konfiguraci nástroje Ambasador jako celku. Například lze takto definovat porty na kterých má Ambasador poslouchat.

- Sledovací služba, která konfiguruje Kubernetes Ambasador k využívání externí služby, která monitoruje jeho běh.
- Mapovací služba, která mapuje příchozí požadavky s Kubernetes Service.

2.6 Architektura systému

Před samotnou implementací systému je potřeba správně navrhnout jeho architekturu a definovat funkcionalitu jednotlivých komponent. Každá komponenta systému musí správně plnit svoji jedinečnou funkci a komunikovat s dalšími komponentami.

Pokud by byla vytvořena jedna monolitická aplikace, velmi by se snížila možnost jejího škálování, jelikož by nebyly vždy plně využity všechny její funkcionality. Naopak rozdělení systému na jednotlivé komponenty na základě jejich funkcionality přináší výhodu právě v jednoduchém škálování. Pokud bude například komponenta komunikující s Kubernetes klastrem plně vytížena, lze jednoduše přidat její další instanci a rozložit zátěž.



Obrázek 2.11: Architektura systému

Celý systém se skládá z následujících komponent:

- Klientská aplikace.
- Kontejnerové REST API s rozhraním pro klientské aplikace.
- Obrazové API.
- Databázový server.

- Kubernetes klastr.
- Instance klasifikátorů s modely neuronových sítí.
- Instance uživatelských modulů.

2.7 Komponenty systému

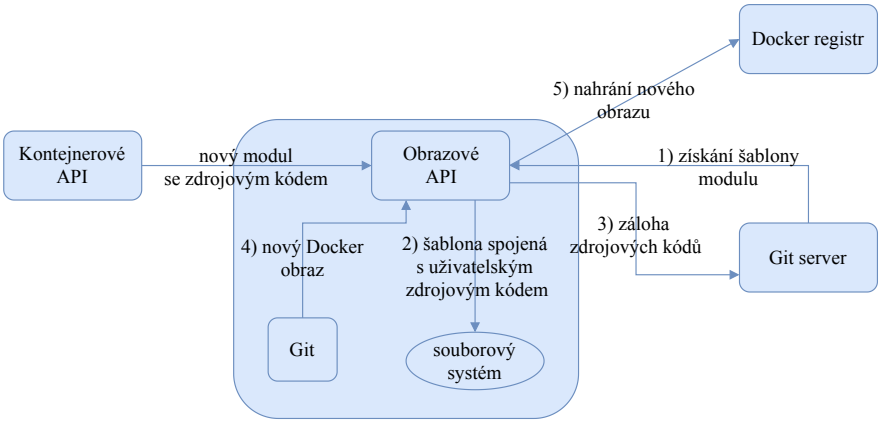
2.7.1 Kontejnerové REST API s rozhraním pro klientské aplikace

Kontejnerové REST API je hlavní komponentou celého systému. Tato aplikace slouží jednak jako rozhraní pro klientské aplikace, které na toto API posílají požadavky, ale také obsluhuje obrazové API a Kubernetes klastr. Ve chvíli, kdy uživatel skrze klientskou aplikaci vytvoří novou instanci klasifikátoru, aplikace pomocí Kubernetes Java API obsluhuje klastr a zadává mu požadavky pro vytvoření nového nasazení klasifikátoru s požadovanými atributy. Mezi tyto atributy patří například anotační informace pro Ambasador nebo informace o trénovacích datech pro samotný model sítě uvnitř klasifikátoru. Dále kontejnerové API slouží k vytváření nových modulů, kdy zasílá požadavky obrazovému API a následně obsluhuje Kubernetes klastr podobným způsobem jako při vytváření sítí.

2.7.2 Obrazové API

Obrazové API slouží k připravení Docker obrazu se zdrojovým kódem uživatelského modulu. Pokud uživatel vytvoří nový modul pomocí klientské aplikace, je zdrojový kód modulu pomocí kontejnerového API předán této aplikaci, která tento kód spojí s připravenou šablonou modulů. Uživatel tedy při vytváření modulu musí pouze dodržet rozhraní pro správné napojení jeho metod a šablony. Po spojení uživatelského kódu a šablony dojde k vytvoření Git repozitáře na serveru, kam je tento kód nahrán. Po úspěšném vytvoření repozitáře je ze šablony s uživatelským kódem vytvořen Docker obraz, který je následně odeslán do Docker registru. K tomuto registru má přístup Kubernetes klastr. Diagram komunikace vytvoření nového modulu je na obrázku 2.12.

Další možností je vytvoření modulu z Git repozitáře. V tomto případě je tento repozitář naklonován a následně je z něj vytvořen Docker obraz, který je opět odeslán do Docker registru. Proces vytvoření Docker obrazu je téměř totožný s procesem vytvoření obrazu při zadání zdrojového kódu v klientské aplikaci.



Obrázek 2.12: Proces vytvoření Docker obrazu ze zdrojového kódu zadaného v klientské aplikaci

2.7.3 Klientská aplikace

Klientská aplikace je jedním z nástrojů sloužících k interakci uživatele se systémem. V této komponentě uživatel vytváří nové instance klasifikátorů a předává jim potřebné informace pro naučení. Uživatel v aplikaci vybere jaký problém potřebuje řešit a pokud je to potřeba, předá trénovací data. Poté je mu vytvořena nová instance klasifikátoru, na kterou může odesílat požadavky. V klientské aplikaci může uživatel také vidět log záznamy jednotlivých sítí nebo testovat vytvořený klasifikátor.

Dále zde uživatel může vytvořit moduly, definovat jejich typ a propojit je s klasifikátory. Moduly lze vytvářet buďto v klientské aplikaci, kdy uživatel vloží svůj kód do předpřipravené šablony, dále má uživatel možnost vložit odkaz na Git repozitář. Při vytváření modulu, je možné specifikovat, skrze jakou klasifikační třídu je modul spojen s konkrétním klasifikátorem. Klientská aplikace komunikuje s kontejnerovým API pomocí REST API.

2.7.4 Databázový server

Komponenty systému potřebují pro správné fungování udržovat určité servisní informace. Mezi tyto informace například patří propojení mezi klasifikátory a uživatelskými moduly nebo cesty k trénovacím datům či informace, zdali je již neuronová síť uvnitř klasifikátoru naučena na trénovacích datech. Tyto informace jsou uloženy na databázovém serveru, ke kterému se připojují jak jednotlivé klasifikátory, tak i kontejnerové API s obrazovým API.

2.7.5 Kubernetes klastr

Kubernetes klastr slouží jako prostředí, ve kterém se automaticky nasazují kontejnery s klasifikátory a uživatelskými moduly. V okamžiku, kdy uživatel

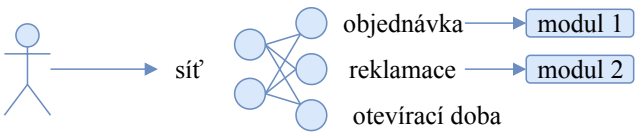
pomocí klientské aplikace zadá požadavek na nasazení modulu nebo klasifikátoru, kontejnerové API tento požadavek přijme a v rámci Kubernetes klastru vytvoří nové objekty Kubernetes Deployment a Kubernetes Service s anotacemi pro směrovací nástroj Ambasador. Jakmile je kontejner uvnitř Podu připraven, v klientské aplikaci se zobrazí zpráva a uživatel může odesílat požadavky.

2.7.6 Instance neuronových sítí a uživatelských modulů

Docker registr obsahuje připravené obrazy se zdrojovým kódem neuronových sítí a připraveným REST rozhraním pro komunikaci. Takováto aplikace se v rámci systému Chobot nazývá klasifikátor. Při nasazení jsou vytvořeny Kubernetes objekty typu Service a Deployment, které jsou skrze Ambasador přístupné z internetu. Uživatel poté může odeslat požadavek na síť, která jej oklasifikuje a pokud je k dané třídě klasifikace připojený, modul odešle uživatelův požadavek společně s třídou klasifikace na daný modul.

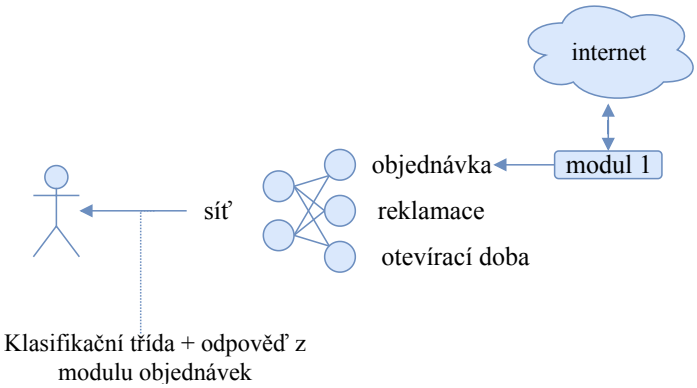
Moduly slouží k přidání další funkcionality k jednotlivým instancím klasifikátorů. Každý modul je napojený skrze třídu klasifikace na jeden určený klasifikátor. Jeden modul patří jednomu klasifikátoru, ale jeden klasifikátor může mít n napojených modulů. Pokud klasifikátor odešle požadavek na uživatelský modul, dojde k vykonání operace, která přijme jako vstupní parametry třídu klasifikace a požadavek, který uživatel poslal ke klasifikaci. Uvnitř metody se nachází uživatelem zadaný zdrojový kód.

Příkladem takové komunikace může být model neuronové sítě připravený jako chatbot pro potřeby určitého e-shopu. Model má několik tříd klasifikace a na určité třídy jsou namapované moduly. Pokud model přijme požadavek od uživatele, který vyhodnotí jako dotaz na objednávku, předá tento požadavek připojenému modulu.



Obrázek 2.13: Komunikace uživatele se sítí a modulem

Ve chvíli kdy modul přijme požadavek, vykoná uživatelem specifikované úkony. Například může odeslat požadavek na externí systém, aby byl získán stav objednávky. Poté vrátí síti informace, která je společně s třídou klasifikace předá odesílateli.



Obrázek 2.14: Odpověď uživatelského modulu

2.8 Zabezpečení systému

Zabezpečení systému a poskytnutí přístupu ke klasifikátorům a uživatelským modulům pouze autorizovaným uživatelům je důležitou součástí návrhu a implementace celého systému. Jelikož se celý systém skládá z několika komponent, musí být tyto komponenty zabezpečeny zvlášť. Tato kapitola popisuje jakým způsobem jsou jednotlivé komponenty zabezpečeny a jak probíhá komunikace mezi nimi.

2.8.1 Zabezpečení Kubernetes klastru

Kubernetes klastr je klíčovým prvkem celého systému Chobot. Všechny instance klasifikátorů a uživatelských modulů jsou vytvářeny právě uvnitř tohoto klastru. Je tedy potřeba zabezpečit přístup k této komponentě společně i s jejími přenosovými kanály. Kubernetes umožňuje zabezpečit přenos dat pomocí TLS certifikátu a komunikovat skrze HTTPS kanál[18]. Dále podporuje autorizaci klientských aplikací pomocí X509 certifikátů, které jsou vytvořeny při startu klastru nebo pomocí statických Bearer tokenů. Další vrstvou zabezpečení klastru jsou RBAC[5]. Tato funkcionalita umožňuje omezit uživatelská práva na základě jeho rolí. Uživateli lze omezit přístup ke všem objektům uvnitř určitého jmenného prostoru, zakázat mu veškerou editaci, nebo vytváření nových objektů.

2.8.1.1 Omezení prostředků nasazeným Podům

Zabezpečení přístupu ke klastru z externího prostředí je velice důležitou součástí návrhu a vývoje celého systému. Neméně důležité je zajistit omezení

[5]Role-Based Access Control

pro nasazované aplikace tak, aby jejich chod nemohl žádným způsobem ovlivnit chod celého klastru[6]. Základním předpokladem je, že všechny klasifikátory a uživatelské moduly jsou vytvářeny a nasazovány bez administrátorských oprávnění. Kubernetes v základním nastavení přiděluje všem vytvořeným kontejnerům administrátorská práva. Toto nastavení je v systému Chobot vypnuto a kontejnery tak nemohou využívat všechny prostředky Kubernetes a ani jakýmkoliv způsobem ovlivňovat jeho chod a konfiguraci. Všem vytvářeným klasifikátorům, ale i všem vytvářeným uživatelským modulům je omezen výkon procesoru, úložiště a paměti, kterou mohou spotřebovat. Dále je uživatelským modulům, které obsahují neznámý zdrojový kód, omezeno volání ostatních nasazených aplikací v rámci Kubernetes klastru. Tohoto nastavení je zajištěno pomocí konfigurace *HostNetwork* během vytváření jejich Podů.

Klasifikátorům je naopak nastaveno omezení, které určuje, že přijímají pouze příchozí požadavky z Ambasador směrovače, ostatní požadavky jsou automaticky odmítnuty. Toto nastavení lze provést globálně tak, že je aplikováno na všechny nasazené aplikace s určitým selektorem.

```
kind: NetworkPolicy
metadata:
  name: network-policy
  namespace: default
spec:
  policyTypes:
  - Ingress
  ingress:
  - from:
    - ipBlock:
      cidr: 172.17.0.0/16
      except:
      - 172.17.1.0/24
    - namespaceSelector:
      matchLabels:
        project: some_project
    - podSelector:
        matchLabels:
        role: some_label
        ports:
          - protocol: TCP
            port: 5000
```

Ukázka zdrojového kódu 2.3: Příklad nastavení konfigurace povolující aplikacím s určitým selektorem přijímat volání pouze z určité IP adresy

Dále jsou všem nasazeným aplikacím omezeny používané porty a přístup k souborovému systému klastru.

[6]Restriction policy on Pods

2.8.1.2 Zabezpečení brány Ambasador

Kubernetes Ambasador slouží jako přístupová API brána pro jednotlivé klasifikátory a uživatelské moduly. Podle URL požadavku směruje volání uživatele na konkrétní Pod v rámci klastru. Dále ambasador umožňuje zabezpečit připojení pomocí TLS a využívat tak protokol HTTPS. Pro správné nastavení TLS je potřeba uložit certifikát do Kubernetes klastru. Pro uložení citlivých údajů slouží v rámci Kubernetes objekty zvané Secret. Tyto objekty v sobě uchovávají citlivé informace a uživatel může při jejich vytváření nastavit jaká jsou oprávnění potřebná k přístupu k těmto datům. Po úspěšném uložení certifikátu jej lze zpřístupnit pomocí Kubernetes API a nastavit HTTPS port při konfiguraci Ambasadoru pomocí YAML konfiguračních souborů.

```
apiVersion: ambassador/v1
kind: Module
name: tls
config:
  server:
    secret: ambassador-certs
    spec:
  ports:
    - name: http
      protocol: TCP
      port: 80
    - name: https
      protocol: TCP
      port: 443
```

Ukázka zdrojového kódu 2.4: Konfigurace SSL pro Kubernetes Ambasador

2.8.2 Zabezpečení komunikace mezi klasifikátory a moduly

Komunikace mezi klasifikátory a moduly probíhá uvnitř Kubernetes klastru a využívá interní síť Kubernetes. Ta by měla odstínit všechny nežádoucí útoky z vnějšího připojení. Pro aditivní zabezpečení lze do klastru zavést další interní směrovač, který by sloužil pro kontrolu a zajištění HTTPS komunikace mezi klasifikátory a moduly.

Uživatelské moduly nemohou zasílat požadavky v rámci klastru a nemohou tedy ani zaslat požadavek na žádný klasifikátor. Naopak klasifikátory posílají požadavky na připojené uživatelské moduly. Uživatelský modul přijímá požadavky pouze společně s JWT tokenem, který musí obsahovat podpis klíčem daného modulu. Instance klasifikátoru tak při odeslání požadavku musí tento token připojit.

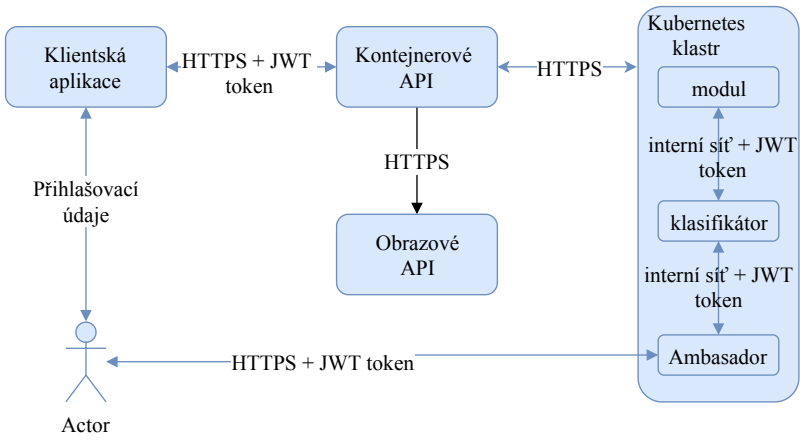
Pokud uživatel odešle požadavek bez JWT tokenu na klasifikátor, je tento požadavek klasifikátorem odmítnut. Uživatel tak musí využívat jak HTTPS připojení, tak společně s požadavky odesílat i správný token klasifikátoru.

2.8.3 Zabezpečení kontejnerového API

Kontejnerové API komunikuje s klientskou aplikací a obsluhuje požadavky uživatele. Po úspěšném přihlášení skrze klientskou aplikaci vygeneruje kontejnerové API JWT token, který předá klientské aplikaci. Tím je uživatel autentifikován a kontejnerové API dokáže pro každý uživatelův požadavek správně určit, zdali má uživatel oprávnění k manipulaci s danými prostředky. Komunikace mezi klientskou aplikací a kontejnerovým API probíhá pomocí protokolu HTTPS. Kontejnerové API i obrazové API využívají k zabezpečení aplikace knihovnu Spring security. Ta umožňuje nastavení TLS komunikace pomocí parametrů v konfiguračním souboru aplikace.

```
server.port=8443
server.ssl.enabled=true
server.ssl.key-store=classpath:server.jks
server.ssl.key-store-type=PKCS12
server.ssl.key-store-password=secret
server.ssl.key-alias=server
server.ssl.key-password=secret
```

Ukázka zdrojového kódu 2.5: Konfigurace SSL pro Spring Boot aplikace



Obrázek 2.15: Zabezpečení komunikace mezi komponentami systému

2.9 Profil uživatele systému – případy použití

2.9.1 Administrátoři informační systémů – detekce abnormálních log záznamů

Bezpečnost aplikací je velmi důležitou součástí návrhu i realizace každého projektu. Důležitou součástí tvorby aplikace je zabezpečení jednotlivých komponent proti útoku. I přes všechny snahy, nelze v reálném prostředí zabezpečit

aplikaci před všemi možnými útoky. Proto je potřeba sledovat a monitorovat události v aplikacích stále rostoucí.

Typickým uživatelem systémů, které slouží ke klasifikaci log záznamů, jsou systémoví administrátoři. V dnešní době, kdy je kladen důraz na robustní generování log záznamů u všech aplikací, je množství vyprodukovaných log záznamů enormní a díky tomu se stává manuální kontrola stále složitějším procesem. Na trhu je dnes velké množství SIEM systémů[7], které umožňují zpracovávat log záznamy z několika aplikací najednou a tím umožní administrátorovi sledovat události v určité aplikaci s informacemi o kontextu z dalších aplikací.

Ne ve všech případech však zná administrátor chování sledované aplikace tak dokonale, aby dokázal vytvořit pravidla, která odhalí abnormality v jejích log záznamech. Proto může nastat situace, kterou manuální prohledávání log záznamů neodhalí, a tím se výrazně prodlouží reakční doba na tuto událost. Automatická klasifikace log záznamů a detekce abnormalit umožní administrátorovi reagovat na události v aplikacích v téměř reálném čase. V okamžiku, kdy síť nasazená v systému Chobot označí určitou událost v log záznamech jako abnormální, může být administrátorovi odeslán informační e-mail nebo může být provedena jiná akce, kterou administrátor specifikuje v rámci uživatelského modulu.

2.9.2 E-shopy nebo donáškové služby – chatbot pro komunikaci se zákazníkem

Chatboti jsou v dnešní době stále žádanější službou. Umožňují komunikaci s koncovým zákazníkem a nabízí základní obsluhu zákazníka bez nutnosti reálné komunikace s operátorem. Některé aplikace umožňují provést objednávku určitého produktu a následné sledování objednávky právě pomocí konverzačních nástrojů. Pro příklad lze uvést půjčovnu automobilů, kdy uživatel provede skrze chatbota objednávku, specifikuje čas výpůjčky, nebo například pouze zjistí, kdy má daná prodejna otevřeno.

Díky chatbotům lze snížit náklady na operátory, kteří obsluhují zákaznické požadavky. Pokud si například zákazník přeje zjistit stav objednávky, odešle dotaz s číslem objednávky chatbotovi. Ten podle textu zprávy rozpozná, že se jedná o dotaz na objednávku a předá tento dotaz modulu, který odešle požadavek na objednávkový systém. Vrácená informace se zákazníkovi zobrazí na displej bez potřeby lidského zásahu.

2.9.3 E-shopy automatická kategorizace produktů na základě jejich fotografií

Mnoho e-shopů dnes rozřazuje produkty do kategorií. V těchto kategoriích je zákazníkům většinou umožněno dodatečné filtrování podle určitých atributů

[7]Security Information and Event Management system

nabízených produktů, Například pro e-shop s oblečením se může jednat o kategorie: *dámská trika*, filtrování: *barva, vzor, velikost, střih* a další. Aby takovéto filtrování pracovalo správně, je nutné každý produkt oklasifikovat pro každé filtrovací pravidlo. Pro výše zmíněná dámská trička musí pracovník e-shopu prohlédnout fotografii každého trička a určit jaká je jeho barva, jaký je vzor a střih. Tato manuální činnost lze automatizovat, kdy se fotografie každého trička odešle klasifikačnímu modelu a ten určí, jaký vzor se na triku nachází. Stejně tak lze provést i pro barvu či střih.

2.10 Procesy systému

Pokud bude uživatel chtít využívat klasifikátory poskytované systémem Chobot, bude muset projít procesem vytvoření klasifikátoru a případně i procesem přidání uživatelského modulu. Následující sekce popisuje tyto procesy.

2.10.1 Vytvoření a nasazení nového klasifikátoru

Pokud si uživatel přeje vytvořit novou instanci klasifikátoru, přejde v klientské aplikaci na hlavní stránku. Zde vidí seznam všech již vytvořených klasifikátorů, jejich aplikační status a další informace. Pro vytvoření nového klasifikátoru klikne uživatel na tlačítko **plus** a následně je přesměrován na formulář pro vytváření nového klasifikátoru. Zde vyplní jméno a vybere typ problému, který chce řešit.

Na základě typu problému se následně zobrazí vstupní pole, do kterého uživatel pomocí připravené komponenty nahraje trénovací data. Poté, pokud je potřeba, specifikuje další potřebné atributy pro učení sítě jako je například definice jak rozdělit vstupní data na prvky při analýze log záznamů. Následně klikne na potvrzovací tlačítko a klientská aplikace odešle POST požadavek na kontejnerové API. To po přijetí požadavku ověří správnost vstupních dat, pokud nejsou validní vrátí klientské aplikaci chybovou zprávu. Pokud jsou vstupní data validní, uloží kontejnerová aplikace data do databáze, vygeneruje další atributy jako je například URL pro připojení a vrátí novou instanci klasifikátoru doplněného o jeho identifikátor.

Klientská aplikace po přijetí odpovědi od kontejnerového API zobrazí chybovou hlášku, pokud byla data nevalidní, nebo provede přesměrování uživatele na stránku s detailem klasifikátoru. Zde může uživatel vidět informace o klasifikátoru a jeho aplikační status. Jelikož ještě nedošlo k provedení nasazení klasifikátoru uvnitř Kubernetes klastru, zobrazí se uživateli informace o tom, že klasifikátor zatím nebyl nasazen a tlačítko pro nasazení.

Ve chvíli, kdy uživatel klikne na toto tlačítko, odešle klientská aplikace POST požadavek na kontejnerové API. To po přijetí požadavku získá data o klasifikátoru z databáze a vytvoří novou Kubernetes Service, kde definuje URL pro připojení v podobě Ambasador anotací, nastaví štítky podle identifikátoru klasifikátoru v podobě *uživatelské_jméno-jméno_klasifikátoru* a předá

tuto Service Kubernetes Java API, které ji přidá do ETCD databáze. Po vytvoření nové Kubernetes Service vytvoří kontejnerové API také objekt Kubernetes Deployment, kde definuje jméno Docker obrazu klasifikátoru, jeho omezení, nastaví mu identifikátor a opět předá tento požadavek Kubernetes Java API, které jej zapíše do ETCD databáze.

Po přidání konfigurace objektu Kubernetes Deployment do ETCD databáze, zaznamená tuto změnu plánovač a volá příslušné kontrolery Kubernetes, které začnou vytvářet jednotlivé Pody a další potřebné objekty. V tu chvíli vrátí kontejnerové API klientské aplikaci informaci o provedeném nasazení. Klientská aplikace informuje uživatele o provedení nasazení, a čeká, dokud instance klasifikátoru neodpoví kladně na volání pro zjištění stavu. Již během nasazování a trénování klasifikátoru může uživatel v klientské aplikace prohlížet aplikační log záznamy nebo vytvářet uživatelské moduly.

2.10.2 Vytvoření nového uživatelského modulu

Pokud má uživatel vytvořený klasifikátor, může pro něj vytvářet uživatelské moduly a propojovat je skrze třídy klasifikace. Uživatel přejde na detail klasifikátoru a zde vybere záložku s moduly. Na této záložce se mu zobrazí již vytvořené moduly a tlačítko plus. Po jeho stisknutí klientská aplikace přesměruje uživatele na stránku, kde se nachází formulář pro vytvoření nového modulu.

Na této stránce uživatel vybere třídu klasifikace, jméno modulu a jeho typ. Pokud vybere uživatel typ Git repozitáře, zobrazí se mu vstupní pole pro zadání odkazu na repozitář. Pokud ale uživatel vybere typ *lambda* zobrazí se mu v rámci klientské aplikace nová záložka, kde zadává zdrojový kód nového modulu.

Po zadání zdrojového kódu klikne uživatel na tlačítko pro vytvoření modulu a klientská aplikace odešle POST požadavek na kontejnerové API. To tento požadavek ověří a v případě, že je vše v pořádku, uloží data do databáze. Pokud data validní nejsou, vrátí klientské aplikaci chybový kód. Po uložení dat do databáze odešle kontejnerové API požadavek na vytvoření obrazu obrazovému API, kde definuje identifikátor modulu.

Obrazové API přijme tento požadavek a začne s vytvářením nového obrazu pro uživatelský modul. Kroky obrazového API jsou následující.

1. Jako první krok obrazové API stáhne šablonu, kde jsou zdrojové kódy s definicí kontrolerů, zabezpečení modulu a s definicí Dockerfile a umístí je do uživatelské složky na serveru.

2. Poté do stejné složky vloží zdrojové kódy definované uživatelem.

3. Poté vytvoří nový Git repozitář, kam tyto zdrojové kódy společně s šablonou nahraje pro případné obnovení při ztrátě dat.

4. V dalším kroku vytvoří Docker obraz z modulu a odešle jej do privátního Docker repozitáře.

Po vytvoření modulu je uživateli pomocí klientské aplikace zobrazena informace a ten může pokračovat v procesu nasazení modulu. Tento proces je stejný jako při nasazení klasifikátoru.

2.10.3 Požadavek na klasifikátor

Klientská aplikace zobrazuje uživateli potřebné informace pro odeslání požadavku na klasifikátor společně s připraveným ukázkovým požadavkem v podobě curl příkazu. Pokud chce uživatel pouze otestovat klasifikátor, stačí mu zkopírovat tento požadavek a vložit jej do příkazové řádky. Dále musí upravit vstupní data a požadavek odeslat.

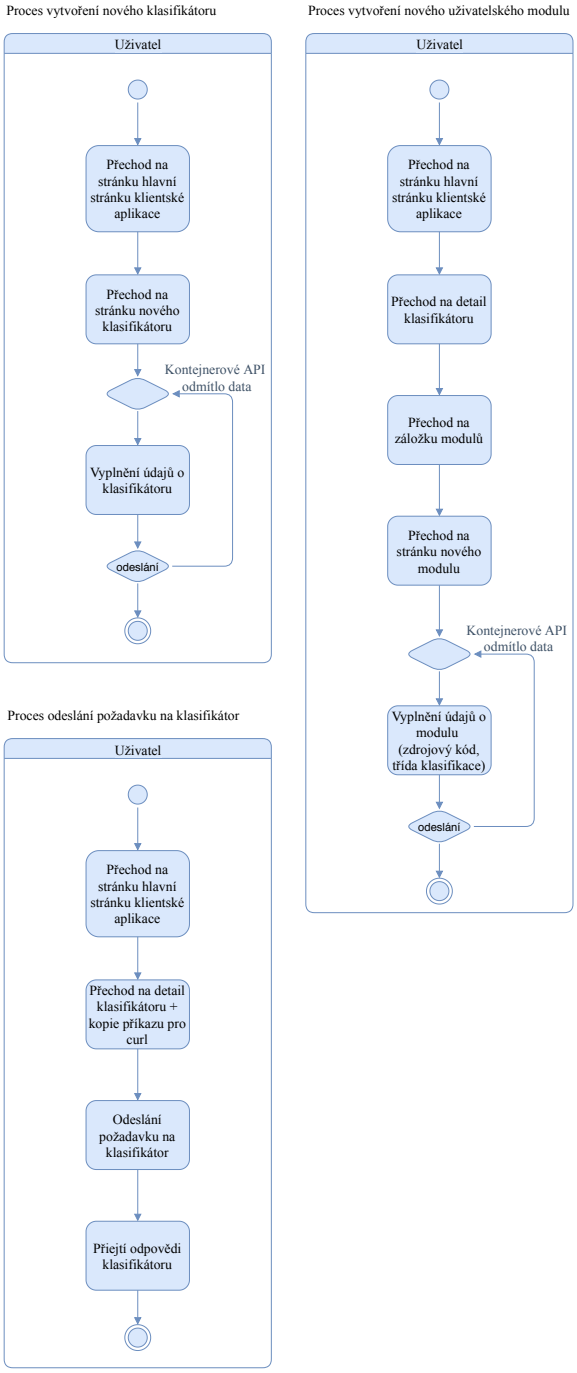
Po přijetí takovéhoto požadavku klasifikátor zkontroluje uživatelovu autorizaci a pokud je vše v pořádku, předá klasifikátor vstupní data modelu. Ten tato data ohodnotí a vrátí třídu klasifikace. Poté klasifikátor vybere připojený modul, který je napojen skrze stejnou třídu klasifikace jako je ta, kterou vrátil model a odešle na něj požadavek. Po přijetí odpovědi klasifikátor vrátí tato data uživateli.

Popsané procesy jsou na obrázku 2.16

2.11 Závěr

V této kapitole jsou popsány jednotlivé komponenty systému jejich funkcionalita a využití v rámci systému Chobot. Každá komponenta systému má svojí specifickou funkci, proto i požadavky popsány v této kapitole byly rozděleny právě podle jednotlivých komponent. Dále jsou v této kapitole popsány případy užití systému, použité technologie a modely zvolené pro klasifikaci.

Během procesu analýzy systému se autor práce zabýval jak správným a udržitelným způsobem propojit jednotlivé komponenty tak, aby jejich funkcionalita byla rozdělena do logických celků a nevznikl monolit spravující veškeré funkce systému. Na základě poznatků získaných během analýzy vznikl systém a jeho jednotlivé komponenty. Implementace toho systému je popsána v kapitole Realizace.



Obrázek 2.16: Procesy systému

Kapitola 3

Realizace

V této kapitole jsou popsány postupy, které byly použity při implementaci systému Chobot. Kapitola popisuje vývoj a důležité funkcionality jednotlivých komponent a jejich vzájemnou komunikaci.

3.1 Realizace kontejnerového API

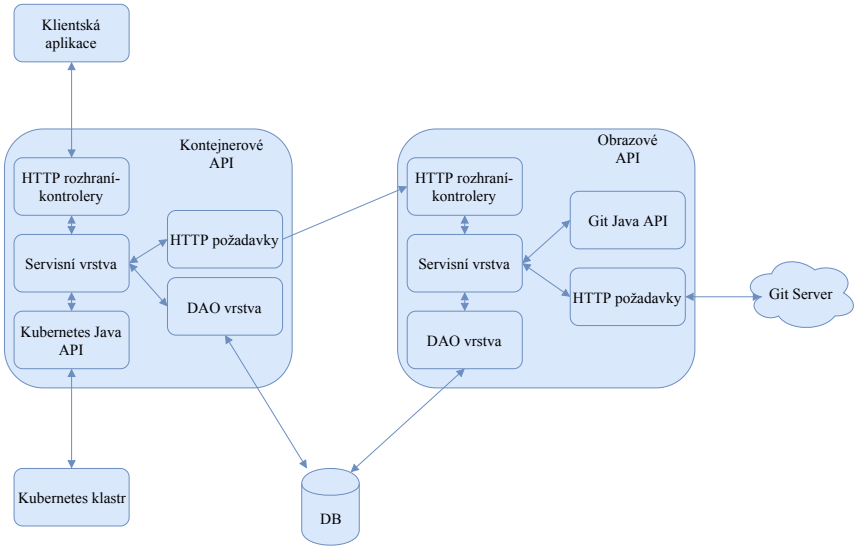
Kontejnerové API komunikuje s klientskou aplikací, pro kterou je hlavním zdrojem dat. Klientská aplikace odesílá požadavky a kontejnerové API vrací požadovaná data, případně provádí další operace. Další funkcionalitou kontejnerového API je komunikace s Kubernetes klastrem a obrazovým API, kterým zadává požadavky na základě požadavků klientské aplikace. Kontejnerová aplikace lze rozdělit na čtyři logické celky.

- Datová vrstva pro práci s databázemi.
- Servisní vrstva pro logické operace.
- Servisní vrstva pro komunikaci a obsluhu Kubernetes klastru.
- Kontrolery pro komunikaci s klientskou aplikací.

Kontejnerové API je napsáno v programovacím jazyce Kotlin a využívá funkcionalit knihovny Spring Boot[19]. Tato komponenta má vytvořené REST rozhraní které slouží pro komunikaci s klientskou aplikací, dále odesílá požadavky na REST rozhraní obrazového API a pomocí Java API komunikuje s Kubernetes klastrem. Schéma propojení a styl komunikace s kontejnerovým API je zobrazeno na obrázku 3.1.

3.1.1 Datová vrstva

Celý systém využívá pro ukládání servisních informací PostgreSQL databázi. V této databázi jsou uložené konfigurační informace o projení klasifikátorů



Obrázek 3.1: Komunikace s kontejnerovým API

s uživatelskými moduly, konfigurační údaje modelů pro klasifikátory nebo odkazy se jmény Docker obrazů pro moduly jednotlivých uživatelů. Dále jsou zde uloženy informace o uživatelích systému a další informace. Tuto databázi využívají k uchovávání informací jednotlivé instance klasifikátorů, kontejnerové API a obrazové API. Instance uživatelských modulů k této databázi nemají z bezpečnostních důvodů přístup.

3.1.1.1 Popis entit systému

- Tabulka **Network** – v této tabulce se nachází informace o jednotlivých instancích klasifikátorů. Systém obsahuje několik předem připravených obrazů klasifikátorů pro jednotlivé typy problémů. V okamžiku, kdy uživatel vytváří novou instanci klasifikátoru, definuje její typ. Tento typ určuje, jaký obraz klasifikátoru bude použit a jaké jsou jeho atributy potřebné pro učení. Typ klasifikátoru je určen pomocí propojení tabulek **Network** a **Network type**. Dále jsou během učení a běhu klasifikátoru potřeba různé servisní informace. Tyto informace jsou uloženy v tabulce **Network Parameter**. Propojení mezi klasifikátory a moduly je následně uloženo v tabulce **Module**.

  Každý klasifikátor má dále definované jméno, je propojený s určitým uživatelem a má připojené moduly. Informace o stavu klasifikátoru jsou uloženy ve sloupci *status* a informují o tom, zdali je klasifikátor již nasazený v Kubernetes klastru, nebo zdali se jedná o nově vytvořený klasifikátor. Dále tato tabulka udržuje informace o parametrech připojení ke klasifikátoru.

- Tabulka **Network type** – každý klasifikátor je určitého typu, který je určen uživatelem při definici problému v rámci klientské aplikace. Systém obsahuje předpřipravené obrazy klasifikátorů, ze kterých jsou následně vytvářeny instance, které jsou trénovány uživatelskými daty. Každá instance klasifikátoru má tedy uložený svůj typ v podobě odkazu na tuto tabulku, ve které se nachází jméno Docker obrazu, ze kterého je klasifikátor při nasazení vytvořen.

- Tabulka **Network Parameter** – během učení a běhu klasifikátoru jsou potřeba určité servisní informace. Tyto informace jsou uloženy v tabulce Network Parameter. Každý parametr náleží jedné určité instanci klasifikátoru a může obsahovat informace o uložení trénovacích dat, nebo například pravidlo pro rozklad vstupních řetězců při klasifikaci log záznamů. Klasifikátor následně během svého učení čte informace uložené v této tabulce a získá tak odkaz na požadovaná data, nebo zmíněné rozkladové pravidlo.

- Tabulka **Module** – tato tabulka obsahuje informace o uživatelských modulech. Každý modul je napojen na klasifikátor a má definovanou třídu klasifikace, skrze kterou klasifikátor určuje, zdali požadavek na daný modul zaslat. Tyto informace jsou uloženy ve sloupcích *responseClass* a *networkId*. Dále mají moduly, stejně jako klasifikátory, definovaný aktuální stav nasazení ve sloupci *status* a informace o parametrech pro připojení a odeslání požadavku.

  Uživatel může vytvořit klasifikátor dvěma způsoby. První možností je zadání zdrojového kódu do klientské aplikace, který je následně spojen s předpřipravenou šablonou. Druhou možností je předání odkazu na Git repozitář. Typ modulu je určen ve sloupci *type* a je důležitý pro obrazové API, které na základě typu modulu provádí potřebné operace pro vytvoření a nahrání Docker obrazu do repozitáře systému.

- Tabulka **User** – v této tabulce se nachází informace o jednotlivých uživatelích systému. Každý uživatel má jméno, příjmení, přihlašovací údaje, e-mail a heslo. Díky propojení uživatele a klasifikátorů, lze vybrat všechny uživatelovy klasifikátory a na ně připojené moduly.

3.1.1.2 DAO vrstva kontejnerového API

Pro spolupráci s databází využívá kontejnerové API framework Spring Data JPA, který pracuje s mapovacími POJO[8] třídami, které reprezentují jednotlivé databázové entity v rámci aplikace a zároveň jsou doplněny o anotace z knihovny javax.persistance. Meta informace poskytované těmito anotacemi umožňují mapování jednotlivých entit systému na tabulky databáze.

[8]Plain Old Java Objects – třída obsahující pouze atributy, ale žádnou aplikační logiku

Jednotlivé namapované entity jsou následně využívány v aplikačních DAO rozhraních. Ta dědí svoji funkcionalitu od **Spring Data CrudRepository** rozhraní. Spring Data **CrudRepository** nabízí funkcionalitu vytváření metod pro čtení, zápis, aktualizaci a mazání entit z databáze, pouze definováním hlavičky dané metody.

```
@Entity
@Table(name = "network_parameter")
data class NetworkParameter(
    @NotNull
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "network_parameter_id", unique = true,
        nullable = false)
    val id: Long? = null,

    @NotNull
    @Column(name = "name", nullable = false)
    var name: String,

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "network_id")
    var network: Network? = null
) {}
```

Ukázka zdrojového kódu 3.1: Ukázka definice entity v rámci kontejnerového API

```
import org.springframework.data.jpa.repository.JpaRepository

interface NetworkRepository : JpaRepository<Network,Long>{
    fun findByName(name: String): Set<Network>
}
```

Ukázka zdrojového kódu 3.2: Ukázka rozhraní JpaRepository

Toto rozhraní je následně injektováno do dalších vrstev kontejnerového API, kde je využíváno pro získání dat z databázového serveru a mapování těchto dat na entity systému.

3.1.2 Logická vrstva

Logická část kontejnerového API lze rozdělit na tři rozdílné sekce. První sekce zastřešuje CRUD operace pro komunikaci s klientskou aplikací. Při požadavku z klientské aplikace přijmou servisní třídy požadavek a pomocí DAO vrstvy získají data z databáze. Tato data následně předají kontrolerům. Druhou sekcí funkcionality je komunikace kontejnerové aplikace s Kubernetes klastrem pomocí jeho Java API. Kontejnerová aplikace zadává klastru požadavky na vytvoření nových nasazení modulů a klasifikátorů a předává klastru potřebné informace. Třetí sekcí funkcionality je komunikace s obrazovým API. Tyto tři sekce jsou rozděleny do dvou balíčků, první balíček **kubernetes** slouží ke

komunikaci s Kubernetes klastrem a druhý balíček zastřešuje servisní třídy pro provádění CRUD operací klasifikátorů a modulů. Dále obsahuje servisní třídy pro komunikaci s obrazovým API.

3.1.2.1 Servisní třídy – CRUD operace

Metody servisních tříd, které slouží pro CRUD operace nejdříve načtou data z databáze a provedou případnou transformaci těchto objektů na DTO objekty[9]. Ty slouží k přesunu dat mezi komponentami systému. Během vytváření nových klasifikátorů a modulů dochází k ověření správnosti zadaných dat. Jména klasifikátorů i modulů musí být unikátní v rámci množiny uživatelových nasazených komponent, dále dochází ke kontrole správnosti zadaných atributů pro klasifikátor na základě vybraného typu a k dalším kontrolám vstupních dat.

Další funkcionalitou kontejnerového API je přijetí a uložení trénovacích dat pro klasifikátory. Kontejnerové API má definované souborové úložiště, kam ukládá přijatá data do stromové hierarchie. Kořenový adresář obsahuje adresáře pro jednotlivé uživatele, a tyto adresáře obsahují podadresáře podle jmen klasifikátorů. Tato data jsou posléze načtena klasifikátory, během jejich nasazení.

3.1.2.2 Servisní třídy – komunikace s obrazovým API

Kontejnerové API během vytváření nového uživatelského modulu ověřuje zadaná data a kontroluje, zdali byly přijaté všechny potřebné informace pro vytvoření nového modulu. Dále na základě operace rozhodne, jaký kontroler obrazového API má zavolat. Možnými operacemi jsou

- Vytvoření nového modulu.
- Aktualizace stávajícího modulu.
- Smazání modulu.
- Vrácení se k existujícímu modulu.

Obrazové API na základě zvolené operace sestaví modul a nastaví jej jako aktuální verzi. V případě operace mazání modulu tento modulu smaže a nenastaví žádnou verzi modulu jako aktivní.

Komunikace s obrazovým API probíhá skrze jeho REST API, kdy klientská aplikace zasílá požadavky pomocí třídy **RestTemplate**. Ta jako parametry přijímá URL kontroleru, kam požadavek odeslat, tělo požadavku nebo například potřebné hlavičky.

[9]Data Transfer Object

3.1.2.3 Kubernetes Java API

Kubernetes podporuje několik možností, jak komunikovat s kontrolery uvnitř klastru. Standardním a nejpoužívanějším způsobem je komunikace skrze aplikaci příkazové řádky zvané kubectl. Další možností je využití REST API, kdy uživatel zasílá REST požadavky na kontrolery klastru. Poslední možností je využití aplikačních API pro podporované programovací jazyky. Všechny tyto možnosti v jádru pracují s REST API Kubernetes klastru.

Pro připojení ke klastru pomocí aplikačního API lze zvolit jeden z podporovaných programovacích jazyků a stáhnout aplikační API jako závislost do projektu. Mezi podporované jazyky patří například Java, go, Haskel nebo také velmi oblíbený Python. Pro komunikaci mezi kontejnerovým API a Kubernetes klastrem bylo zvoleno Kubernetes Java API. Toto rozhraní lze přidat jako závislost pomocí sestavovacího a balíčkovacího nástroje Maven, který tuto závislost stáhne do projektu z centrálního úložiště. Veškeré operace jsou prováděny v rámci Kubernetes klastru. Kontejnerová aplikace slouží pouze pro zadávání požadavků. Po vytvoření požadavku je tento požadavek přijat kontrolerem uvnitř klastru a zapsán do ETCD databáze.

Pro připojení ke klastru slouží metoda **setDefaultApiClient(config)** třídy **Configuration**. Tato metoda přijímá jako parametr výchozího klienta. Výchozí klient hledá kubeconfig konfigurační soubor, jehož lokace je uložená v proměnné *$KUBECONFIG* nebo v adresáři *$HOME/.kube/config*. Pokud si uživatel přeje využít jiný konfigurační soubor, může využít připravenou metodu **setApiClient()** třídy **Configuration**. Tato metoda přijme jako parametr objekt třídy **Config**, která v sobě udržuje odkaz na kubeconfig konfigurační soubor načtený z uživatelem zadané cesty.

```
import io.kubernetes.client.util.Config
import io.kubernetes.client.Configuration

private fun configureConnection() {
  val client = Config.fromConfig(kubeConfigPath)
    Configuration.setDefaultApiClient(client)
    val api = CoreV1Api()
    return api
}
```

Ukázka zdrojového kódu 3.3: Příklad nastavení konfigurace pro připojení ke klastru Kubernetes

Pro komunikaci s Kubernetes API je v rámci kontejnerové aplikace vytvořena servisní třída **IKubernetesService**, která celý proces nasazení nového klasifikátoru nebo modulu zastřešuje. Ve chvíli, kdy uživatel zadá v klientské aplikaci pokyn k nasazení, je této servisní třídě zadán požadavek. Ta následně volá metody jednotlivých napojených tříd, které vytvářejí objekty potřebné pro nasazení klasifikátoru nebo modulu a předávají je klastru pomocí Java API. Servisní třída **KubernetesService** nejdříve získá jmenný prostor

uvnitř klastru, ve kterém má být nasazení provedeno, následně zavolá metody servisní třídy starající se o vytvoření Kubernetes Service a poté metody servisní třídy, které vytvoří objekt Kubernetes Deployment. Díky tomu, že je Kubernetes klastru bezstavový, není důležité, zdali je nejdříve vytvořen objekt Kubernetes Service nebo objekt Kubernetes Deployment.

Objekt Kubernetes Service v rámci Kubernetes Java API představuje třída **V1Service**. Tato třída v sobě drží všechny potřebné informace včetně nastavení portu na kterém běží, určení selektoru a také Ambasador anotace které jsou potřeba pro správné směrování požadavků. Pro komunikaci s klastrem a vytváření nových objektů typu Service slouží třída **CoreV1Api** která pomocí metody **createNamespacedService** zadá požadavek klastru na vytvoření nové Kubernetes Service.

```
private fun createServiceSpec(selectorApp: String, servicePort:
    Int, servicePortName: String, targetPort: String):
    V1ServiceSpec {
    val spec = V1ServiceSpec()
    spec.selector = HashMap()
    spec.selector["app"] = selectorApp
    spec.selector["id"] = selectorApp

    val port = V1ServicePort()
    port.port = servicePort
    port.name = servicePortName
    port.targetPort = IntOrString(targetPort)
    spec.ports = Arrays.asList(port)
    return spec
}
```

Ukázka zdrojového kódu 3.4: Konfigurace portu a selektoru pro nový objekt Kubernetes Service

Poté co je vytvořena instance Kubernetes Service, iniciuje servisní třída vytvoření nového objektu Kubernetes Deployment. Pro vytvoření slouží metoda **createNamespacedDeployment** třídy **ExtensionsV1beta1Api**. Ta přijímá jako parametr jmenný prostor, kde má k nasazení dojít a také objekt třídy **ExtensionsV1beta1Deployment**. Tento objekt obsahuje šablonu pro vytvoření jednotlivých Podů. Tato šablona je instancí třídy Kubernetes Java API **V1PodTemplateSpec** a jsou v ní uvedeny informace o selektoru, skrze který jsou vytvořené Pody propojeny s objektem typu Kubernetes Service, který na ně následně směruje požadavky. Dalším atributem tohoto objektu je instance třídy **V1Container**, kde je během nasazení nastaven identifikátor Docker obrazu, ze kterého má být během nasazení vytvořen kontejner společně s informacemi o portu a systémových omezeních pro vytvořený kontejner. Během sestavení šablony kontejneru jsou také připojena média obsahující trénovací data.

Jednotlivé šablony kontejnerů mají také definované servisní proměnné prostředí, ve kterých jsou uvedeny identifikátory klasifikátoru. Po startu kontej-

neru začne klasifikátor proces učení z trénovacích dat. Během učení může potřebovat další informace uvedené v databázi, proto díky identifikátoru uvedeném v této proměnné prostředí dokáže sestavit dotaz do databáze a získat potřebná data určená právě pro něj.

```
private fun createTemplateContainers(deploymentName: String,
    imageId: String, connectionPort: Int, networkId: String,
    train_data_path: String): List<V1Container> {
    val container = V1Container()
    container.name = deploymentName
    container.image = imageId

    val port = V1ContainerPort()
    port.name = "http-api"
    port.containerPort = connectionPort


    val portSwagger = V1ContainerPort()
    portSwagger.name = "api-swagger"
    portSwagger.containerPort = 8080
    container.ports = Arrays.asList(port, portSwagger)

    val networkIdEnv = V1EnvVar()
    networkIdEnv.name = "NETWORK_ID"
    networkIdEnv.value = networkId

    container.env = Arrays.asList(networkIdEnv)

    val resource = V1ResourceRequirements()
    resource.limits = HashMap()
    resource.limits["cpu"] = Quantity("0.1")
    resource.limits["memory"] = Quantity("1Gi")
    resource.requests = HashMap()
    resource.requests["memory"] = Quantity("1Gi")
    container.resources = resource
    return Arrays.asList(container)
}
```

Ukázka zdrojového kódu 3.5: Konfigurace Docker obrazu, proměnných prostředí a omezení Podu při vytváření nového objektu typu Kubernetes Deployment

Po vytvoření Kubernetes Deployment vrátí kontejnerové API klientské aplikaci zprávu o úspěšnosti operace. Klientská aplikace následně tuto zprávu zobrazí uživateli.

Servisní třída **KubernetesService** slouží také k provedení smazání existujícího nasazení klasifikátoru nebo uživatelského modulu. Implementace rozhraní **IKubernetesService** nejdříve v klastru vyhledá existující objekt Kubernetes Deployment podle zvoleného selektoru a následně pomocí metody **deleteNamespacedDeployment** rozhraní **ExtensionsV1beta1Api** pro-

vede jeho smazání. Stejným způsobem je smazán i objekt Kubernetes Service a to pomocí metody **deleteNamespacedService** rozhraní **CoreV1Api**.

3.1.2.4 REST rozhraní

Kontejnerové API slouží jako hlavní zdroj dat pro klientskou aplikaci a jako hlavní administrační rozhraní pro komunikaci uživatele se systémem. Aplikace nabízí několik REST kontrolerů pro práci s uživatelskými daty, instancemi klasifikátorů a uživatelských modulů. Každý kontroler obsahuje několik metod, které zpracovávají požadavky, provádějí potřebné operace a následně vrací požadovaná data. Výsledná data jsou doplněna o HTTP kódy, podle kterých klientská aplikace rozpozná, zdali byla daná operace úspěšná. Pro vytvoření jednotlivých kontrolerů byla využita funkcionalita knihovny spring-web, konkrétně spring-boot-web[20]. Tato knihovna nabízí anotace s jejichž pomocí lze rychlý způsobem vytvořit kontroler, který následně volá operace nabízené třídami nižších vrstev.

```
@RestController
@RequestMapping("/api/v1/user/{idUser}/network")
class NetworkController {

    ...

    @GetMapping("/{idNetwork}")
    fun getNetworkDetail(@PathVariable("idUser") idUser: Long,
        @PathVariable("idNetwork") idNetwork: Long):
        ResponseEntity<Network> {
        logger.info("get specific network " + idNetwork + " for
            user " + idUser)
        val user = userRepository.findById(idUser)
        if (user.isPresent) {
            val network = networkRepository.findByIdAndUserId(
                idNetwork, idUser)
            if (network.isPresent) {
                return ResponseEntity(network.get(), null,
                    HttpStatus.OK)
            }
           logger.info("network not found" + idNetwork)
           return ResponseEntity.notFound().build()
        }
        logger.info("user not found " + idUser)
        return ResponseEntity.notFound().build()
    }
}
```

Ukázka zdrojového kódu 3.6: Kontroler kontejnerového API pro získání informací o konkrétním klasifikátoru

Jednotlivé kontrolery kontejnerového API jsou rozděleny podle typů objektů, se kterými pracují. Každý kontroler podporuje standardní CRUD operace. Následující seznam popisuje jednotlivé operace, které kontrolery nabízejí.

- **ModuleController** – tento kontroler podporuje CRUD operace uživatelských modulů, dále nabízí metody pro vytvoření nového nasazení modulu v rámci Kubernetes klastru, jeho smazání a získání aplikačních log záznamů modulu.
- **NetworkController** – kontroler obsluhující CRUD operace klasifikátorů společně s metodami pro získání aplikačních log záznamů, vytvoření a zrušení nasazení klasifikátoru v rámci Kubernetes klastru. Dále podporuje vytvoření nových parametrů klasifikátorů s metodami pro příjem trénovacích dat a textových parametrů.
- **NetworkTypeController** – tento kontroler vystavuje metodu pro zjištění všech typů klasifikátorů, které systém podporuje.
- **UserController** – tento kontroler podporuje CRUD operace s daty o uživatelích.

Pokud se po přijetí požadavku vyskytne neočekávaná chyba, kterou kontejnerové API nedokáže zpracovat dojde k vyhození výjimky. Tuto vytvořenou výjimku následně zachytí třída **CustomExceptionHandler**, která ji zpracuje a skryje uživateli, nebo vrátí obecné informace o chybě. Samotný text výjimky nebo další informace však neposkytuje.

3.2 Realizace obrazového API

Uživatelské moduly slouží jako funkcionalita pro přidání uživatelských operací navázaných na jednotlivé klasifikátory. Zdrojový kód modulů je předán obrazovému API, které z něj vytvoří Docker obraz a ten následně nahraje do privátního Docker repozitáře. Obrazové API je samostatná aplikace napsaná v jazyce Kotlin s využitím funkcionality projektu Spring Boot[21]. Tato aplikace přijímá požadavky pouze od kontejnerového API pomocí REST rozhraní.

3.2.1 Vytvoření obrazu z uživatelského kódu

Vytvoření Docker obrazu, který je následně nasazen v rámci Kubernetes klastru lze provést dvěma způsoby. Prvním způsobem je zadání zdrojového kódu do klientské aplikace, dále je podporovaná možnost vložit odkaz na Git repozitář.

3.2.1.1 Vytvoření obrazu modulu pomocí klientské aplikace

Nejjednodušší možností vytvoření nového modulu je pomocí klientské aplikace. Uživatel do online editoru uvnitř klientské aplikace zadá zdrojový kód funkční části modulu, který musí splňovat předepsané rozhraní. Po odeslání

tohoto zdrojového kódu instruuje kontejnerové API obrazové API, které převezme zdrojový kód uložený v databázi a začne proces vytvoření modulu. Jako první krok vytvoří obrazové API Git repozitář ze zdrojových kódů uživatele a odešle jej na Git server. Tento repozitář následně slouží jako záloha, pro případ výpadku dat a také bude případně uživateli umožněno vrátit se k předchozí verzi zdrojových kódů. Pro operace s Git repozitáři slouží servisní třída GitService, která umožňuje vytvoření nového repozitáře, jeho aktualizaci nebo klonování již existujícího repozitáře.

Obrazové API má v konfiguračním souboru uložen odkaz na šablonu, která je skládána s uživatelským kódem. Tato šablona obsahuje Python Flask REST API, které slouží k přijímání požadavků, kontrole zabezpečení a volání uživatelského kódu. Po přijetí požadavku od klasifikátoru předá API šablony tento požadavek uživatelskému kódu. Data požadavku se skládají z informací od klasifikátoru, kde se nachází hlavní třída klasifikace, která byla určena, originální data požadavku, který byl zaslán klasifikátoru a meta-informace o požadavku.

Po stažení šablony do uživatelského adresáře je šablona spojena s uživatelským zdrojovým kódem a začne proces vytváření Docker obrazu. Pro práci s Docker API je použita knihovna **docker-client** od společnosti Spotify. Ta nabízí veškerou funkcionalitu potřebnou pro úspěšné nahrání obrazu na Docker repozitář. Šablona spojená s uživatelským kódem obsahuje Dockerfile, který slouží k sestavení obrazu. Tento Dockerfile je použit pro sestavení obrazu, který je následně označkován (tag) a nahrán do repozitáře.

```
override fun buildImage(workdir: String, module: Module) {
    val imageName = "some_name"
    val docker: DockerClient = DefaultDockerClient.fromEnv().
        build()

    val imageIdFromMessage = AtomicReference<String>()
    val param = DockerClient.BuildParam("EXPOSE_PORT", port)

    val returnedImageId = docker.build(
            Paths.get(workdir), imageName, ProgressHandler {
                message ->
                val imageId = message.buildImageId()
                if (imageId != null) {
                    imageIdFromMessage.set(imageId)
        }
      }, param)

    docker.tag(imageName, dockerRepository + imageName)
    docker.push(dockerRepository + imageName)
}
```

Ukázka zdrojového kódu 3.7: Vytvoření nového Docker obrazu pomocí knihovny Docker-client

Po vytvoření Docker obrazu, upraví obrazové API status o daném modulu v databázi na vytvořený a uživatel následně může provést jeho nasazení. Je-

likož celý proces vytvoření trvá několik sekund, je tato operace volána kontejnerovým API asynchronně a API nečeká na odpověď. Odpovědí je aktualizace stavu modulu v databázi.

3.2.2 Vytvoření obrazu z Git repozitáře

Pokud uživatel nepoužije online editor uvnitř klientské aplikace, může předat odkaz na Git repozitář, ze kterého bude následně použit jeho zdrojový kód. Tento zdrojový kód musí stejně jako při využití online editoru splňovat rozhraní, aby s ním mohlo API uvnitř šablony komunikovat. Po stažení zdrojového kódu z Git serveru, je tento kód opět spojen se šablonou a další kroky jsou ekvivalentní postupu při němž je využit online editor uvnitř klientské aplikace.

3.3 Realizace klientské aplikace

Klientská aplikace slouží ke komunikaci uživatele s ostatními komponentami systému. Aplikace zobrazuje uživateli požadovaná data a odesílá požadavky na kontejnerové API. Při vývoji této komponenty byl využit Javascriptový framework Angular ve verzi 6, společně s knihovnou Angular Material. Celá aplikace je rozdělena do modulů, které mají různou funkcionalitu. Hlavní modulem je *root* modul, který slouží k navigaci mezi jednotlivými moduly[22]. Dále udržuje závislosti mezi moduly aplikace a potřebnými knihovnami a slouží jako hlavní přístupový bod při vstupu do klientské aplikace. Další moduly v sobě udržují funkcionalitu pro jednotlivé logické části a obsahují pod moduly[23]. Jednotlivé moduly jsou následující:

- Autorizační modul – **authorization**.
- Modul klasifikátorů – **network**.
- Modul uživatelských modulů – **module**.
- Modul sdílené funkcionality – **shared**.
- Modul aktuálně přihlášeného uživatele – **user**.

3.3.0.1 Autorizační modul

Autorizační modul slouží k uchovávání informací o aktuálním uživateli, jeho identifikátoru a JWT tokenu, který slouží jako autorizace uživatele vůči kontejnerovému API. Dále nabízí metodu **getLoggedUser**, která vrací informace o uživateli a je využita v HTTP servisních třídách.

3.3.0.2 Modul klasifikátorů

Tento modul slouží k vytváření nových instancí klasifikátorů, zobrazení jejich přehledu, detailu a zjišťování log záznamů. Modul se skládá z několika komponent. Pro komunikaci s kontejnerovým API mají všechny moduly servisní třídu, která využívá komponentu **HttpClient** projektu Angular. Tato komponenta odesílá HTTP požadavky na požadovanou URL adresu a vrací objekt třídy **Observable**. Třída **Observable** slouží jako obálka, ve které se nachází požadovaná data. Komponenta modulu odešle pomocí servisní třídy požadavek na kontejnerové API a následně provede operaci **subscribe** objektu **Observable**, který jí servisní třída vrátí. Ve chvíli kdy kontejnerové API vrátí data, jsou tato data předána komponentě, která s nimi posléze dále pracuje[24].

```
@Injectable()
export class NetworkService {
    private handleError: HandleError;

    constructor(private http: HttpClient, httpErrorHandler:
        HttpErrorHandler) {
        this.handleError = httpErrorHandler.createHandleError('
            UsersService');
    }


    getNetworks(id: number): Observable<Network[]> {
        return this.http.get<Network[]>(API_ROOT_DOMAIN + '/
            network/')
        .pipe(
             catchError(this.handleError('getNetworks', []))
        );
    }
}
```

Ukázka zdrojového kódu 3.8: Ukázka servisní třídy pro získání dat z kontejnerového API

```
@Component({
  selector: 'app-network-list',
  templateUrl: './network-list.component.html',
  styleUrls: ['./network-list.component.css']
})
export class NetworkListComponent implements OnInit {
    constructor(private networkService: NetworkService) {}

    loadUserNetworks() {
        this.networkService.getNetworks(this.user.id).subscribe(
            networks => this.networks = networks
        );
    }
}
```

Ukázka zdrojového kódu 3.9: Ukázka komponenty, která čeká a konzumuje

data přijatá servisní třídou

Kromě servisní třídy obsahuje tento modul i komponenty pro zobrazení dat a komunikaci s uživatelem. Každá komponenta se skládá z několika souborů a jednotlivé komponenty jsou hierarchicky členěné. Hlavní funkcionalitou je zobrazení informací o již vytvořených klasifikátorech, o možnostech k jejich připojení a zobrazení jejich aplikačních log záznamů. Dále tento modul obsahuje komponentu s formulářem pro vytvoření nového klasifikátoru.

3.3.0.3 Modul uživatelských modulů

Tato část klientské aplikace obsahuje funkcionálně velmi podobné komponenty jako modul klasifikátorů. Oproti modulu klasifikátorů je zde umístěn online editor, který slouží k zadání uživatelského zdrojového kódu. Jako editor byl použit **ace-editor** ze stejnojmenné knihovny komponent. Tento editor nabízí mnoho druhů barevných stylů a podporuje velkou škálu programovacích jazyků. Pro využití tohoto editoru je potřeba pouze jeho instalace a posléze jej lze přidat pomocí selektoru **ace-editor** jako jakoukoliv jinou komponentu.

```
<div ace-editor
    [(text)]="module.code"
    [mode]="'python'"
    [theme]="'monokai'"
    [options]="options"
    [readOnly]="false"
    [autoUpdateContent]="true"
    [durationBeforeCallback]="1000"
    (textChanged)="onChange($event)">
</div>
```

Ukázka zdrojového kódu 3.10: využití a konfigurace komponenty ace-editor

Jedním z atributů tohoto editoru je sekce *options*. Ta obsahuje nastavení editoru jako je maximální délka řádků nebo zvýraznění kurzoru. Při editaci zdrojového kódu uvnitř editoru dojde k aktivaci funkce definované v atributu *textChanged* a uživatelem zadaný zdrojový kód je v proměnné *module.code*. Při uložení modulu jsou uživatelem zadané informace, společně se zdrojovým kódem zaslány na kontejnerové API ve formátu base64.

3.3.0.4 Modul aktuálně přihlášeného uživatele

Tento modul obsahuje komponenty pro editaci a zobrazení informací o aktuálně přihlášením uživateli.

3.3.0.5 Modul sdílené funkcionality

Tento modul obsahuje komponenty, které jsou využívané napříč celou aplikací. Jedná se například o editační roury, které modifikují vložený řetězec, nebo na jeho základě vracejí jiný textový řetězec. Na příkladu níže je ukázka roury

která na základě hodnot číselníku vrací textový řetězec, který je následně zobrazen uživateli.

```
@Pipe({name: 'moduleTypePipe'})
export class ModuleTypePipe implements PipeTransform {

    transform(value: number): string {
        if (value === 0) {
            return 'lambda';
        }else if (value === 1) {
            return 'git repository';
        }else if (value === 2) {
            return 'docker image';
        }
        return 'unknown';
    }
}
```

Ukázka zdrojového kódu 3.11: Ukázka definice roury pro transformaci číselníkových hodnot na textové řetězce

3.4 Implementace společné funkcionality pro klasifikátory

Jednotlivé klasifikátory jsou nasazeny jako samostatné aplikace v rámci Kubernetes klastru a mají nastavený port, na kterém přijímají požadavky od uživatele. Každý klasifikátor se skládá ze REST API, které má připravené tři kontrolery[25]. První kontroler slouží pro kontrolu stavu klasifikátoru a je využíván klientskou aplikací pro zobrazení informací o tom, zdali nasazení klasifikátoru v rámci klastru proběhlo v pořádku. Tento kontroler může využít i uživatel pro vlastní monitorování aplikace bez nutnosti odesílat autentifikační token. Druhý kontroler slouží pro zobrazení dokumentace daného klasifikátoru. Každý klasifikátor má připravenou Swagger dokumentaci, skrze kterou může být testován uživatelem. Tato dokumentace je připojená ke klientské aplikaci nebo ji lze navštívit na samostatné adrese.

Posledním kontrolerem nabízeným instancí klasifikátoru je kontroler klasifikace. Pro úspěšné zpracování požadavku odeslaného na tento kontroler je nutné, aby uživatel zaslal společně s požadavkem autentifikační JWT token, který slouží pro kontrolu uživatelovy identity. Každý klasifikátor má během vytváření nastavenou proměnnou prostředí, kde jsou informace o identifikátoru daného klasifikátoru. Tento identifikátor je propojen s tabulkou obsahující informace o uživateli a připojených modulech. Při každém požadavku uživatele dojde ke kontrole zdali je zaslaný token správně podepsaný a zdali jsou informace uvnitř tokenu stejné, jako je identifikátor sítě a identifikátor uživatele. Tento token uživatel získá v klientské aplikaci.

Po úspěšné autentizaci uživatele dojde k samotné klasifikaci přijatých dat. Neuronové sítě uvnitř všech typů klasifikátoru mají stejné rozhraní a struktura

jejich odpovědi je také téměř totožná. Rozhraní kontrolerů a komunikace kontrolerů s neuronovou sítí je tedy stejné pro všechny typy klasifikátorů. Poté co neuronová síť oklasifikuje přijatá data, vrátí odpověď v podobě slovníku zpět kontroleru. Ten poté načte všechny připojené moduly a zkontroluje, zdali není některý z modulů připojený skrze hlavní třídu klasifikace vrácené neuronovou sítí. Pokud ano, odešle na něj požadavek pomocí POST metody a vrácená data z modulu přidá k datům neuronové sítě do slovníku, který následně vrátí uživateli.

```
curl -X POST
     -H "Authorization: eyJ ... I5w"
     -H "Content-Type: application/json"
     --data '{"message":"Hello world"}'
     "localhost/user-network/network/predict"
```

Ukázka zdrojového kódu 3.12: Odeslání požadavku na klasifikátor pomocí nástroje curl

3.5 Implementace a měření klasifikátoru log záznamů

3.5.1 Příprava dat pro neuronovou síť

Příprava log souborů pro zpracování neuronovou sítí obsahuje několik kroků. Nejdříve je potřeba jednotlivé záznamy rozdělit na slovníky. Na rozdělení záznamů je použita knihovna Pygrok, která pro každý záznam log souboru vrátí mapu obsahující jména položek jako klíče a text jako hodnoty. Následující ukázka kódu demonstruje použití knihovny Pygrok pro rozdělení jednoduchého log záznamu.

```
pattern = "%{IP:client} %{WORD:method} %{URIPATHPARAM:request} %{
    NUMBER:bytes} %{NUMBER:duration}"
row = '55.3.244.1 GET /index.html 15824 0.043'
grok = Pygrok(pattern)
result = grok.search(row)

result =>  {'client': '55.3.244.1',
      'method': 'GET',
      'request': '/index.html',
      'bytes': '15824',
      'duration': '0.043'}
```

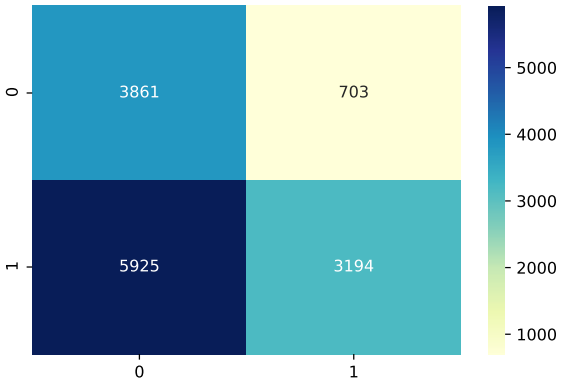
Ukázka zdrojového kódu 3.13: Využití knihovny Pygrok

Z každého záznamu jsou odstraněny speciální znaky a sloupce, které obsahují nepotřebné hodnoty. Následně je z celé datové sady vytvořen znakový bigram (character bigram). Pro transformaci datové sady na znakový bigram slouží **CountVectorizer** z knihovny **scikit learn**. Ze vstupní datové sady takto vznikne znakový bigram, který lze předat neuronové síti jako trénovací data.

3.5.2 Implementace a měření neuronové sítě

Po přípravě trénovacích dat začne proces učení modelu neuronové sítě. Tento model využívá architekturu Kohonenových map a učení bez učitele. Na základě velikosti trénovacích dat dojde k vypočtení počtu iterací učícího algoritmu a velikosti vstupní vrstvy neuronové sítě. Během učení se pro každý záznam z trénovacích dat vypočte vítězný neuron, který je společně s jeho okolím posunut blíže ke vstupnímu vektoru. Velikost okolí, které bude posunuto určuje parametr sousedské funkce *sigma*.

Pro shlukování log záznamů a hledání anomálií byl vytvořen model neuronové sítě s mřížkou velikosti 2×2 se sousedskou funkcí nastavenou na hodnotu 1. Tento model byl naučen na 13 000 záznamech z apache acccess log souboru. Následně proběhlo měření nad tímto naučeným modelem. Učení obsahovalo 800 iterací. Pro jednotlivé záznamy byl vypočítán vítězný neuron, který byl uložen do mapy výskytů (heatmap) zobrazené na obrázku 3.2.



Obrázek 3.2: Mapa výskytů příslušnosti vstupních dat k neuronu klastru velikosti 2×2

Pro model byla vypočtena také matice záměn (confusion matrix).

	abnormální	normální
abnormální	TP	FN
normální	FP	TN

Kde

- **TP** – True Positive jsou abnormální log záznamy, které jsou správně klasifikované jako abnormální.
- **FN** – False Negative jsou abnormální log záznamy, které jsou špatně klasifikované jako normální.

- **FP** – False Positive jsou normální log záznamy, které jsou špatně klasifikované jako abnormální.
- **TN** – True Negative jsou normální log záznamy, které jsou správně klasifikované jako normální.

Jednotlivé záznamy byly ručně ohodnoceny, aby bylo možné hodnotit úspěšnost modelu pomocí F-score. Z matice změn byly dále vypočteny metriky **recall** a **precision**. Metrika **recall** indikuje počet abnormálních log záznamů v datové sadě, které byly správně ohodnoceny. Pokud má model vysokou hodnotu metriky recall, je menší pravděpodobnost, že bude klasifikovat abnormální log záznamy jako normální. Metrika **precision** pro změnu ukazuje poměr abnormálních log záznamů ve všech log záznamech, které byly klasifikovány jako abnormální.

$$Recall, r = \frac{TP}{TP + FN}$$
$$Precision, p = \frac{TP}{TP + FP}$$

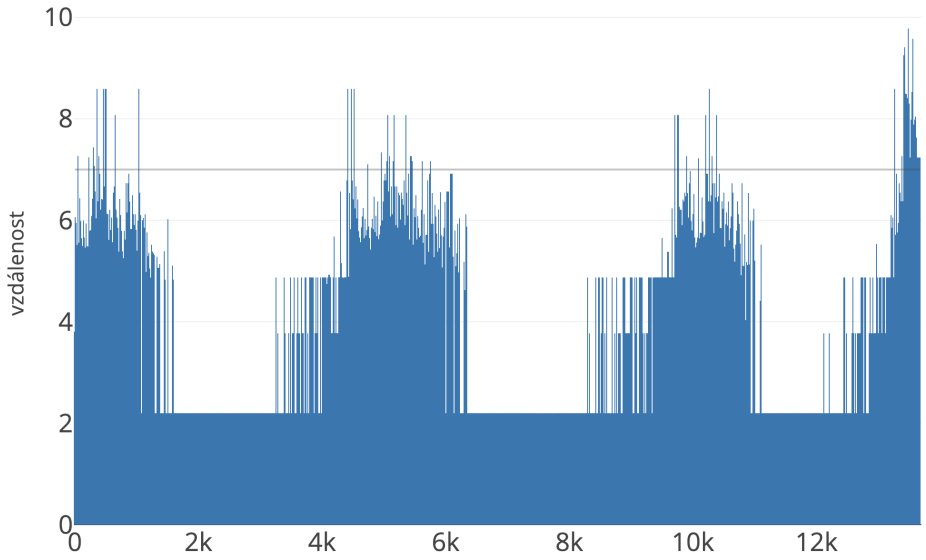
Pokud má model vysokou hodnotu recall a nízkou hodnotu precision, znamená to, že téměř všechny log záznamy klasifikuje jako abnormální, což není požadovaný stav. Stejně tak pokud má model vysokou hodnotu metriky precision, ale nízkou hodnotu metriky recall. **F-score** je metrika kombinující výsledky metrik recall a precision. Počítá se jako

$$F - score = \frac{2rp}{r + p} = \frac{2 \cdot TP}{2 \cdot TP + FP + FN}$$

Vysoké F-score indikuje, že jak hodnota metriky recall, tak i hodnota metriky precision jsou vysoké.

Pro hodnocení přesnosti modelu byl zvolen algoritmus SOM-DIST[9]. Při klasifikaci modelu je využito předpokladu, že abnormálních log záznamů je ve vstupní datové sadě mnohem méně než log záznamů normálních. Model se tedy na tyto log záznamy nedokáže dostatečně přizpůsobit. Naučený model je tedy nechán ohodnotit (přiřadit vítězný neuron) pro všechny log záznamy ze vstupní datové sady pomocí metody **winner()** a následně je zjištěn vektor vah vítězného neuronu[26]. Poté je vypočtena Euklidova vzdálenost mezi váhami vítězného neuronu a vstupním vektorem klasifikovaného log záznamu. Log záznamy, které mají tuto vzdálenost větší, než je určitá specifikovaná konstanta, jsou označeny jako abnormální. Po prozkoumání grafu vzdáleností bylo konstanta vzdálenosti nastavena na hodnotu 7. [10]

[10]Při nasazení model určuje hodnotu vzdálenosti pro abnormální log záznamy programově tak, že vypočte průměrnou vzdálenost ze 3 % záznamů seřazených podle vzdálenosti. Množina obsahující 3 % záznamů s největší vzdáleností od vítězného neuronu obsahuje nejvíce abnormálních log záznamů. Hodnota této vzdálenosti pak slouží jako práh. Všechny další přijaté záznamy, jejichž vzdálenost od vítězného neuronu je vyšší než tento práh jsou následně označeny jako abnormální.



Obrázek 3.3: Jednotlivé vzdálenosti mezi vektorem vah vítězného neuronu a vektorem vstupního log záznamu

Všechny log záznamy, které měly vzdálenost od vektoru vah vítězného neuronu větší než 7, byly poté označeny jako abnormální. Matice změn a výpočet F-score je následující.

	abnormální	normální
abnormální	164	106
normální	40	13373

$$r_{SOM} = \frac{164}{164 + 106} = 0.61$$

$$p_{SOM} = \frac{164}{164 + 40} = 0.80$$

$$F - score_{SOM} = \frac{2 \cdot 164}{2 \cdot 164 + 40 + 106} = 0.69$$

Hodnota F-score vyšší než 0.8 je považována za velice dobrou, nižší 0.8, ale vyšší než 0.6 jakožto přijatelná. Hodnoty nižší než 0.6 jsou považovány za nedostatečné. SOM model s hodnotou 0.69 je tedy na klasifikaci log záznamů přijatelný.

Na stejné vstupní datové sadě bylo provedeno stejné měření, ale místo SOM byl použit KMEANS shlukovací algoritmus se dvěma klastry. Ten klastr, který obsahoval větší množství abnormálních log záznamů byl označen jako klastr abnormálních a všechny log záznamy v tomto klastru byly označeny také jako abnormální.

Jeho matice změn je následující

	abnormální	normální
abnormální	270	0
normální	10219	3194

$$F - score_{KMEANS} = \frac{2 \cdot 270}{2 \cdot 270 + 10219 + 0} = 0.05$$

Z výsledného F-score lze soudit, že tento postup je pro KMEANS nevhodný.

3.6 Implementace klasifikačního modelu pro účely chatbota

3.6.1 Předzpracování dat pro model

Pro účely jednoduchého chatbota je potřeba, aby uživatel definoval, jaké otázky budou tomuto chatbotovi pokládány a jaké odpovědi má chatbot vracet. Model klasifikátoru si následně tyto vstupní otázky převede do číselného formátu a začne proces učení. Vstupní data jsou od uživatele přijata v podobě JSON, kde se jednotlivé záznamy skládají ze tří, případně čtyř položek. První položkou je **tag**. Ten slouží jako identifikátor položky v rámci vstupních dat a také jako třída klasifikace, kterou bude vracet neuronová síť. Další položkou je pole možných dotazů v sekci **patterns**. Každý záznam v tomto poli reprezentuje vzor otázky pro neuronovou síť. Z těchto dat následně probíhá učení neuronové sítě. Další položkou jsou **responses**. Ve chvíli, kdy neuronová síť označí vstupní text určitou třídou klasifikace, je z položek uvnitř odpovědí náhodně vybraná jedna odpověď, která je vrácena uživateli. Další položky slouží pro nastavení a vyhledávání kontextových informací.

```
{
   "intents":[
      {
         "tag":"greeting",
         "patterns":["Hi", "How are you", ... ],
         "responses":["Hello, thanks for visiting", ...],
         "context_set":""
      },{
         "tag":"hours",
         "patterns":["What hours are you open?", ... ],
         "responses":["Our hours are 9am-9pm every day", ...],
         "context_filter":"openhours"
      },{
         "tag":"rental",
         "patterns":["Can we rent a car?", "I'd like to rent a
             car" ... ],
         "responses":["Are you looking to rent today or later
             this week?"],
         "context_set":"rentalday"
      },{
```

```
            "tag":"today",
            "patterns":["today"],
            "responses":["For rentals today please call"],
            "context_filter":"rentalday"
        },{
            "tag":"later",
            "patterns":["later"],
            "responses":["Next-day rentals please come"],
            "context_filter":"rentalday"
        }
    ]
}
```

Ukázka zdrojového kódu 3.14: Trénovací data pro chatbota, společně s kontextuálními proměnnými a připravenými odpověďmi

Po přijetí trénovacích dat začne model s přípravou vstupních vět. Vstupní data musejí být převedena do číselné podoby, aby s nimi mohla pracovat neuronová síť. Nejprve je nutné vstupní věty rozdělit na slova. K tomu je využita funkce **word_tokenize** knihovny nltk. Ta rozdělí vstupní text na jednotlivá slova, která jsou poté převedena do základního tvaru.[11][27]. Například slova *fishing, fished, fisher* jsou převedena na slovo *fish*. Pro vstupní řetězce uvedené v předchozí ukázce budou jednotlivé kmeny slov vypadat následovně:

```
['hi', 'how', 'ar', 'you', 'what', 'hour', 'ar', 'you', 'op', '
    can', 'we', 'rent', 'a', 'car', 'today', 'lat']
```

Ukázka zdrojového kódu 3.15: Kmeny slov pro slova z předchozí ukázky

Pro převedení slov do základní formy využívá model funkcionalitu třídy **LancasterStemmer** z knihovny nltk. Ta nejdříve stáhne slovník *punkt*, který používá k nalezení kmene slova a následně každé slovo převede pomocí metody **stem(word)**.

```
import nltk
from nltk.stem.lancaster import LancasterStemmer
nltk.download('punkt')

ignore_words = ['?']
stemmer = LancasterStemmer()
words = []
sentences = ["Hi", "How are you", "Is anyone there?", ... ]
for sences in sentences:
    w = nltk.word_tokenize(sences)
    words.extend(w)

    words = [stemmer.stem(w.lower()) for w in words if w not in
        ignore_words]
```

Ukázka zdrojového kódu 3.16: Využití knihovny nltk pro převedení slov do základní formy

[11] stemming – česky stematizace je proces převedení slova na jeho základní tvar (kmen slova)

Po převedení slov do základní formy přichází na řadu jejich převedení do číselné formy. Existuje řada možností jak převést vstupní řetězce do číselné formy. Jednou z možností je převést vstupní řetězce na bigram znaků, stejně jako je provedeno u klasifikace log záznamů. Pro účely chatbota však více vyhovuje jednotlivá slova nerozdělovat a využít model zvaný **bag-of-words**[28].

Bag-of-words model je jednou z technik NLP [12], která slouží k převodu psaného textu na vektor číselných hodnot. Každé slovo v základním tvaru se přidá do množiny slov tak, že vznikne slovník obsahující jednotlivá unikátní slova z celé vstupní datové sady. Následně je pro každou větu ve vstupní datové sadě vytvořen vektor, který má délku slovníku a obsahuje 1 pokud se dané slovo nachází ve větě a 0 pokud ne. Pro věty *„I'd like to rent a car“* a *„Can we rent a car?“* ze vstupního řetězce tak vzniknou následující vektory.

	I	would	like	to	rent	a	car	can	we
s1	1	1	1	1	1	1	1	0	0
s2	0	0	0	0	1	1	1	1	1

Takto jsou zpracovány všechny věty ze vstupní datové sady a následně jsou předány neuronové síti k naučení.

3.6.2 Architektura a konfigurace neuronové sítě pro účely klasifikace textu

Neuronová síť uvnitř chatbot klasifikátoru má dopřednou architekturu skládající se ze tří vrstev. První vrstva je vstupní, má stejně neuronů jako je velikost slovníku a slouží k přijetí vstupních dat. Následuje jedna skrytá vrstva obsahující stejný počet neuronů. Poslední vrstva je výstupní, ta obsahuje tolik neuronů, kolik je klasifikačních tříd předaných uživatelem. Jako ztrátová funkce pro tuto neuronovou síť byla použita *categorical_crossentropy*. Tato ztrátová funkce se používá, pokud jsou vstupní data kategorická, tedy číselné hodnoty představují určitou kategorii, například jedna znamená, že se ve vstupní větě vyskytuje dané slovo[29].

```
chatbot_model = Sequential()
chatbot_model.add(Dense(self.num_of_tensors,
            input_shape=self.input_shape))
chatbot_model.add(Dense(self.num_of_tensors))
chatbot_model.add(Dense(self.output_shape,
          activation='softmax'))
chatbot_model.compile(loss='categorical_crossentropy',
            optimizer='adam',
            metrics=['acc'])


chkpt_callback = ModelCheckpoint(filepath=model_data_path,
                 monitor='acc',
```

[12]Natural language processing

```
                verbose=0, save_best_only=True,
                save_weights_only=False,
                mode='max', period=1)

chatbot_model.fit(train_x, train_y,
          epochs=num_of_epochs,
          batch_size=batch_size,
          callbacks=[chkpt_callback])
```

Ukázka zdrojového kódu 3.17: Vytvoření modelu chatbota pomocí knihovny Keras

3.6.3 Kontextualizace komunikace s chatbotem

K modelu chatbota byla přidána i funkcionalita kontextualizace. Pokud se například uživatel zeptá, zdali si může půjčit auto, model klasifikuje otázku třídou **rental** a přidá si do slovníku identifikátor uživatele společně s kontextem, který je uveden u této klasifikační třídy ve vstupních datech. Do kontextu si tedy přidá identifikátor uživatele a řetězec **rentalday**. Pro každou další otázku, oklasifikuje model odpověď určitými klasifikačními třídami a kontroluje, zdali určitá třída nemá jako **context_filter** právě řetězec **rentalday**. Pokud ano, vrátí odpověď pro tuto třídu, i kdyby nebyla v top-1 klasifikaci. Po pozdravení nebo rozloučení je kontext pro daného uživatele opět vynulován.

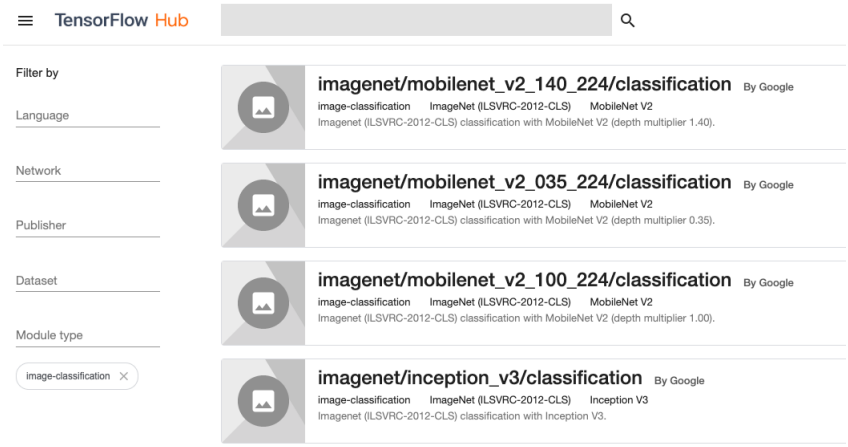
3.7 Implementace a měření modelů klasifikující obrázky

3.7.1 Implementace klasifikátoru přenosového učení

Při implementaci modelů neuronových sítí existuje celá řada řešení, která vývoj usnadňují. Na trhu se navíc nachází velké množství řešení, které nabízejí možnost klasifikovat obrázky bez nutnosti cokoliv programovat či instalovat. Tyto služby existují jako řešení Software jako služba, kdy uživatel pouze zašle svůj požadavek na rozhraní klasifikátoru, který mu následně vrátí popis toho co se na obrázku nachází. Stejný přístup byl zvolen i pro klasifikátory v systému Chobot. Pro implementaci samotného klasifikátoru pro přenosové učení na obrázkové datové sadě byly zvoleny projekty TensorFlow a TensorFlow Hub.

TensorFlow Hub je framework pro tvorbu modelů strojového učení, které lze znovu používat na stejná nebo podobná vstupní data. TensorFlow Hub nabízí metody pro zakonzervování nových modelů a využití již existujících modelů, které dodržují rozhraní předepsané v TensorFlow Hub. Díky tomu může programátor vybrat základní model, který následně využije jako extraktor příznaků a pouze přidá výstupní vrstvy a model doučí. Typ extraktoru

programátor zadá jako odkaz do zdrojového kódu a následně dojde k jeho stažení.



Obrázek 3.4: TensorFlow Hub

Díky stejnému rozhraní TensorFlow Hub modelů je jednoduché nahradit model za jiný a tím například měnit výkonnost klasifikátoru[30]. Výkonnost klasifikátoru je velmi ovlivněna výběrem extraktoru. Například extraktory skupiny ResNet jsou zaměřeny na kvalitní zpracování vstupních dat s vyšší přesností klasifikace, zatímco extraktory MobileNet naopak provádějí klasifikaci velmi rychle, ale s nižší přesností. Programátor tedy jen vybere, jaké má požadavky na extraktor a na jejich základě stáhne ten, který těmto požadavkům vyhovuje.

Pro sestavení klasifikátoru definuje programátor URL extraktoru a následně během sestavení TensorFlow grafu dojde k jeho stažení. Celý proces vytvoření nového modelu neuronové sítě s využitím již předtrénovaného extraktoru vlastností se tedy skládá z výběru vhodného extraktoru na stránkách TensorFlow-hub a jeho následného stažení. Poté se k extraktoru připojí výstupní vrstva.

```
def feature_extractor(x):
    feature_extractor_module = hub.Module(feature_extractor_url)
    return feature_extractor_module(x)


image_size = hub.get_expected_image_size(
            hub.Module(feature_extractor_url))
features_extractor_layer = layers.Lambda(
                feature_extractor,
                input_shape=image_size + [3]
              )

features_extractor_layer.trainable = False
```

```
model = tf.keras.Sequential([features_extractor_layer,layers.
    Dense(image_data.num_classes, activation='softmax')])
```

Ukázka zdrojového kódu 3.18: Získání extraktoru z TensorFlow Hub a přidání nové výstupní vrstvy

Jelikož jsou vrstvy extraktoru již předučené, je nutné deaktivovat možnost jejich následného učení. K připravenému extraktoru je nakonec přidána výstupní vrstva, která slouží k samotné klasifikaci. Tato vrstva obsahuje přesně tolik neuronů, kolik obsahuje vstupní datová sada klasifikačních tříd.

Po vytvoření modelu neuronové sítě je spuštěn proces jeho inicializace a učení. Inicializace se skládá z nastavení TensorFlow relace (session) a sestavení modelu. Inicializace relace slouží k zapouzdření operací prováděných v rámci zdrojového kódu TensorFlow, jelikož TensorFlow provádí pozdní vyhodnocení (lazy evaluation). Jednotlivé operace uvedené a definované programátorem ve zdrojovém kódu nejsou reálně provedeny s daty proměnných ihned, jelikož tyto proměnné slouží jako zástupné proměnné (placeholder), do kterých budou následně při inicializaci nahrána data. Po vytvoření relace a sestavení modelu dojde ke zpracování kódu a nahrání hodnot do zástupných proměnných. Během sestavení modelu je také zvolen Optimizer a chybová funkce. Chybová funkce počítá hodnotu aktuální chyby během procesu učení a Optimizer je implementace algoritmu, který má za úkol co nejvíce snížit hodnotu této funkce.

```
sess = K.get_session()
init = tf.global_variables_initializer()
sess.run(init)
model.compile(optimizer=tf.train.AdamOptimizer(), loss='
    categorical_crossentropy', metrics=['accuracy'])
```

Ukázka zdrojového kódu 3.19: Inicializace modelu neuronové sítě

Po inicializaci modelu neuronové sítě přichází na řadu proces učení výstupní vrstvy pomocí trénovacích dat. Neuronové sítě ve většině případů pracují s daty v podobě číselných hodnot. Je tedy potřeba, aby uživatelem předaná data byla převedena do číselné podoby a zároveň, aby proces předání trénovacích dat byl pro uživatele co nejpřívětivější. TensorFlow nabízí podporu funkcionalit knihovny Keras, která je zaměřena na uživatelsky přívětivé způsoby definice modelu neuronových sítí a přípravu trénovacích dat pro tyto modely. Jednou z metod, sloužících pro přípravu trénovacích dat je metoda **flowFromDirectory**, která načte obrázky uložené v podadresářích v předaném adresáři a jména těchto podadresářů slouží jako jména klasifikačních tříd.

```
image_generator = tf.keras.preprocessing.image.ImageDataGenerator
    (rescale=1 / 255)
image_data = image_generator.flow_from_directory(str(data_root),
    target_size=image_size)
```

Ukázka zdrojového kódu 3.20: Načtení a příprava dat pro učení neuronové sítě

Uživatel tedy pouze vytvoří adresář a následně v něm vytvoří podadresáře pojmenované podle tříd klasifikace a do nich vloží trénovací obrázky. Před procesem učení jsou tyto obrázky převedeny na matice obsahující číselné hodnoty a tyto matice jsou předány jako trénovací data. Při procesu učení s učitelem je vhodné rozdělit trénovací množinu na dvě podmnožiny, trénovací a testovací (validační). Na trénovací množině probíhá učení a testovací množina slouží k ověření, zdali nedochází k přeučení. Jakmile se začne křivka přesnosti na trénovacích datech výrazně vzdalovat od křivky přesnosti na testovacích datech, dochází zřejmě k přeučení modelu a je potřeba provést opravné kroky.

Učení neuronové sítě probíhá po epochách. Každá epocha znamená jedno provedení dopředného fáze učení a následné zpětné propagace chyby. Během každé epochy je neuronová síť učena na podmnožině obrázků. Velikost této podmnožiny určuje proměnná **batch**, která je nastavena po přijetí trénovacích dat. Čím více epoch je provedeno během učení a čím větší je hodnota proměnné batch, tím déle proces učení trvá. Během učení zobrazuje neuronová síť svoji aktuální přesnost a hodnotu ztrátové funkce. Po každé epoše, lze model uložit do samostatného souboru a tím zabránit v degradaci přesnosti například při přeučení modelu. Pro průběžné ukládání vah a konfigurace modelu je potřeba vytvořit objekt **callbacks**, který je předán učící funkci. Tento objekt obsahuje konfiguraci objektu třídy **ModelCheckpoint**, kde jsou uvedeny cesty pro uložené podmodely a konfigurace, co vše má být uloženo. Stejně tak lze uložit naučený model po skončení procesu učení pomocí připravené funkce **tf.simple_save()**. **Simple_save** uloží všechny potřebné proměnné modelu na předem specifikovanou cestu.

```
cp_callback = tf.keras.callbacks.ModelCheckpoint(checkpoint_path,
    save_weights_only=True, verbose=1)
model.fit((item for item in image_data), epochs=5,
   steps_per_epoch=steps_per_epoch, callbacks = [cp_callback])
```

Ukázka zdrojového kódu 3.21: Vytvoření ModelCheckpoint pro ukládání průběžných výsledků a spuštění trénování modelu neuronové sítě

Po skončení učení a uložení modelu jej lze znovu načíst z paměti a provést predikci nad zvoleným připraveným obrázkem. Nejdříve je potřeba načíst uložený model z paměti, převést obrázek na číselnou hodnotu a následně použít metodu **predict(image_data)**. Tato metoda vrátí třídu klasifikace a jistotu modelu. Vrácená třída klasifikace však není v podobě textového řetězce, ale identifikátoru výstupního neuronu. Tento identifikátor je tedy následně potřeba převést na textovou podobu.

```
def predict(image):
  result_batch = model.predict(image)
  labels_batch = label_names[np.argmax(result_batch, axis=-1)]
  return labels_batch[0]
```

Ukázka zdrojového kódu 3.22: Predikce a dekódování výsledků neuronové sítě

3.7.2 Měření přesnosti modelů a volba extraktoru příznaků

Klasifikace obrázků je komplexní problém, k jehož řešení lze přistupovat různými způsoby. TensorFlow nabízí několik architektur extraktorů příznaků, které se liší v přesnosti klasifikace v rychlosti a nárocích na zařízení. V systému Chobot byly využity dva modely pro klasifikaci obrázků:

- MobileNet – MobileNet je množina modelů připravených pro použití primárně na mobilních telefonech, ale díky jejich kvalitním výsledkům dochází k nasazení těchto modelů i na jiných zařízeních a pro jiné účely. Hlavními kritérii pro volbu modelů z této kategorie jsou rychlost a nízké nároky na zařízení. Modely MobileNet mohou být použity pro detekci objektů, klasifikaci obrázků, segmentace obrazu a další. Při výběru, jaký model z této kategorie zvolit je potřeba předem znát rozlišení obrázků, které budou klasifikovány a počet parametrů použitých v modelu. MobileNet modely jsou na výběr ve dvou verzích. Hlavní rozdíl mezi první a druhou verzí je výrazné snížení počtu parametrů a zvýšení přesnosti modelu. Tabulka 3.7.2 zobrazuje rozdíly v přesnosti jednotlivých modelu při využití ILSVRC datové sady [13].

Model	Top-1 přesnost [14]	Počet parametrů	MACs [15]
MobileNet v1	70,6 %	4,2 M	575 M
MobileNet v2	74,7 %	3,4 M	300 M

- ResNet – Druhou velmi často využívanou skupinou modelů jsou modely ResNet. Tyto modely se specifikují vysokou přesností, kdy při učení na obrázcích z datové sady ILSVRC prokazovaly top-5 chybovost pouhých 3,57 %, což je lepší výsledek než při klasifikaci člověkem. Top-1 přesnost tohoto modelu je 80,62 %. Neuronové sítě ResNet pracují na principu přeskakování vrstev a tím se síť místo učení nových funkcí doučuje zbytkové funkce.

3.7.3 Implementace předtrénovaného klasifikátoru

Pro klasifikaci obrázků s již natrénovanou neuronovou sítí lze využít, stejně jako při dotrénování, připravené modely knihoven uvnitř TensorFlow Hub. Ten nabízí základní připravené modely s tisícem klasifikačních tříd, které lze

[13] ImageNet - Large Scale Visual Recognition Challenge

[14] Přesnost hlavní klasifikační třídy vrácené modelem. Například pokud je modelu předán obrázek kočky a on navrátí následující klasifikace 1 - Tygr 0,4%, 2 - Kočka 0,3%, 3 - Pes 0,1%, 4 - Myš 0,1%, 5 - Morče 0,1%, tak top-1 přesnost je chybová, protože modelem klasifikovaná hlavní třída (Tygr 0,4%) je mylná. Naopak top-5 přesnost je správná, jelikož v prvních pěti třídách klasifikace se kočka nachází.

[15] Počet fúzovaných operací násobení a sčítání

stáhnout již připravené pro nasazení v rámci systému Chobot. Postup vytvoření stažení modelu je velmi podobný, jako při dotrénování, kdy je definován základní model s již nastavenými váhami.

Druhou možností vytvoření klasifikátoru je využití knihovny Keras, která nabízí rozhraní pro stažení a nasazení již předtrénovaných modelů[31]. Programátor definuje architekturu modelu podle jména a následně zvolí jaké váhy se mají do modelu importovat. Váhy se odvíjejí od datové sady, na které byl model trénován a podle ní bude model klasifikovat obrázky.

```
from keras.applications import ResNet50
import keras.preprocessing.image
from keras.applications import imagenet_utils


def load_and_predict(image_data):
  model = ResNet50(weights="imagenet")

  with graph.as_default():
    predictions = model.predict(image)
    results = imagenet_utils.decode_predictions(predictions)
```

Ukázka zdrojového kódu 3.23: Vytvoření neuronové sítě z předučeného modelu ResNet50

V ukázce zdrojového kódu 3.23 dojde nejdříve k definici modelu. Knihovna Keras nabízí celou řadu naučených modelů, které mají stejné rozhraní, ale jinou architekturu. Programátor tak pouze vybere, jakou architekturu použije na základě problému, který potřebuje řešit. Na výběr jsou následující řešení:

- VGG16
- VGG19
- ResNet50
- InceptionV3
- InceptionResNetV2
- Xception
- MobileNet
- MobileNetV2
- DenseNet
- NASNetMobile
- NASNetLarge

Během vytváření modelu lze také zvolit, jaké připravené váhy modelu mají být načteny. Programátor má možnost vybrat si mezi ImageNet, nebo například načíst vlastní váhy z paměti počítače. Poté je jeho model již připraven na klasifikaci pomocí metody **predict(dataObrazku)**.

Pro systém Chobot byl vybrán model NASNetLarge, společně s připravenými váhami ImageNet. Tento model dokáže klasifikovat vstupní obrázky do 1 000 klasifikačních tříd s top-1 přesností 82,5 % a top-5 přesností 96 %. Tento model se skládá z 88 milionů parametrů a jeho velikost se pohybuje okolo 350 MiB. Není tedy vhodný pro klasifikaci na mobilních zařízeních. Autorem tohoto modelu je společnost Google. Zajímavostí tohoto modelu je fakt, že nebyl vytvořen člověkem. Společnost Google vytvořila model, který naučila architekturu a učící se strukturu konvolučních neuronových sítí. Tento model následně navrhl architekturu neuronové sítě NasNet. Jméno modelu vychází z Neural Architecture Search Network, které patří do skupiny AutoML modelů popsaných v příloze Automatické strojové učení.

3.8 Šablona pro uživatelské moduly

Všechny uživatelské moduly jsou nasazeny jako samostatné aplikace v rámci Kubernetes klastru. Komunikují s klasifikátory pomocí REST rozhraní. Stejně jako klasifikátory mají moduly kontrolery pro:

- Zjištění stavu modulu.
- Přijetí požadavku od klasifikátoru.
- Dokumentaci.

Každý uživatelský modul se skládá z kontrolerů, které jsou definovány jako šablona systému a uživatelského kódu. Uživatel tedy při vytváření modulu nemusí brát v potaz infrastrukturu systému a může se soustředit pouze na vlastní implementaci chování modulu. Jediným požadavkem, který musí zdrojový kód uživatelského modulu splnit, je dodržení rozhraní. Rozhraní, které musí uživatelský kód splňovat je následující:

```
def handle(context_data, request_data):
  ... user code ...
```

Ukázka zdrojového kódu 3.24: Rozhraní, uživatelského kódu modulu

Proměnná **context_data** udržuje informace o volání jako je čas, případně identifikátor volajícího a třídu klasifikace vrácenou neuronovou sítí společně s její přesností. Proměnná **request_data** obsahuje originální požadavek, který přijala neuronová síť. Uživatel v rámci kódu může využívat informace obsažené v těchto proměnných, volat externí API, nebo provádět další operace a návratová hodnota vrácena tímto kódem je poté přidána k datům klasifikátoru a předána volajícímu uživateli či aplikaci.

3.9 Závěr

V této kapitole je popsán proces vývoje jednotlivých komponent systému Chobot. Během implementace vznikl systém, který umožňuje uživateli automatizovaně nasadit klasifikační model pro ohodnocení log záznamů, klasifikaci obrázků nebo značkování vstupního textu pro účely informačních chatbotů. Dále vznikla aplikace pro vytváření modulů z uživatelem zadaného zdrojového kódu a jejich následné propojení s klasifikátory. Celý systém lze ovládat skrze vyvinutou klientskou aplikaci, která umožňuje základní testování a monitoring nově vytvořených klasifikátorů a uživatelských modulů.

Kapitola 4

Testování systému

Tato kapitola se zabývá testováním systému Chobot a jeho jednotlivých komponent. Každá komponenta byla podrobena manuálnímu i automatickému testování. Jednotlivé typy provedených testů a jejich strategie jsou popsány v následujících podkapitolách a celkové výsledky testování jsou v sekci Závěr.

4.1 Automatické testování

4.1.1 Automatické testování kontejnerového API

Během vývoje kontejnerového API vzniklo několik testů, které mají za úkol kontrolovat správnost funkcionality komponent této aplikace. Funkcionalita kontejnerového API se skládá z několika vrstev, které spolu navzájem komunikují. Kontejnerové API je Spring Boot aplikace a framework Spring nabízí velké množství předpřipravených knihoven pro usnadnění vývoje a testování aplikací. Jednou z těchto knihoven je spring-boot-test startovací balíček, který podporuje vytváření a konfiguraci testů pomocí anotací. Programátor pouze přidá anotaci **@SpringBootTest** při deklaraci třídy a Spring tuto třídu bude považovat za test a spouštět během testování aplikace. Dále lze díky anotacím určit s jakým kontextem má být test spuštěn. Lze určit zdali má při testu nabíhat kontext celé aplikace, nebo jen určité specifické závislosti. Během vývoje byly vytvořeny jak jednotkové testy, tak testy integrační kontrolující spojení mezi jednotlivými komponentami systému. Ukázka 4.1 ukazuje test připojení ke Kubernetes klastru a získání jmenného prostoru.

```
@RunWith(SpringRunner::class)
@SpringBootTest
class NamespaceConnectionTest {

  @Autowired
  private lateinit var namespaceService: INamespaceService

  private lateinit var api: CoreV1Api
```

```
  @Before
  fun getKubernetesApi() {
    val client = Config.defaultClient()
    Configuration.setDefaultApiClient(client)
    api = CoreV1Api()
  }

  @Test
  fun testDefaultNamespace() {
    val namespace = namespaceService.getOrCreateNamespace(api,
        "default")
    assert(namespace.apiVersion == "v1")
    assert(namespace.metadata.name == "default")
  }
}
```

Ukázka zdrojového kódu 4.1: Test servisní třídy pro získání jmenného prostoru

Kontejnerové API je při testování připojeno k testovací množině dat a je kontrolována následující skupina operací

- Vytvoření nové instance klasifikátoru.
- Nasazení instance klasifikátoru.
- Vytvoření nového uživatelského modulu (integrační test).
- Nasazení vytvořeného uživatelského modulu.
- Zrušení nasazení uživatelského modulu.
- Zrušení nasazení instance klasifikátoru.
- Editace údajů o uživateli.

4.1.2 Automatické testování obrazového API

Během vytváření nového obrazu modulu komunikuje obrazové API s několika externími aplikacemi. Proto je primárně potřeba otestovat komunikaci mezi těmito aplikacemi. Mezi integračními testy je i test komunikace a zpracování odpovědi mezi obrazovým API a Git serverem, ze kterého jsou stahovány šablony klientských modulů. Protože volání a vytváření nových repozitářů během testů by zbytečně zatěžovalo Git server, je k těmto testům využita knihovna Mockito. Ta vytváří zástupný objekt Git serveru a vrací předem předdefinovanou odpověď.

```
@SpringBootTest
@RunWith(MockitoJUnitRunner::class)
@TestPropertySource(properties = arrayOf(
  "image.git.repo.uri=some_url",
  "gitlab.url=some"))
```

```
class GitConnectionTest {
  @Mock
  private val restTemplate: RestTemplate? = null

  @InjectMocks
  private val gitService = GitServiceImpl()

  @Value("\${gitlab.url}")
  private val gitServerUrl: String? = null

  private lateinit var entity: HttpEntity<String>

  @Before
  fun createHeadersEntity() {
    val headers = HttpHeaders()
    headers.add("PRIVATE-TOKEN", "some-token")
    entity = HttpEntity<String>(headers)
  }

  @Test
  fun testUserModuleTemplateCheckout() {
    Mockito.`when`(restTemplate?.postForEntity("$gitServerUrl/
        projects?name=$username-$projectName", entity, String::
        class.java))
      .thenReturn(ResponseEntity("", HttpStatus.CREATED))

    val newRepoUrl = gitService.createGitlabProject(username,
        projectName)
    assert(newRepoUrl != null)
  }
}
```

Ukázka zdrojového kódu 4.2: Ukázka testu využívajícího zástupné objekty knihovny Mockito

4.1.3 Automatické testování společné funkcionality klasifikátorů a uživatelských modulů

Jednotlivé instance klasifikátorů sdílejí společnou funkcionalitu, která je testována pomocí knihoven Pytest a Betamax. Knihovna Betamax umožňuje nahrát žádost směrovanou klasifikátorem na existující uživatelský modul a při příštím spuštění testu poskytovat toto nahrané volání společně s jeho odpovědí bez nutnosti toho, aby daný uživatelský modul skutečně existoval. Dále testy klasifikátorů zahrnují i testování autorizace komunikace mezi uživatelem a REST API klasifikátoru a také testy autorizované komunikace mezi klasifikátorem a uživatelskými moduly. Testován byl jak pozitivní, tak i negativní průchod.

4.2 Manuální testování

Pro potřeby manuálního testování byla vytvořena množina testovacích scénářů, která byla otestována několika uživateli. Všichni testeři byli nejdříve seznámeni se základní myšlenkou systému a proběhlo krátké školení, kde byly demonstrovány jednotlivé funkcionality systému. Všichni testeři nejdříve provedli volné testování bez scénáře a následně procházeli jednotlivé kroky testovacích scénářů a kontrolovali odpovědi systému. Jednotlivé testy byly rozděleny do skupin a proběhlo jak testování pozitivních, tak i negativních scénářů.

4.2.1 Testovací scénáře

Během vývoje systému vznikla celá řada manuálních testovacích scénářů. Tyto testovací scénáře jsou rozděleny do dvou skupin. Na následující ukázce je jeden z pozitivních testovacích scénářů na kontrolu klientské aplikace a integrace všech dalších komponent potřebných pro vytvoření nového klasifikátoru obrázků.

Krok: Uživatel přejde na adresu klientské aplikace systému Chobot.
Očekávaná odpověď: Zobrazí se mu přihlašovací stránka.
Krok: Uživatel vyplní přihlašovací jméno a heslo a klikne na tlačítko ”login”.
Očekávaná odpověď: Aplikace jej přesměruje na hlavní stránku aplikace, kde jsou zobrazeny všechny uživatelem vytvořené klasifikátory.
Krok: Uživatel klikne na tlačítko ”+”.
Očekávaná odpověď: Aplikace jej přesměruje na stránku pro vytvoření nového klasifikátoru.
Krok: Uživatel klikne na tlačítko ”save”.
Očekávaná odpověď: Aplikace zobrazí chybovou hlášku o špatně vyplněném jménu klasifikátoru.
Krok: Uživatel zadá do pole jména klasifikátoru pomlčku a klikne na tlačítko ”save”.
Očekávaná odpověď: Aplikace zobrazí chybovou hlášku o špatně vyplněném jménu klasifikátoru.
Krok: Uživatel vyplní jméno nového klasifikátoru.
Očekávaná odpověď: Jméno vyplněno.
Krok: Uživatel klikne na tlačítko ”save”.
Očekávaná odpověď: Aplikace zobrazí chybovou hlášku o špatně vybraném typu klasifikátoru.
Krok: Uživatel klikne na pole pro výběr typu klasifikátoru.
Očekávaná odpověď: Zobrazí se pole s výběrem typu klasifikátoru.
Krok: Uživatel vybere typ ”Custom image classification”.
Očekávaná odpověď: Typ klasifikátoru vybrán a na formuláři se zobrazí nové pole pro zadání trénovacích dat.
Krok: Uživatel klikne na tlačítko ”save”.
Očekávaná odpověď: Aplikace zobrazí chybovou hlášku o chybějících trénovacích datech.
Krok: Uživatel klikne na tlačítko ”Import file”.

Očekávaná odpověď: Zobrazí se vyskakovací okno pro vybrání trénovacích dat z lokálního disku uživatele.
Krok: Uživatel vybere trénovací data a potvrdí svůj výběr.
Očekávaná odpověď: V poli "training data file"je cesta k vybraným datům uživatele.
Krok: Uživatel klikne na tlačítko "save".
Očekávaná odpověď: Aplikace jej přesměruje na stránku s detaily o nově vytvořeném klasifikátoru a na spodní části obrazovky se zobrazí informační hláška o úspěšném vytvoření nového klasifikátoru.
Krok: Data zobrazená na stránce s detaily se shodují s daty zadanými uživatelem.
Očekávaná odpověď: Data se shodují.

4.2.2 Integrační testování

V rámci manuálního testování proběhlo také integrační testování, kdy byly kontrolovány jednotlivé zprávy, které si komponenty systému zasílají. Během testování bylo kontrolováno chování systémů v případě chybového stavu, nebo úplného výpadku jedné komponenty. V rámci integračního testování proběhlo také testování obnovy systému a jeho jednotlivých komponent. Během testů byly postupně vypínány některé komponenty a testovalo se jejich znovu naběhnutí a funkcionalita celého systému. Během testu se potvrdil předpoklad, že po vypnutí jedné z komponent dojde k výraznému omezení funkcionality celého systému. Nejméně kritickou komponentou pro celý systém je klientská aplikace. Při její nedostupnosti je systém nadále použitelný. Pro uživatelskou interakci je však tato komponenta klíčová. Naopak při výpadku celého Kubernetes klastru je celý systém zablokován na své funkcionalitě. Během testu došlo k ověření faktu, že ostatní komponenty fungují v pořádku po znovu naběhnutí jiných komponent a nepotřebují restart.

4.3 Statická analýza kódu

Pro statickou analýzu kódu byl využit nástroj **SpotBugs**, který prochází sestavený zdrojový kód a podle předem nastavených pravidel hledá programátorské chyby a nepřesnosti. Výsledek statické analýzy zdrojového kódu je přehledný HTML dokument, kde jsou označeny řádky souborů, kde se potenciální chyba nachází a její popis. Programátor má následně možnost v rámci konfigurace určit, které balíčky mají být kontrolovány, a které mají být přeskočeny.

4.4 Závěr

Jak během automatického testování tak i během manuálního testování bylo nalezeno množství chyb a nedostatků. Většina chyb byla následně opravena

a proběhlo opakované testování, které již dopadlo úspěšně. Během manuálního testování, byla nalezena chyba, kdy po delším nepoužívání klasifikátoru dojde k nesprávnému odpojení relace připojení do databáze. To způsobí, že první připojení k databázi po nečinnosti skončí chybovým stavem. Všechna další připojení již opět proběhnou korektně. Tuto chybu zřejmě způsobuje databázový ovladač knihovny SQLAlchemy. Pro konzultaci k vyřešení tohoto problému byl kontaktován autor této knihovny.

Kapitola 5

Nasazení systému

Jednotlivé komponenty systému mají připravený Dockerfile obsahující proces nasazení a jednotlivé kroky potřebné pro správnou komunikaci uvnitř systému. Nejdříve dojde ke stažení bázového obrazu, který slouží jako základní prostředí pro běh dané komponenty. Dalším krokem je instalace potřebných knihoven jako například nástroje Maven pro sestavení kontejnerového API, nebo instalace knihoven pro Python aplikace. Poté jsou jednotlivým komponentám systému vystaveny porty, na kterých čekají na požadavky od ostatních komponent. Posledním krokem uvedeným v Dockerfile je samotné spuštění kontejnerizované aplikace[32].

```
FROM java:8-jdk
WORKDIR /usr/bin
VOLUME /root/.m2
run curl -sLO http...apache-maven-3.3.9-bin.zip && unzip apache-
    maven-3.3.9-bin.zip
ENV M2_HOME /usr/bin/mvn
ENV PATH $PATH:$M2_HOME/bin

WORKDIR /container_api

COPY container_api.jar /container_api.jar
ENTRYPOINT ["java","-jar","-Dspring.profiles.active=docker-devel"
    ,"/container_api.jar"]
```

Ukázka zdrojového kódu 5.1: Ukázka Dockerfile pro kontejnerové API

Po vytvoření konfiguračních Dockerfile souborů jsou jednotlivé komponenty spojeny v jeden Docker-compose soubor, který propojuje komunikaci jednotlivých komponent. Stejně jako pro kontejnerové API, tak i pro ostatní komponenty systému jsou vytvořeny Dockerfile konfigurace.

```
version: '2'
services:
  chobot-container-api:
    build: ./container_api
    container_name: chobot-container-api
```

```
    ports:
      - 8090:8090
      - 8000:8000
    volumes:
      - '../container_api:/container_api'
    environment:
      - "SPRING_PROFILES_ACTIVE=docker-devel"
    links:
      - chobot-db
    depends_on:
      - chobot-db

  chobot-db:
    build: ./db
    container_name: chobot-db
    ports:
      - 3306:3306
```

Ukázka zdrojového kódu 5.2: Ukázka konfigurace Docker-compose souboru

Kapitola 6

Budoucnost systému a další rozvoj

Veškeré plánované funkcionality systému Chobot byly implementovány a otestovány. Nicméně je již naplánovaný další rozvoj systému pro jeho širší využití. Prvotně, je potřeba přidat více možností implementace uživatelských modulů. Jelikož je komunikace mezi klasifikátory a moduly prováděna pomocí volání REST API modulu, lze přidat podporu dalších programovacích jazyků bez nutnosti zasahovat do existujících klasifikátorů nebo obrazového API.

Pro rozšíření využitelnosti aplikace a uživatelské přívětivosti je plánován rozvoj klientské aplikace. Plánovaným rozvojem je přidání možnosti, aby uživatel měl více verzí jednoho modulu a během nasazení vybral, která verze má být nasazena. Obrazové API, kontejnerové API i datová vrstva jsou připraveny a tato funkcionalita je již implementována. V klientské aplikaci však zatím chybí. Dále je plánováno uživateli umožnit připojit všechny instance modulů a klasifikátorů k aplikaci Elasticsearch a v rámci klientské aplikace zobrazit rozhraní Kibana. Tím by měl uživatel větší přehled o běhu modulů a v případě problémů by mohl lépe reagovat.

Dalším plánovaným rozšířením je větší integrace AutoML klasifikátorů popsaných v příloze Automatické strojové učení, pro větší personalizaci jejich chování na základě uživatelských dat. Tento proces ale vyžaduje zevrubné testování jednotlivých funkcionalit AutoML knihoven, jelikož zatím nejsou ve stabilní verzi.

Závěr

V rámci této práce proběhl výzkum systému pro automatizované nasazování a podporu životního cyklu modelů strojového učení a k nim navázaných modulů. Během výzkumu vznikla série funkčních a nefunkčních požadavků na systém jako celek, tak i na jeho jednotlivé komponenty. Dále proběhl výzkum jehož výsledkem byly vhodné architektury modelů neuronových sítí pro klasifikaci obrázků, log záznamů a vstupního textu pro účely chatbotů. Pro celý systém byl vytvořen návrh architektury a jednotlivé komponenty systému byly vyvinuty podle provedené analýzy. Vznikl tak systém, který dokáže automatizovaně nasadit připravené modely klasifikátorů, předat jim vstupní data od uživatele a výsledný naučený model vystavit tak, aby jej mohl uživatel posléze používat. K vystaveným klasifikátorům lze také navázat uživatelské moduly, které rozšiřují jejich funkcionalitu.

V teoretické části této práce jsou popsány principy neuronových sítí, jejich učení a architektura. Znalosti získané během tvorby teoretické práce, jsou využity v kapitole Analýza a návrh, kde je přehled a diskuze zvolených modelů pro klasifikaci. V této kapitole jsou také popsány požadavky na systém, jeho architektura a zabezpečení. Během definice architektury se autor snažil co nejvíce rozložit systém na funkční komponenty, které půjde jednoduše škálovat a budou mít jednoduchou a samostatnou administraci. Dále jsou v této kapitole popsány procesy systému, případy užití a zvolené technologie pro automatizované nasazení.

Praktická část této práce je rozdělena na tři části, **Realizace**, **Testování systému** a **Nasazení**. Kapitola Realizace popisuje jednotlivé komponenty a proces jejich vývoje. Dále jsou v této kapitole popsány modely klasifikátorů a jejich měření. Další část této práce se zabývá testováním vzniklého systému Chobot. Testování proběhlo jak manuálním, tak automatickým způsobem. Kapitola Nasazení popisuje jak systém nasadit do produkčního nebo testovacího prostředí.

V poslední části se autor zabývá budoucím rozvojem systému, zlepšením uživatelského zážitku a metodám větší personalizace klasifikačních modelů

na základě uživatelem předaných dat. Personalizace architektury a parametrů klasifikačních modelů a jejich nasazení v systému Chobot popisuje příloha **Automatické strojové učení**.

Během analýzy a vývoje tohoto systému se autor práce seznámil s velkým množstvím nových technologií. Za největší přínos autor považuje získání přehledu o modelech strojového učení, jejichž přínos autor považuje za důležitý z pohledu budoucího směrování odvětví informačních technologií. Dále se autor seznámil s administrací a obsluhou Kubernetes klastru, který se projevil jako klíčová komponenta při vývoji systému Chobot.

Literatura

[1] Purkait, N.: *Hands-on neural networks with keras : design and create neural networks using deep learning and... artificial intelligence principles.* S.l: Packt publishing limited, 2019, ISBN 9781789536089.

[2] Shanmugamani, R.: *Deep learning for computer vision : expert techniques to train advanced neural networks using TensorFlow and Keras.* Birmingham, UK: Packt Publishing, 2018, ISBN 1788295625.

[3] Loss Functions in Neural Networks. [online], [cit 14-03-2019]. Dostupné z: `https://isaacchanghau.github.io/post/loss_functions/`

[4] Zaccone, G.: *Deep Learning with TensorFlow : Explore neural networks and build intelligent systems with Python, 2nd Edition.* Birmingham: Packt Publishing, 2018, ISBN 9781788831109.

[5] Loy, J.: *Neural network projects with python : the ultimate guide to using python to explore the true power... of neural networks through six projects.* S.l: Packt publishing limited, 2019, ISBN 9781789138900.

[6] Kostadinov, S.: *Recurrent Neural Networks with Python Quick Start Guide Sequential Learning and Language Modeling with TensorFlow.* Birmingham: Packt Publishing Ltd, 2018, ISBN 9781789132335.

[7] Zafar, I.: *Hands-on convolutional neural networks with TensorFlow : solve computer vision problems with modeling in TensorFlow and Python.* Birmingham, UK: Packt Publishing, 2018, ISBN 9781789130331.

[8] Karim, M. R.: *Practical Convolutional Neural Networks : Implement advanced deep learning models using Python.* Birmingham: Packt Publishing, 2018, ISBN 9781788392303.

[9] Li, W.: *Automatic Log Analysis using Machine Learning : Awesome Automatic Log Analysis version 2.0.* Diplomová práce, Uppsala University, Department of Information Technology, 2013.

[10] Zaccone, G.: *Getting started with tensorflow: get up and running with the latest numerical computing library by Google and dive deeper into your data*. Birmingham: Packt Publishing, 2016, ISBN 1-78646-906-5.

[11] Gulli, A.: *Deep learning with Keras : implement neural networks with Keras on Theano and TensorFlow*. Birmingham, UK: Packt Publishing, 2017, ISBN 1787128423.

[12] IaaS, PaaS and SaaS – IBM Cloud service models. [online], [cit 9-03-2019]. Dostupné z: `https://www.ibm.com/cz-cs/cloud/learn/iaas-paas-saas`

[13] Kubernetes Concepts. [online], [cit 11-03-2019]. Dostupné z: `https://kubernetes.io/docs/concepts/`

[14] Wu, C.: *Devops with kubernetes. Accelerating software delivery with container orchestrators, 2nd edition*. S.l: Packt publishing limited, 2019, ISBN 9781789533996.

[15] Ambassador Configuration. [online], [cit 3-03-2019]. Dostupné z: `https://www.getambassador.io/reference/configuration`

[16] Ambasador Architecture. [online], [cit 1-03-2019]. Dostupné z: `https://www.getambassador.io/concepts/architecture`

[17] Bryant, D.: Deploying Java Apps with Kubernetes and the Ambassador API Gateway. [online], [cit 10-03-2019]. Dostupné z: `https://blog.getambassador.io/deploying-java-apps-with-kubernetes-and-the-ambassador-api-gateway-c6e9d9618f1b`

[18] Yilmaz, O.: *Kubernetes design patterns and extensions : enhance your container-cluster management skills and efficiently develop and deploy applications*. Birmingham, UK: Packt Publishing Ltd, 2018, ISBN 9781789619270.

[19] Spring Boot Reference Guide. [online], [cit 25-03-2019]. Dostupné z: `https://docs.spring.io/spring-boot/docs/current-SNAPSHOT/reference/htmlsingle/`

[20] Antonov, A.: *Spring Boot cookbook : over 35 recipes to help you build, test, and run Spring applications using Spring Boot*. Birmingham, UK: Packt Publishing, 2015, ISBN 9781785284151.

[21] Khan, A.: *Hands-on object-oriented programming with Kotlin : build robust software with reusable code using OOP principles and design patterns in Kotlin*. Birmingham, UK: Packt Publishing, 2018, ISBN 9781789617726.

[22] Routing & Navigation. [online], [cit 25-03-2019]. Dostupné z: https://angular.io/docs/ts/latest/guide/router.html

[23] NgModule. [online], [cit 26-03-2019]. Dostupné z: https://angular.io/docs/ts/latest/guide/ngmodule.html

[24] Coury, F.; Lerner, A.; Murray, N.; aj.: *ng-book 2*. San Francisco: Fullstack.io, první vydání, 2016, ISBN 978-0991344611.

[25] Gaspar, D.: *Mastering Flask web development : build enterprise-grade, scalable Python web applications*. Birmingham, UK: Packt Publishing, 2018, ISBN 9781788995405.

[26] Gulli, A.: *TensorFlow 1.x deep learning cookbook : over 90 unique recipes to solve artificial-intelligence driven problems with Python*. Birmingham, UK: Packt Publishing, 2017, ISBN 978-1-78829-359-4.

[27] Hardeniya, N.: *Natural language processing : Python and NLTK : learning path : learn to build expert NLP and machine learning projects using NLTK and other Python libraries*. Birmingham, UK: Packt Publishing, 2016, ISBN 9781787285101.

[28] Bird, S.: *Natural language processing with Python*. Beijing Sebastopol, CA: O'Reilly Media Inc, 2009, ISBN 0596516495.

[29] Manaswi, N.: *Deep learning with applications using Python : chatbots and face, object, and speech recognition with TensorFlow and Keras*. Berkeley, CA: Apress, 2018, ISBN 978-1-4842-3515-7.

[30] How to Retrain an Image Classifier for New Categories. [online], [cit 22-03-2019]. Dostupné z: https://www.tensorflow.org/hub/tutorials/image_retraining

[31] Image Preprocessing - Keras. [online], [cit 22-02-2019]. Dostupné z: https://keras.io/preprocessing/image/

[32] Docker overview. [online], [cit 22-03-2019]. Dostupné z: https://docs.docker.com/engine/docker-overview/

[33] What is AutoML? [online], [cit 28-03-2019]. Dostupné z: https://www.automl.org/automl/

[34] Getting Started - Autokeras. [online], [cit 01-03-2019]. Dostupné z: https://autokeras.com/start/

[35] Google AI creates novel neural network NASNet. [online], [cit 16-04-2019]. Dostupné z: https://www.cbronline.com/news/google-ai-creates-novel-neural-network-nasnet

[36] Auto-Keras: An Efficient Neural Architecture Search System. [online], [cit 16-04-2019]. Dostupné z: https://arxiv.org/pdf/1806.10282.pdf

Příloha A

Automatické strojové učení

Automatické strojové učení[16] je podmnožinou strojového učení, kdy jsou modely a jejich parametry vybírány automaticky na základě vstupních dat[33]. Uživatel tak nemusí mít znalosti strojového učení, aby vytvořil vhodný model, který bude sloužit ke klasifikaci, regresi, analýze dat nebo k řešení jiného problému. AutoML nabízí mimo jiné i techniky předzpracování dat, tak aby uživatel, který například nezná NLP procesy nemusel převádět zdrojové texty do podoby číselných vektorů. Kompletní seznam procesů, na které se dají aplikovat funkcionality AutoML je následující:

- Předzpracování dat.
- Výběr vhodných příznaků z trénovací množiny dat.
- Výběr vhodné skupiny modelů.
- Optimalizace parametrů modelů.
- Analýza výsledků modelů.

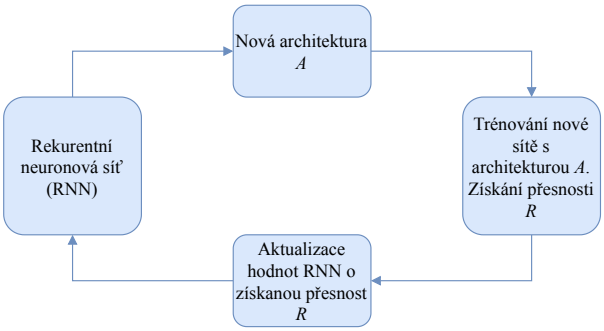
Díky tomu jsou AutoML projekty schopné vytvářet vhodný model, který je přizpůsoben vstupním datům uživatele a tím nabídnout přesnější výsledky než modely, jejichž architektura je předem definovaná na základě typu problému, ale ne pro specifická vstupní data.

Ačkoliv je AutoML stále začínající odvětví, existuje již několik pokročilých projektů, jejichž výsledky jsou i nasazené v produkčním prostředí. Zřejmě nejznámější a nejpokročilejší projekt nabízí společnost Google, jehož algoritmus AdaNet umožňuje automatizované vyhledávání vhodné architektury neuronové sítě na základě vstupních dat. Tento projekt vytvořil například konvoluční neuronovou síť ResNet popsanou v kapitole Realizace. Dalšími známými projekty jsou auto-sklearn, H2O, nebo Autokeras.

[16] Auto Machine Learning – AutoML

A.1 Hledání vhodné architektury sítě – NAS

Učení AutoML modelu se skládá z několika kroků. Nejdříve jsou vstupní data analyzována a částečně předzpracována tak, že jsou převedena do číselných vektorů. Dále je z celé množiny vstupních dat vytvořeno několik menších podmnožin, které slouží k vyhledání a testování vhodných kandidátů modelu. Proces hledání vhodné architektury modelu se nazývá NAS – Neural Architecture Search. NAS je rekurentní neuronová síť, která je naučena na stovkách až tisících architektur neuronových sítí tak, že sama dokáže navrhovat a testovat nové architektury na základě trénovacích dat. NAS tedy nejdříve vytvoří několik[17] základních modelů architektury, které jsou učeny na podmnožinách trénovacích dat. Během učení těchto modelů je měřena jejich přesnost a ty modely, které vykazují velkou nepřesnost jsou ukončeny. Naopak modely, které vykazují určitou přesnost jsou upraveny tak, aby se jejich přesnost ještě více zvýšila. Tento proces se iterativně opakuje dokud není nalezen model s požadovanou přesností.



Obrázek A.1: Rekurentní neuronová síť (RNN) vytvoří architekturu sítě A. Nová síť s architekturou A je učena na podmnožině trénovacích dat s přesností R. Výsledná přesnost R je předána RNN, která následně vytvoří novou architekturu A'. Ta by měla dosáhnout větší přesnosti klasifikace R'

Rekurentní síť určena pro vytváření nových architektur využívá posilovaného zpětnovazebního učení (reinforcement learning), kdy je přesnost nových trénovaných architektur předávána kontrolující síti jako impuls. Samotný proces hledání nových architektur může být implementován několika způsoby. Prvním, nejpoužívanějším způsobem je tzv. **Banditovo trénování**. Banditovo trénování na začátku procesu vytvoří stovky až tisíce možných architektur a na všech těchto architekturách spustí trénování. Následně periodicky po určitém časovém intervalu prochází všechny trénované architektury, změří jejich přesnost a polovinu architektur s nejnižší přesností ukončí. Toto opakuje, dokud nezbude poslední nejvýkonnější architektura.

[17] stovky až tisíce

Dalším způsobem hledání vhodné architektury jsou evoluční algoritmy. Jednotlivé výsledky architektur jsou měřeny a následně jsou tyto architektury spojovány. Tímto způsobem dochází k vytvoření nové generace, která spojuje výsledky předchozích a tím přináší lepší výsledky. Tento proces je velmi využíván při generování architektur konvolučních neuronových sítí.

Vstupní trénovací data pro hledání vhodné architektury jsou pro účely NAS zmenšena, pokud by byl každý testovaný model učen na celé množině trénovacích dat, trval by proces nalezení modelu velice dlouhou dobu. Tato redukce dat s sebou přináší riziko předčasného ukončení vývoje modelu, který ze začátku vykazuje nepřesné výsledky, ale po delší době učení a modifikaci by jeho přesnost vzrostla natolik, že překoná konečný výsledný model. I přes toto zrychlení při hledání modelu je algoritmus hledání architektury sítě NAS velmi zdlouhavý.

A.1.1 ENAS

Algoritmus efektivního hledání architektury neuronové sítě ENAS[18] využívá předpokladu, že jednotlivé modely mohou mezi sebou sdílet nastavení vah a využívat výhod přenosového učení. Modely, které vycházejí z předchozích generací přebírají mimo jiné i jejich nastavení vah a tím je proces učení nových generací velmi zrychlen.

A.2 Autokeras

Autokeras je projekt pro hledání architektury neuronové sítě využívající algoritmů NAS a ENAS[34]. Tento projekt je postaven na knihovně Keras a díky jednoduchému rozhraní nabízí poměrně příjemné uživatelské rozhraní. Proces učení je možné omezit časovým intervalem, během kterého jsou architektury učeny.

```
clf = ImageClassifier(verbose=True, augment=False)
clf.fit(x_train, y_train, time_limit=3 * 60 * 60)

clf.final_fit(x_train, y_train, x_test, y_test, retrain=True)
y = clf.evaluate(x_test, y_test)
print(y * 100)
```

Ukázka zdrojového kódu A.1: Vytvoření nového klasifikátoru pomocí knihovny Autokeras

V systému Chobot se nachází připravené klasifikátory využívající funkcionalitu této knihovny. Pro rozlišení, jakým způsobem byl model vytvořen jsou tyto modely v klientské aplikaci označeny příponou autoML. Během implementace systému Chobot byl vytvořen klasifikátor pomocí algoritmu NAS. Klasifikátor byl během testování trénován na obrázkové datové sadě

[18] Efficient Neural Architecture Search

Cifar10. Tento klasifikátor využívá třídu **ImageClassifier**. Ta přijímá jako jeden z hlavních atributů maximální čas hledání. Po tuto dobu probíhá hledání vhodné architektury neuronové sítě a po jejím skončení uloží nejlepší nalezenou architekturu. Poté je tato architektura pomocí metody **final_fit()** naučena. Během hledání architektury, ani během samotného učení nejlepší architektury není potřeba specifikovat žádný další parametr učení.

Pro testování třídy **ImageClassifier** byla zvolena datová sada Cifar10. Hledání architektury neuronové sítě probíhalo po dobu jedenácti hodin a následně proběhlo učení nejlepšího nalezeného modelu. Po naučení modelu byl měřena přesnost na testovací množině dat. Naměřená přesnost dosahovala 79,8 %. Pro porovnání byl zvolen model konvoluční neuronové sítě NasNet od společnosti Google, jehož architektura byla vytvořena také pomocí technik automatického strojového učení. Model NasNet Large dosahuje na stejné datové sadě přesnosti 82,7 %[35]. Toto porovnání ukazuje, že architektura konvoluční neuronové sítě vytvořena pomocí algoritmu NAS dosahuje velmi dobrých výsledků. Pokud by proces hledání architektury byl spuštěn ještě delší dobu, výsledná neuronová síť by dosahovala ještě lepších výsledků[36].

Autokeras nabízí i celu řadu již předučených modelů, které lze dotrénovat na učících datech, nebo rovnou nasadit k použití. Mezi tyto modely lze zařadit například detekci objektů na fotografii, generování hlasu z textu, nebo analýzu sentimentu ze vstupního textu.

```
from autokeras.pretrained.object_detector import ObjectDetector
detector = ObjectDetector()
detector.load()
results = detector.predict(image_path, output_file_path=
    output_path)
```

Ukázka zdrojového kódu A.2: Ukázka využití předtrénovaného modelu pro detekci objektů na fotografii

A.3 Implementace metod AutoML do systému Chobot

Autokeras nabízí celou řadu připravených modelů a postupů jak implementovat metody automatického strojového učení do existujících aplikací. V rámci systému Chobot bylo využito několik již předtrénovaných modelů, ale také výše zmíněný model vytvořený pomocí algoritmu NAS. Výčet modelů je následující.

- Klasifikace sentimentu z psaného textu. Tento model slouží k určení sentimentu v psaném textu. Model je již předtrénovaný na datové sadě oklasifikovaných recenzí na portálu IMDB. Hlavní funkcionalitu zastřešuje objekt třídy **SentimentAnalysis**, který dědí své rozhraní od nadřazené

třídy **TextClassifier**. Jelikož je model již předtrénovaný, stačí pouze vytvořit nový objekt při jehož inicializaci dojde ke stažení NLP modelu **BERT**[19] sloužícího k předzpracování vstupních dat. Samotná klasifikace je provedena zavoláním metody **predict(text)**.

```
from autokeras.pretrained.text_classifier import
    SentimentAnalysis
sentiment_analysis = SentimentAnalysis()
sentiment_analysis.predict("This model works well.")
```

- Klasifikace tématu psaného textu. Stejně tak jako model pro klasifikaci sentimentu využívá tento model rozhraní třídy **TextClassifier** a modelu **BERT**.

- Detekce a klasifikace objektů na obrázku. Tento předtrénovaný model umožňuje klasifikovat objekty na obrázku. K této funkci navíc přidává možnost označení, na jaké pozici obrázku se daný objekt nachází. Jelikož se jedná o již předtrénovaný model, nelze jeho chování ani naučené klasifikační třídy dále upravovat. O veškerou funkcionalitu klasifikátoru se stará třída **ObjectDetector**. Ta má, stejně jako další modely knihovny Autokeras, připravenou metodu **predict(image)**, která přijme obrázek a vrátí hlavní klasifikační třídu společně s její pozicí na obrázku.

- Klasifikace obrázků pomocí modelu vytvořeného algoritmem NAS. Model využívající algoritmu NAS pro určení architektury neuronové sítě pro klasifikaci obrázků.

- Klasifikace textu pomocí modelu vytvořeného algoritmem ENAS. Tento model opět využívá vyhledávání nejlepší vhodné architektury na základě předaných dat. Navíc obsahuje i metody pro zpracování psaného textu do podoby číselných vektorů. Není tedy potřeba, aby programátor jakkoliv připravoval trénovací data. O vytvoření modelu se stará objekt třídy **TextClassifier**. Ten má stejně jako jiné objekty v rámci knihovny Autokeras unifikované rozhraní. Pro vyhledání nové architektury slouží metoda **fit(trénovací_data, klasifikační_třídy)** a pro vytvoření finálního modelu slouží metoda **final_fit(trénovací_data, klasifikační_třídy)**.

```
from autokeras.text.text_supervised import TextClassifier
clf = TextClassifier(verbose=True)
clf.fit(x=x_train, y=y_train, time_limit=12 * 60 * 60)
clf.final_fit(x=x_train, y=y_train)
```

Všechny tyto modely jsou implementovány jako klasifikátory v rámci systému Chobot a je možné využít jejich funkcionalitu. V rámci budoucího rozvoje je naplánováno větší začlenění funkcí této knihovny do systému Chobot.

[19] https://arxiv.org/abs/1810.04805

A.4 Závěr

AutoML je velice zajímavý přístup jak automatizovat vytváření architektury a samotný trénink neuronových sítí. Ačkoliv je již několik existujících projektů, jedná se stále o nové odvětví. Knihovna Autokeras díky jednoduchému rozhraní a implementaci algoritmu ENAS nabízí velice zdařilou alternativu ke Google AutoML, které není nabízeno se svobodnou licencí a je zpoplatněno. Jelikož je tento projekt stále ve fázi vývoje, obsahuje mnoho chyb. Některé z těchto chyb byly opraveny autorem této diplomové práce v rámci vývoje systému Chobot.

Příloha B

Instalační manuál

B.1 Docker a Kubernetes klastr

Pro správný běh všech komponent celého systému Chobot je nejdříve potřeba nainstalovat nástroj Docker společně s Kubernetes klastrem. Pokud je systém Chobot instalován na operačních systémech Microsoft Windows nebo macOS, lze využít připravenou instalaci Kubernetes klastru v rámci nástroje Docker a pouze povolit klastr v konfiguraci nástroje Docker. Pokud se jedná o jiný operační systém, je potřeba Kubernetes klastr nainstalovat samostatně. Po instalaci klastru je vhodné připravit monitorovací prostředí pro případné zjištění chyb při nasazení klasifikátorů nebo uživatelských modulů. Nejjednodušším monitorovacím nástrojem (kromě samotného nástroje kubectl a výpisu do příkazové řádky) je nástroj Kubernetes Dashboard. Ten přehledným způsobem zobrazuje informace o všech Kubernetes objektech v rámci celého klastru. Pro jeho instalaci je potřeba do příkazové řádky zadat následující příkaz:

```
$ kubectl apply -f https://raw.githubusercontent.com/kubernetes/
    dashboard/master/aio/deploy/recommended/kubernetes-dashboard.
    yaml
```

Pro přihlášení do Kubernetes Dashboard lze využít Bearer token nebo konfigurační soubor kubeconfig. Bearer token lze získat zadáním příkazu:

```
$ kubectl -n kube-system describe secret $(kubectl -n kube-system
     get secret | grep admin-user | awk '{print $1}')
```

Tento příkaz vypíše do konzole dlouhý seznam všech údajů vypadající podobně jako na následující ukázce.

```
Name:         kubernetes-dashboard-token-4q6tn
Namespace:    kube-system
Labels:       <none>
Annotations:  kubernetes.io/service-account.name=kubernetes-
    dashboard
              kubernetes.io/service-account.uid=a4c414f3-61c6-11
                  e9-b94e-025000000001
```

```
Type:  kubernetes.io/service-account-token

Data
====
ca.crt:     1025 bytes
namespace:  11 bytes
token:      <TOKEN>
```

V tomto seznamu je potřeba nalézt záznam, jehož jméno začíná předponou **kubernetes-dashboard-token**. Po získání tokenu se pomocí příkazu příkazové řádky **kubectl proxy** spustí nástroj Kubernetes Dashboard a umožní přihlášení právě pomocí získaného tokenu[20].

Dále je potřeba upravit konfigurační soubor kontejnerového API a nastavit přístupový Bearer token jako hodnotu parametru **kube.token**. Lze využít stejný token, který byl použit pro přihlášení ke Kubernetes Dashboard. Tento token využívá aplikace pro přístup ke klastru. V dalším kroku je potřeba nakonfigurovat **název hostitelského systému** (hostname) Kubernetes klastru do parametrů **host.url** a **ambasador.service.url.internal**. Jelikož kontejnerové API běží uvnitř Docker kontejneru a klastr běží buďto na vzdáleném serveru nebo na hostitelském operačním systému, nelze ke klastru přistupovat skrze *localhost*. Pokud klastr běží na hostitelském operačním systému, a tento operační systém je macOS nebo Microsoft Windows, lze využít předdefinovaný název pro přístup na hostitelský systém **host.docker.internal**. Pokud se jedná o jiný operační systém, lze adresu klastru získat zadáním příkazu do příkazové řádky **ip -a**[21].

B.2 Kubernetes Ambasador

Dalším krokem při instalaci systému je konfigurace nástroje Ambasador. Nejdříve je potřeba vytvořit nasazení tohoto nástroje pomocí příkazu:

```
$ kubectl apply -f https://getambassador.io/yaml/ambassador/
    ambassador-rbac.yaml
```

Posléze je nutné vytvořit nový objekt Kubernetes Service, který slouží jako přístupový bod. Kubernetes Service pro Ambasador má následující konfiguraci:

```
---
apiVersion: v1
kind: Service
metadata:
  name: ambassador
```

[20] Více informací o instalaci a konfiguraci nástroje Kubernetes Dashboard: https://github.com/kubernetes/dashboard

[21] Budoucí verze nástroje Docker pro všechny operační systémy by měli podporovat host.docker.internal. Více na https://github.com/docker/for-linux/issues/264

```
spec:
  type: LoadBalancer
  externalTrafficPolicy: Local
  ports:
   - port: 80
     targetPort: 8080
  selector:
    service: ambassador
```

Tato konfigurace nové Kubernetes Service je nasazena do klastru pomocí příkazu:

```
$ kubectl apply -f ambassador-service.yaml
```

Podrobný návod instalace a konfigurace tohoto nástroje lze nalézt na adrese: https://www.getambassador.io/user-guide/getting-started/

B.3 Konfigurace obrazového API

Obrazové API využívá nástroj Docker pro sestavení Docker obrazů uživatelských modulů. Jelikož tato aplikace také běží uvnitř Docker kontejneru, je potřeba upravit v rámci jejího konfiguračního souboru hodnotu parametru **docker.registry.url** na název hostitelského počítače.

Dále je potřeba vytvořit privátní Docker repozitář. Ten lze vytvořit zadáním příkazu:

```
$ docker run -d -p 5000:5000 --restart=always --name registry
   registry:2
```

Pokud je systém instalován na operačním systému **macOS**, je potřeba upravit konfigurační údaje pro nástroj Docker. V **exe** adresáři přiloženého média se nachází adresář **docker-config**, ve kterém je soubor **config.json**. Do tohoto souboru je potřeba zadat **Docker login** společně s **heslem** ve formátu base64. Dále je potřeba upravit hodnotu atributu **docker.registry.url** v konfiguračním souboru na Docker login.

Posledním konfiguračním krokem je nastavení přístupu ke Gitlab serveru. Nový kontejner s běžící instancí Gitlab serveru lze získat pomocí příkazu:

```
sudo docker run --detach \
  --hostname gitlab.example.com \
  --publish 443:443 --publish 80:80 --publish 22:22 \
  --name gitlab \
  --restart always \
  --volume /srv/gitlab/config:/etc/gitlab \
  --volume /srv/gitlab/logs:/var/log/gitlab \
  --volume /srv/gitlab/data:/var/opt/gitlab \
  gitlab/gitlab-ce:latest
```

Po vytvoření Gitlab serveru již stačí pouze vytvořit autentizační klíč pomocí návodu zde: `https://docs.gitlab.com/ee/user/profile/personal_`

`access_tokens.html`. Adresa nového Gitlab serveru a přístupový token je poté potřeba uložit do konfiguračního souboru obrazového API. Následně je potřeba nahrát šablonu uživatelských modulů na Gitlab server a upravit konfiguraci obrazového API.

B.4 Start komponent systému

Konfigurace jednotlivých obrazů je v adresáři **exe/docker**, ze kterého proběhne i jejich start. Pro spuštění je tedy potřeba přejít do tohoto adresáře a zadat příkaz:

```
$ docker-compose up
```

Po startu systému dojde k vystavení klientské aplikace na port 3000. Aplikace je pak přístupná na URL *http://localhost:3000/*, přístupové údaje jsou *test-user* a *heslo*.

Vypnutí systému se provádí stiskem Ctrl+C nebo zadáním příkazu:

```
$ docker-compose stop
```

Tímto příkazem dojde k vypnutí běžících kontejnerů, jejich obrazy a sestavení však budou nadále v hostitelském systému. Pro jejich odstranění je potřeba získat seznam jejich identifikátorů. Seznam všech sestavení se zobrazí po zadání příkazu:

```
$ docker ps -a
```

Pro odstraněn kontejnerů slouží příkaz:

```
$docker rm <identifikator_kontejenru>
```

Pro smazání bázových obrazů kontejnerů slouží příkaz:

```
$ docker rmi <identifikator_obrazu>
```

Identifikátory obrazů se zobrazí po zadání příkazu:

```
$ docker images
```

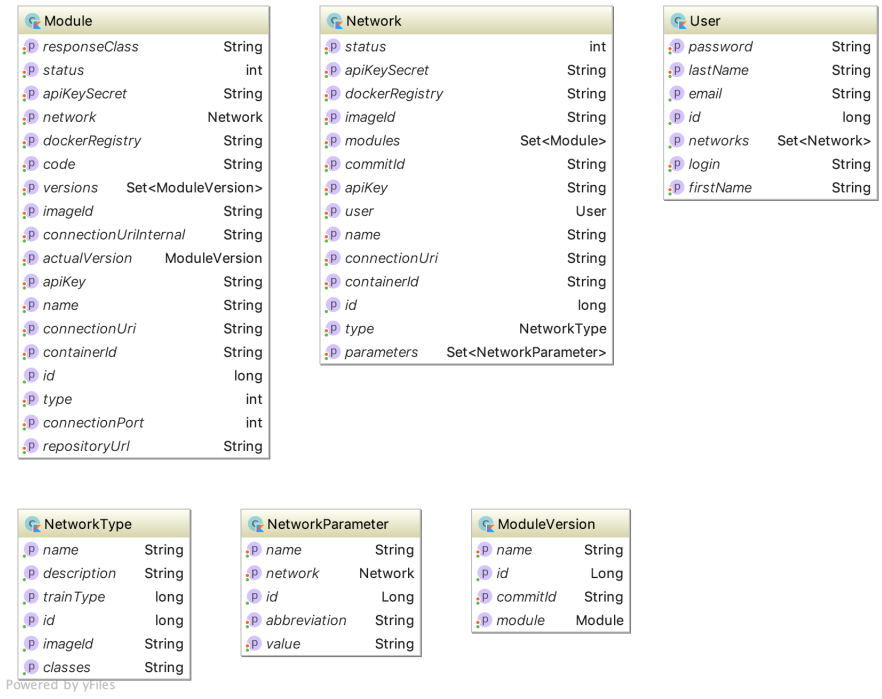
Příloha C

Relační databázový model

Obrázek C.1: Relační databázový model

Příloha D

Třídní model DAO vrstvy kontejnerového API



Obrázek D.1: Třídní model DAO vrstvy kontejnerového API

Příloha E

Seznam použitých zkratek

API Application Interface

CRUD Create, Read, Update, Delete

IP Internet Protocol

JSON JavaScript Object Notation

POJO Plain Old Java Object

REST Representational State Transfer

SQL Structured Query Language

URI Uniform Resource Identifier

URL Uniform Resource Locator

SOM Self-Organizing Map

HTML Hypertext Markup Language

DTO Data Transfer Object

autoML Automated Machine Learning

NLP Natural Language Processing

NAS Neural Architecture Search

ENAS Effective Neural Architecture Search

CNN Convolutional Neural Network

RNN Recurrent Neural Network

RBF Radial Basis Nunction network

JWT JSON Web Token

GPU Graphics Processing Unit

YAML YAML Ain't Markup Language

RBAC Role-Based Access Control

ILSVRC ImageNet Large Scale Visual Recognition Challenge

IaaS Infrastruktura jako služba

PaaS Platforma jako služba

SaaS Software jako služba

ReLU Rectified Linear Unit

AutoML Automatické strojové učení

NAS Neural Architecture Search

Příloha F

Seznam použitých pojmů

Framework je množina nástrojů knihoven a konvencí, které mají za cíl odstínit rutinní a základní problémy do modulů, které lze později znovu využívat.

Load balancing , neboli vyvažování zátěže je technika rozdělení zátěže mezi několik uzlů, počítačů či jiných zařízení.

Load balancer hardwarové zařízení či softwarový program starající se o load balancing.

Pipeline je funkcionalita platformy Angular, která umožňuje transformaci objektů předávaných komponentám. Tato transformace je iniciována přímo z *HTML* kódu.

REST API je rozhraní navržené podle pravidel REST architektury.

Log je informační záznam, který generuje určitá aplikace. Jsou v něm obsaženy informace o provedených akcích v rámci aplikace, či její chybové stavy.

Chatbot aplikace sloužící k interakci s uživatelem skrze psaný text.

Třída klasifikace je hlavní štítek nebo označení, kterým neuronová síť označí příchozí data.

Klasifikátor je aplikace nasazená v rámci systému Chobot, která se skládá z REST rozhraní pro komunikaci a připraveného modelu neuronové sítě pro klasifikaci příchozích požadavků.

Modul je aplikace nasazená v rámci systému Chobot, která se skládá z předem definované šablony a uživatelského kódu. Modul je vždy napojen na klasifikátor skrze třídu klasifikace.

Model neuronové sítě je model inspirovaný fungováním a strukturou lidského mozku, kdy jednotlivé neurony přijímají vstupní data a na základě nastavení vah aktivují svůj výstup.

Neuron je základní jednotkou neuronové sítě.

Aktivační funkce neuronu transformuje signál přijatý neuronem.

Učení s učitelem je metoda strojového učení, kdy model dostane vstupní data v podobě *[vstup, požadovaný výstup]* a hledá správné nastavení vah tak, aby chyba klasifikace byla minimální.

Učení bez učitele je metoda strojového učení, kdy model sám hledá podobnosti a nové informace v datech a nedostává jako vstupní trénovací data pár *[vstup, požadovaný výstup]*.

Shlukování je metoda, kdy model strojového učení kategorizuje jednotlivé položky na základě jejich podobnosti. Shlukovaní patří do skupiny metod učení bez učitele.

Přenosové učení transfer learning je způsob doučení již naučeného modelu neuronové sítě.

Dolaďování vrstev fine tunning je metoda přidání nových vrstev již naučenému modelu neuronové sítě.

Automatické strojové učení je metoda vyhledávání architektury a následného učení modelu neuronové sítě na základě přijatých trénovacích dat.

Distribuční model v rámci cloudových technologií slouží k nabídnutí výpočetních prostředků uživateli na základě jeho potřeb.

Klastr je seskupení několika počítačů (výpočetních uzlů), které se vůči svému okolí chovají jako jeden stroj.

Klasifikace obrázků je metoda, kdy klasifikátor určí jaký objekt se na obrázku nachází.

Detekce objektu na obrázku slouží k určení polohy určitého objektu na přijatém obrázku. Klasifikátor vrátí informace o tom jaký objekt se na obrázku nachází společně s jeho souřadnicemi.

Klasifikace log záznamů je metoda, kdy jsou jednotlivé záznamy klasifikovány do určitých tříd na základě jejich obsahu. V rámci této diplomové práce jsou log záznamy klasifikovány do dvou kategorií - normální a abnormální.

Klasifikace sentimentu textu je metoda, kdy model strojového učení určí subjektivní informace o daném textu.

Klasifikace tématu textu je metoda, kdy model strojového učení určí o jakém tématu je text předloženého článku (např. sport, hudba).

Statická analýza zdrojových kódů jsou metody analýzy zdrojového kódu aplikace, kdy jsou hledány nesprávné vzorce uvnitř kódu a probíhá kontrola dalších programátorských chyb.

Hledání architektury sítě je proces, při kterém je hledána architektura neuronové sítě na základě vstupních trénovacích dat.

Natural Language Processing je technika převodu psaného textu či mluveného slova do vhodné podoby pro modely strojového učení (obvykle do podoby číselného vektoru).

Příloha G

Obsah přiloženého CD

- `readme.txt` ..... stručný popis obsahu CD
- `exe` ..... adresář se spustitelnou formou implementace
- `src`
  - `impl` ..... zdrojové kódy implementace
    - `container_api` ..... zdrojové kódy kontejnerového API
    - `image_builder` ..... zdrojové kódy obrazového API
    - `client` ..... zdrojové kódy klientské aplikace
    - `networks` ..... zdrojové kódy všech klasifikátorů
    - `module` ..... zdrojové kódy šablony uživatelských modulů
  - `thesis` ..... zdrojové kódy práce
    - `thesis.tex` ..... zdrojová forma práce ve formátu LaTeX
    - `src` ..... obrázky a přílohy práce
- `text` ..... text práce
  - `thesis.pdf` ..... text práce ve formátu PDF
  - `manual` ..... manuál pro práci se systémem Chobot