VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INFORMAČNÍCH SYSTÉMŮ
DEPARTMENT OF INFORMATION SYSTEMS

PROGRAMOVACÍ JAZYK SCALA A JEHO VYUŽITÍ PRO ANALÝZU DAT

SCALA PROGRAMMING LANGUAGE AND ITS USE FOR DATA ANALYSIS

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE **TOMÁŠ KOHOUT**
AUTHOR

VEDOUCÍ PRÁCE **Doc. Ing. JAROSLAV ZENDULKA, Csc.**
SUPERVISOR

BRNO 2019

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav informačních systémů (UIFS) Akademický rok 2018/2019

Zadání bakalářské práce

Student: **Kohout Tomáš**
Program: Informační technologie
Název: **Programovací jazyk Scala a jeho využití pro analýzu dat**
Scala Programming Language and Its Use for Data Analysis
Kategorie: Data mining

Zadání:

1. Seznamte se s programovacím jazykem Scala a jeho využíváním pro analýzu dat.
2. Proveďte rešerši studií porovnávajících jazyk Scala s jinými programovacími jazyky, případně systémy, používanými pro analýzu dat z hlediska jeho výhod a nevýhod.
3. Po dohodě s vedoucím práce zvolte vhodnou případovou studii, na které budete ilustrovat zejména uváděné výhody jazyka Scala.
4. Pro zvolenou případovou studii vytvořte demonstrační aplikaci a ověřte její funkčnost na vhodných datech.
5. Shrňte svoje zkušenosti s použitím jazyka Scala pro analýzu dat.

Literatura:

- Odersky, M., Spoon, L., Venners, B.: Programming in Scala, Third Edition. Artima. 2016. 852 p.
- Swartz, J.: Learning Scala. O´Reilly. 2014. 236 p.

Pro udělení zápočtu za první semestr je požadováno:

- Body 1 až 3.

Podrobné závazné pokyny pro vypracování práce viz http://www.fit.vutbr.cz/info/szz/

Vedoucí práce: **Zendulka Jaroslav, doc. Ing., CSc.**
Vedoucí ústavu: Kolář Dušan, doc. Dr. Ing.
Datum zadání: 1. listopadu 2018
Datum odevzdání: 15. května 2019
Datum schválení: 16. října 2018

Abstrakt

Tato práce se zabývá porovnáním jazyka Scala s ostatními běžně používanými jazyky pro analýzu dat. Tyto jazyky se porovnávají z hlediska manipulace a zobrazení dat, strojvého učení a souběžného zpracování. Z tohoto porovnání následně vyplynou silné a slabé stránky jazyka Scala. Silné stránky jsou demonstrovány na implementované aplikaci pro kategorizaci e-mailů.

Abstract

This thesis deals with comparing the Scala programming language with other commonly used languages for data analysis. These languages are evaluated on the basis of the following categories: data manipulation and visualization, machine learning and concurent processing capabilities. The evaluation then shows the strengths and weaknesses of Scala. The strengths will be demonstrated on application for email categorization.

Klíčová slova

Scala, Multinomiální naivní Bayes, Support Vector Machines, Metoda nejbližších sousedů, AdaBoost, model aktérů, předzpracování textu, klasifikace textu, souběžné zpracování, zobrazení dat, manipulace s daty, strojové učení

Keywords

Scala, Multinimial naive Bayes, Support Vector Machines, k nearest neighbors, AdaBoost, actor model, text preprocessing, text classification, concurrent processing, data visualization, data manipulation, machine learning

Citace

KOHOUT, Tomáš. *Programovací jazyk Scala a jeho využití pro analýzu dat.* Brno, 2019. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Doc. Ing. Jaroslav Zendulka, Csc.

Programovací jazyk Scala a jeho využití pro analýzu dat

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana doc. Ing. Jaroslava Zendulky, Csc. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Tomáš Kohout
5. května 2019

Poděkování

Rád bych poděkoval svému vedoucímu, panu doc. Ing. Jaroslavu Zendulkovi, Csc., za pomoc, trpělivost a cenné rady. Své rodině za neutuchající podporu a tomu nejvyššímu za vedení.

Obsah

Kapitola 1

Úvod

Každý den vznikne $2.5 * 2^{18}$ bajtů dat a v roce 2012 bylo analyzováno pouze 0.5 % ze všech zaznamenaných dat [20]. Ze všech zaznamenaných dat je pouze část určena k analýze (přibližně 22 %). Přesto se jedná o enormní množství dat a není možné je všechna manuálně analyzovat. Pro urychlení analýzy dat se používají různé nástroje a programovací jazyky, které jsou analýze dat více či méně uzpůsobeny.

Cílem této práce je zjistit, zda je možné jazyk Scala použít jako nástroj pro analýzu dat. Nejprve bude v kapitole 2 provedeno porovnání jazyku Scala s jazyky Python a R, které jsou pro analýzu dat běžně používány, z hlediska podpory pro manipulaci s daty, možnosti zobrazení dat, podpory strojového učení a podpory souběžného zpracování. Na základě zjištěných silných stránek jazyka Scala bude v kapitole 3 popsán návrh a implementace demonstrační aplikace. V kapitole 4 je implementovaná demonstrační aplikace podrobena sérii experimentů, která dokazuje funkčnost a použitelnost jazyka Scala pro analýzu dat.

Kapitola 2

Jazyky používané pro analýzu dat

Tato kapitola se věnuje porovnání jazyků Python, R a Scaly z pohledu použití v různých oblastech, které se využívají při analýze dat. Jazyk Python a R jsem vybral, jelikož patří mezi nejčastěji doporučované jazyky pro analýzu dat [17]. Scala v tomto srovnání figuruje na 6. pozici.

V sekci 2.1 bude každý z porovnávaných jazyků popsán do té míry, aby byly příklady uvedené v dalších sekcích snadno srozumitelné. V sekci 2.2 budou popsány knihovny a vlastnosti těchto knihoven v tom kterém jazyce. Sekce 2.3 se bude věnovat knihovnám a způsobům jak tato data přetavit do grafické podoby. Následující sekce 2.4 se věnuje strojovému učení a knihovnám, které to umožňují. Poté, budou v sekci 2.5 rozebrány možnosti souběžného zpracování. V poslední sekci 2.6 bude shrnutí výhod a nevýhod pro každý z porovnávaných jazyků.

2.1 Základní charakteristika jazyků

Scala

Informace z tého části jsou čerpány z knihy *Programming in Scala* [23].

Scala je staticky typovaný programovací jazyk se silnou typovou kontrolou založený na objektově orientovaném a funkcionálním paradigmatu. Program je tedy ve většině případů tvořen definicí tříd, které respektují principy funkcionálního programování a pracují striktně s konstantami a konstantními kolekcemi. Použití proměnných je považováno za špatný návrh programu a často se doporučuje takový návrh přehodnotit.

Scala patří do rodiny jazyků, které běží v Java Virtual Machine (JVM), překládá se tedy do bajtkódu (bytecode). O správu paměti se, stejně jako v Javě, stará Garbage Collector.

Mezi základní charakteristiku Scaly patří stručnost a rozšiřitelnost (slovo Scala je složenina slov Scalable language tj. škálovatelný jazyk).

Ve Scale se nepoužívají středníky pro ukončení příkazu. Mohou se ovšem použít, pokud chceme napsat více příkazů na jeden řádek. Poté středníky zastupují znaky konce řádku.

Pro vytvoření funkce se používá slovo klíčové slovo *def* následované výčtem vstupních parametrů. Každý parametr má specifikovaný typ. Po ukončení výčtu parametrů je funkci přiřazen návratový typ, který může, ale i nemusí být specifikován, neboť překladač Scaly (Scalac) dokáže automaticky zjistit správný typ funkce a to na základě poslední hodnoty obsažené v těle funkce. Překladač tedy disponuje funkcí typového odvození. Způsoby, jak je možné definovat funkci jsou zobrazeny na ukázce kódu 1.

```
def sum(parameter0: Int, parameter1: Int): Int = {
    parameter0 + parameter1
}

// lze zapsat i takto
def sum(parameter0: Int, parameter1: Int) =
    parameter0 + parameter1

// ale také takto
def sum: (Int, Int) => Int = _ + _

// popřípadě takto
val sum: (Int, Int) => Int = _ + _
```

Kód programu 1: Příklad deklarace funkce.

Pro definování proměnné se používá klíčové slovo *var* nebo *val*. Rozdíl mezi těmito klíčovými slovy je ten, že *var* je proměnná a *val* je konstantní proměnná. Při pokusu o přepis hodnoty proměnné definované klíčovým slovem *val* dojde ke kompilační chybě.

Ve Scale je možné na různých místech zaměňovat kulaté závorky za složené závorky. To je umožněno jednak díky tomu, že složené závorky představují blok kódu, jehož hodnota je dána výrazem před poslední složenou závorkou a také tím, že v určitých případech je možné vynechat závorky při volání funkce s jedním parametrem. Pro funkce s více než jedním parametrem je již nutné použít kulaté závorky. Názorný příklad je znázorněn na ukázce kódu 2.

```
def pow: Int => Int = number => number * number

// výsledek jsou 4
pow (2)

// výsledek je 9, protože se použila poslední hodnota před
// ukončovací závorkou
pow {
    1
    2
    3
}
```

Kód programu 2: Příklad použití kulatých a složených závorek.

Pro řízení toku programu Scala definuje základní prostředky jako *if() else* větve či *if() else if() else*, *for* cykly či *while* cykly.

Pro vytvoření třídy se používá klíčové slovo *class*. Každá metoda a atribut bez explicitní definice viditelnosti je považována ze veřejnou. Viditelnost jednotlivých metod a atributů je možné upravit použitím klíčových slov *private* a *protected*. Atributy a metody označené jako *private* nelze přetěžovat. Konstruktor třídy je definovaný výčtem parametrů, které

následují po názvu dané třídy. Pokud třída žádné parametry neobsahuje, tak ji vytvoříme pouze pomocí klíčového slova *new* a jména dané třídy.

Pro vytvoření objektu je použito klíčové slov *objekt*. Scala tím umožňuje nahradit statické atributy a funkce. Je zaručeno, že takovýto objekt bude vytvořen pouze jednou.

Ke každé třídě je možné vytvořit tzv. doprovodný objekt. Atributy vytvořené v tomto objektu, stejně jako metody, jsou dostupné v těle dané třídy a to i v případě, kdy omezíme jejich viditelnost pro ostatní použitím klíčového slova *private* nebo *protected*. Doprovodný objekt a třída, kterou doprovází musejí mít stejný název. V tomto objektu je možné vytvořit speciální metodu *apply*, která slouží k instanciaci dané třídy. Tato metoda se následně volá pouze pomocí jména daného doprovodného objektu a složených závorek.

Scala dále umožňuje jmenné přiřazení jednotlivých parametrů, což zvyšuje přehlednost a snižuje se tím pravděpodobnost, že bude na daný parametr přiřazena špatná hodnota. Toto je předvedeno v ukázce kódu 3 v metodě *apply*. V této ukázce je také předvedena definice třídy, doprovodného objektu a instanciace třídy.

```
object MojeTrida {
    def apply(
      konstanta: Int,
      promenna: Int
    ): MojeTrida =
      new MojeTrida(
        konstanta = konstanta,
        promenna = promenna
      )

    private def add: (Int, Int) => Int =
        _ + _
}

// pro konstantní atributy třídy není
// nutné explicitně deklarovat parametr jako val
class MojeTrida(
  konstanta: Int,
  var promenna: Int
){
    def udelejNeco(): Unit =
      promenna = MojeTrida
        .add(konstanta, promenna)
}

val mojeTridaBezApply = new MojeTrida(1, 0)
// místo MojeTrida.apply(1,0)
val mojeTridaSApply = MojeTrida(1,0)
```

Kód programu 3: Definice a instanciace třídy.

Scala obsahuje specifický typ třídy, která se označuje a definuje pomocí slov *case class*. Použitím klíčového slova *case* označíme danou třídu jako tzv. datovou třídu. Instance této

třídy je vždy neměnná, stejně jako její parametry (pokud není explicitně uvedeno klíčovým slovem *var*, že daný parametr je proměnná). Pro každou datovou třídu je automaticky dostupná metoda *copy*, která vytvoří novou instanci dané třídy a upraví pouze ty atributy, které byly do této metody předány. Ostatní data zůstanou nezměněna. Použití datové třídy je ukázáno na ukázce kódu 4.

```
case class Data(col0: String, col1: String, col2: Int)

val data = Data("col0Data", "col1Data", 20)
// Data("upraveno", "col1Data", 20)
val upravenoData = data.copy(col0 = "otherData")
```

Kód programu 4: Definice a instanciace datové třídy.

Pro každou datovou třídu je při překladu automaticky vytvořený i doprovodný objekt, který obsahuje metodu *apply*. Dále je při kompilaci automaticky vytvořena i metoda *equals*, která porovnává instance stejné datové třídy na základě hodnot jednotlivých atributů.

Scala dále umožňuje tzv. porovnávání vzorů (pattern matching). K tomuto účelu slouží klíčové slovo *match* a pro samotné odchycení daného typu slovo *case*. Použití porovnání vzorů je ukázáno na příkladu 5.

```
val text: String = "match"

text match {
  case "match" =>
    println("match")
  case other =>
    println("ostatní případy: " + other)
}
```

Kód programu 5: Použití porovnávání vzorů.

Scala dále umožňuje vytvořit, pomocí klíčového slova *trait*, rozhraní, které mohou jednotlivé třídy rozšiřovat. Tato rozhraní v sobě mohou obsahovat deklarované nebo definované atributy a metody, ale nemohou být instanciovány a proto neobsahují žádné vstupní parametry. Příklad *traitu* a jeho použití je ukázán na ukázce kódu 6.

```scala
trait Zvire {
    protected val pocetNohou: Int
    protected val zvuk: String
    def vydejZvuk: Unit = println(zvuk)
}
//
class Pes extends Zvire {
    override val pocetNohou: Int = 4
    override val zvuk: String = "haf"
}

val pes = new Pes
// vypíše na konzoli "haf"
pes.vydejZvuk
```

Kód programu 6: Použití traitu.

Python

Python je vysokoúrovňový skriptovací jazyk, který obsahuje jak objektově orientované paradigma, tak také funkcionální, procedurální a imperativní paradigma. Programy v Pythonu mohou být napsány pouze pomocí jednotlivých za použití jednotlivých paradigmat nebo mohou využívat kombinací několika z nich. Např. je možné používat objektově orientované paradigma společně s imperativním paradigmatem nebo imperativní paradigma nahradit funkcionálním a získat tak lepší čitelnost kódu a testovatelnost.

Navzdory tomu, že Python není staticky typovaný, tak se řadí mezi silně typované jazyky. Pro správu paměti používá garbage collector.

Pro vytvoření bloku programu např. pro tělo funkce se používá místo složených závorek odsazení. Standardně jsou to čtyři mezery, ale povolen je i tabulátor. Jako ve Scale, tak ani v Pythonu se pro ukončení příkazu nepoužívají středníky. Výraz ukončuje konec řádky. Zápis více výrazů na jeden řádek je ovšem možný i zde a to za použití právě středníků, které zastupují znak konce řádku.

Pro vytvoření proměnné neexistuje žádné klíčové slovo. Stačí napsat jméno proměnné a přiřadit jí hodnotu. Interpret se postará o rozpoznání typu po spuštění programu.

Pro řízení toku programu slouží základní prvky jako *if else* a *if elif else* větvení, *for* a *while* cykly. Za zmínku stojí, že lze zapsat *while* cyklus s *else* větví, která se vykoná v případě, že podmínka ve *while* cyklu není splněna."

For cykly umožňují jednoduše procházet kolekce po jednom prvku a následně s tímto prvkem pracovat.

Pro vytvoření funkce se, stejně jako ve Scale, používá klíčové slovo *def* a pro vytvoření třídy se používá klíčové slovo *class*.

```scala
class MojeTrida:
    def mojeMetoda(someInt):
        return someInt * someInt
```

Kód programu 7: Deklarace třídy a metody

Z ukázky kódu 7 je patrné, že v definované metodě nejsou žádné pevně dané typy. Python od verze 3.6 sice umožňuje zapisovat typ parametru, ale ty slouží pouze jako anotace a interpret ji dále k ničemu nevyužívá. Pro rozpoznávání typů se používá pojem Duck typing. Idea ukrytá za tímto pojmem říká, že pokud daný objekt chodí jako kachna a kváká jako kachna, potom to musí být kachna. Tzn., že nezáleží na typu daného objektu, ale pouze na tom, zda daný objekt obsahuje atributy a metody, které jsou požadovány danou funkcí [12].

Python obsahuje syntaktické zjednodušení stejně jako Scala. Jedná se předně o tzv. magické metody, které jsou specifikované tím, že začínají dvěmi podtržítky (___). Např. klíčové slovo *in* je namapované na metodu ___*contains*___*()*. Více o magických metodách lze nalézt v [12].

R

Informace v této části jsou čerpány z [6]

R je dynamicky typovaný programovací jazyk, který je implementací jazyka S pod svobodnou licencí. Stejně jako Scala a Python je R multiparadigmatický jazyk, který umožňuje psát programy pomocí objektově orientovaného, funkcionálního, procedurálního či imperativního paradigmatu. Programy napsané v R jsou často v podobě skriptů, kdy je abstrakce soustředěna do definovaných funkcí a následně jsou tyto funkce použity pro dosažení kýženého cíle. Např. pro zjištění, které sloupce z analyzované tabulky dat, mají prázdná místa.

R je jazyk stvořený pro analýzu dat a statistické výpočty. Hojně se používá pro rychlou analýzu dat, protože je snadné v něm cokoli okolo analýzy dat vytvořit velmi snadno. Je nutné říct, že se nehodí prakticky pro žádnou jinou úlohu než pro analýzu dat. R je licencovaný licencí GNU GPL v2 a je velmi snadno rozšiřitelný.

V lednu roku 2019 byl ohodnocen jako 12. nejpopoulárnější programovací jazyk v žebříčku TIOBE.

R používá pro správu paměti garbage collector s tím, že je možné říci, kdy a co chceme smazat z paměti dříve než se garbage collector spustí. Tím můžeme dosáhnout menší paměťové náročnosti, zvláště tehdy, kdy pracujeme s velkým množstvím dat.

Pro náročné výpočty je možné připojit a za běhu programu volat funkce napsané v jazycích C, C++ či Fortranu. R umožňuje manipulovat s objekty v paměti skrze jiné programovací jazyky, mimo již zmíněné jsou to Java, .NET nebo Python. Tento jazyk je čistě objektově orientovaný a zároveň funkcionální.

Pro rozšíření vlastností jazyka slouží balíčky. Tyto balíčky nabízejí další funkce, podobně jako je tomu i v jiných programovacíhc jazycích. Balíčky jsou uloženy primárně v Komplexní archivační síti jazyka R (Comprehensive R Archive Network, dále jen jako CRAN). Další je možné nalézt například na GitHubu.

Základním typem je vektor. Vektor je instanciován tak, že se použije přiřazovací operátor (<- nebo =).

```
x <- 1:6
// výsledek bude [1,2,3,4,5,6]
y <- c(1, 2, 3:6)
```

Kód programu 8: Vytvoření vektoru.

V ukázce kódu 8 bude proměnná x po interpretování obsahovat vektor o hodnotách od 1 do 6 včetně. Stejného výsledku dosáhneme pokud použijeme generickou funkci *c(...)*, která kombinuje objekty do vektoru. Proměnná y bude obsahovat stejnou hodnotu jako proměnná y. Tři tečky v předpisu funkce zastupují variabilní parametry.

Pro přístup k jednotlivým hodnotám na určitém indexu je nutné použít hranaté závorky a index dané hodnoty např. x[1]. Číslování indexů pro vektory začíná od čísla 1, nikoli od 0.

R umožňuje pracovat s vektory zcela přirozeným způsobem. Toto je demonstrováno na ukázce kódu 9.

```
#Predpokladejme, ze x je vektor o~hodnotach [1,2,3,4]
#Vysledny vektor y poté obsahuje hodnoty [10,20,30,40]
    y <- x * 10
```

Kód programu 9: Násobení vektoru

Kromě vektoru je možné vytvářet matice a další. Pro vytvoření funkce je dán následující předpis. Jméno funkce je zapsán před přiřazovací znak <-. Po tomto znaku následuje klíčové slovo *function* a poté kulaté závorky ve kterých je možné deklarovat i definovat vstupní proměnné. Po kulatých závorkách následují složené závorky. Pro vrácení hodnoty z funkce je použito klíčové slovo *return*.

Funkce vytvořená na ukázce 10 zdvojnásobí numerické hodnoty předané argumentem. Pokud se do dané funkce dostane jiný typ než-li numerický, dojde při běhu programu k chybě.

```
doubleIt <- function(value){
        return(value * 2)
}
x <- 1:4

#vysledný vektor doubleX obsahuje hodnoty [2, 4, 6, 8]
doubleX <- doubleIt(x)
```

Kód programu 10: Vytvoření funkce.

2.2 Manipulace s daty

Scala

Informace v této kapitole jsou čerpány z [15].

Ve Scale neexistuje žádná udržovaná knihovna pro manipulaci s daty, ale lze použít implementace napsané pro jazyk Java. Zde se bude jednat o knihovně Tablesaw.

Tablesaw obsahuje datový typ Table, který je složený ze sloupců, které obsahují informaci o tom, jakého typu jsou hodnoty obsažené v tomto sloupci a řádků.

Tablesaw umožňuje importovat data z různých typů souborů a databází.

Pro práci s daty definuje velké množství operací, kdy je možné kombinovat tabulky, spojovat je pomocí JOIN operací a také vytvářet nové tabulky na základě filtrovacích kritérií. Jak takové filtrování vypadá je ukázáno na ukázce kódu 11.

Je dále jednotlivé sloupce odebírat, přidávat a transformovat v nich obsažené hodnoty. Samozřejmostí pak je zobrazení prvních a posledních n řádků.

```
val columns = List(
  StringColumn
    .create(
        "mesto",
        List("JH", "JH", "BR", "BR").asJava
        ),
  IntColumn
    .create(
        "rok",
        Array(1996, 1995, 2000, 1999)
        ),
  StringColumn
    .create(
        "jmeno",
        List("Julie", "Agata", "Petr", "Žofie").asJava
    )
)
// : _* je operátor, který transformuje list
// na jednotlivé hodnoty. Hodí se pro
// variabliní argumenty
val table = Table.create(collumns: _*)

// restulTable obsahuje pouze Julii
val resultTable = table.where(
  table
    .stringColumn("mesto")
    .isEqualTo("JH")
    .and(table.intColumn("rok").isGreaterThan(1995))
)
```

Kód programu 11: Filtrování dat podle sloupců.

Python

Informace v této kapitole jsou čerpány z [2].

Pro jazyk Python exituje knihovna Pandas, která umožňuje nahrát data z rozličných zdrojů a utvořit z nich datový rámec, který má podobu tabulky. Dalším datovým typem, který Pandas poskytuje je tzv. serie. Serie je jednorozměrné pole, které umožňuje držet data jakéhokoli typu. Tento datový typ je také možné chápat jako sloupeček obsažený v datovém rámci.

Pro inspekci dat poskytuje datový rámec velké množství metod. Mezi ně patří metoda *head(n)*, která sluoží pro zobrazení vrchních n řádků. Nebo metoda *describe()*, která vytvoří celkovou analýzu všech číselných sloupců.

Z datového rámce je možné přistupovat jak ke sloupcům, tak k jednotlivým řádkům. Pro přístup k jednotlivým sloupcům se používá název daného sloupce. Pro přístup k řádkům je nutné použít číslo daného řádku. Řádky jsou indexovány od 0.

Pandas dále poskytuje prostředky pro zjištění, zda v některém řádku chybí hodnota pro ten který sloupec a také poskytuje metody, které tyto řádky dokáží odebrat, případně je možné na tato prázdná místa doplnit libovolnou hodnotu.

Pandas poskytuje rozhraní, které umožňuje získat pouze ta data, která splňují určitá kritéria. Tato schopnost je demonstrována na ukázce kódu 12.

```
import pandas as pd
data = {
  'mesto': ["JH" , "JH", "BR" , "BR"],
  'rok': [1996, 1995, 2000, 1999],
  'jmeno': ["Julie", "Agata", "Petr", "Žofie"],
  }
df = pd.DataFrame.from_dict(data)

# vybere pouze "Julii"
df[(df['mesto'] == "JH") &  (df['rok'] > 1995)]
```

Kód programu 12: Filtrování dat podle sloupců.

Samozřejmostí je, že k již vytvořenému datovému rámci je možné přidávat řádky, sloupce či spojovat dva různé datové rámce a následně nad takto spojenými rámci lze provádět operace. Tato funkce je velmi užitečná v případě, kdy pracujeme s relačními databázemi.

R

Tato kapitola čerpá informace z [1].

Pro jazyk R existují dvě knihovny, které jsou nejčastěji zmiňovány jako tzv. "must-learn". Jedná se o knihovny dplyr a data.table. Knihovna dplyr je koncipovaná tak, aby byla uživatelsky příjemná a dobře uchopitelná zatímco knihovna data.table se vyznačuje svou stručností.

Knihovna dplyr obsahuje 5 základních funkcí a to select, filter, arrange, mutate a summarise. Select slouží pro vybrání jednoho nebo více sloupců. Filter se používá pro výběr několika řádků na základě daného filtrovacího kritéria. Arrange dokáže seřadit data podle jednoho či více sloupců a to buď vzestupně či sestupně. Mutate slouží k přidání sloupců a summarize pro získání informací.

```
DF = data.frame(
  jmeno = c("Julie", "Agata", "Petr", "Žofie"),
  rok = c(1996, 1995, 2000, 1999),
  mesto = ["JH" , "JH", "BR" , "BR"]
)
# vybere pouze "Julii", použití funkce filter
filter(DF, rok > 1995 & mest == 'JH')
```

Kód programu 13: Filtrování dat podle sloupců.

Knihovna data.table poskytuje oproti knihovně dplyr velmi stručné rozhraní. Toto rozhraní je následující *DT[i,j,by]*, kde *DT* je instance data.table, *i* slouží k řazení/vybrání jen některých řádků, *j* je funkce, která bude aplikována a *by* je parametr určený k seskupení dat. Na ukázce kódu 14 je demonstrováno filtrování dat na základě hodnot v jednotlivých sloupcích.

```
DT = data.table(
  jmeno = c("Julie", "Agata", "Petr", "Žofie"),
  rok = c(1996, 1995, 2000, 1999),
  mesto = ["JH" , "JH", "BR" , "BR"]
)
# vybere pouze "Julii", použití parametru i
DT[rok > 1995 & mesto == 'JH']
```

Kód programu 14: Filtrování dat podle sloupců.

2.3 Zobrazení dat

Pro zobrazení dat existuje množství knihoven, které umožňují jejich použití pro více jazyků. Jednou z nich je knihovna Plotly [21], která umožňuje vytváření základních grafů, mezi které patří sloupcové či koláčové grafy a také pokročilé 3D grafy, zobrazování dat na mapách a mnoho dalších. Pro zobrazení těchto grafů je výsledek zapsaný do html souboru, kdy po otevření tohoto souboru v internetovém prohlížeči je možné interaktivně pracovat s takto vytvořeným grafem. Všechny tyto grafy lze vytvořit v každém ze zde popisovaných jazyků.

Scala

Scala je na zobrazovací knihovny chudý jazyk. Nabízí se ovšem použití knihoven pro Javu, kde je nejspíš nejvhodnějším řešením Tablesaw, případně JFreeChart. Tablesaw je ovšem pouze obal pro knihovnu Plotly, ale tato knihovna nabízí sama o sobě podporu pro Scalu. JFreeChart je možné použít, ale neposkytuje žádné výhody oproti Plotly.

Python

Matplotlib vznikl v 80. letech 20. století [4] a je nejstarší knihovnou pro zobrazování dat napsanou pro jazyk Python. Je vytvořena tak, aby svým použitím připomínala Matlab. Tato vlastnost se z dnešního pohledu zdá být spíš na překážku, protože Matplotlib poskytuje velmi komplexní a těžkopádné rozhraní. Tato komplexnost je vykoupena velkou škálou grafů, které lze pomocí této knihovny vykreslit. Pro získání představy, co vše lze s touto knihovnou tvořit viz https://matplotlib.org/gallery/index.html.

R

Pro vytvoření grafů v R se mimo Plotly používá i mnoho jiných knihoven. Nejznámější z nich je ovšem knihovna ggplot2 [24]. Ggplot2 je systém pro deklarativní vytváření grafů. Idea této knihovny spočívá v tom, že uživatel poskytne data, určí popis jednotlivých os a které grafické zobrazení se má použít. Ggplot2 se poté postará o zbytek. Ggplot2 je,

společně s knihovnou dplyr 2.2, součástí stejného souboru knihoven[1] určeného pro analýzu dat.

2.4 Strojové učení

Scala

Scala disponuje menším množstvím knihoven pro strojové učení než ostatní porovnávané jazyky. Jednou ze základních knihoven pro strojové učení je SMILE neboli Statistical Machine Intelligence & Learning Engine. Tato knihovna se snaží vytvořit knihovnu, která svým rozsahem a použitelností bude připomínat Python knihovnu scikit-learn. Obsahuje velké množství základních, ale i pokročilých algoritmů pro strojové učení. Mimo jiné nabízí vlastní vizualizační nástroje pro základní vykreslování grafů a čtení z různých typů souborů. Pyšní se i velkou rychlostí zpracování, ale paměťové nároky mohou být značné. Bohužel komunita kolem této knihovny není příliš velká. Na webových stránkách githubu této knihovny je pouze 33 přispěvatelů. Další možnou volbou může být Deeplearning4j implementovaný v jazyce Java. Deeplearning4j implementuje především neuronové sítě a je možné ji využít v kombinaci s Apache Spark či Hadoop. Deeplearning4j umožňuje využívat grafické karty pro urychlení výpočtů. Tuto knihovnu podporuje 242 přispěvatelů.

Python

Nejznámější a nejpoužívanější knihovnou je Scikit-learn, která má širokou podporu komunity a obsahuje mnoho algoritmů pro klasifikaci, regresi či rozhodovací stromy. Scikit-learn je závislý na matematických knihovnách NumPy a SciPy. Tyto knihovny jsou z větší části napsané v jazyku C pro dosažení maximálního výkonu. Jde tedy o velice efektivní řešení matematických výpočtů. Scikit-learn se těší podpory komunity, která čítá k datu 22.12.2018 1 219 [2] podporovatelů, kteří přímo pomáhají vyvíjet a vylepšovat tuto knihovnu. Další velmi známou knihovnou je TensorFlow. Tato knihovna je přímo podporovaná 1 763 lidmi, kteří ji dále vylepšují a zdokonalují. TensorFlow se používá předně pro algoritmy hloubkového učení. Tím, co činí TensorFlow odlišným od ostatních knihoven pro strojové učení je, využívání data flow grafu. Tímto přístupem se člověk, který navrhuje zpracování dat může soustředit spíše na logiku celé věci a není nucený zaměřovat se na drobnosti kolem. Výhodou poté je, že takto napsaný program je možné spouštět jak na procesoru, tak na grafických kartách, v mobilních zařízeních (IOS, Android) či na cloudu s vysokým výpočetním výkonem a mnoha zařízeními.

R

Jazyk R disponuje enormním množstvím balíčků pro strojové učení, kdy se velké množství balíčků specializuje na jeden konkrétní model či oblast např. klasifikaci. Podle studie[18] vypracované v roce 2017 byl nejpoužívanějším balíčkem pro strojové učení označen caret. Caret je podobný jako scikit-learn, neboť nabízí v mnoho (238 [16]) algoritmů v oblasti klasifikace, regrese atd. Mimo jiné také nástroje pro předzpracování dat a nástroj pro vytváření grafů.

[1] Tidyverse: https://www.tidyverse.org/
[2] https://github.com/scikit-learn/scikit-learn

2.5 Souběžné zpracování

Scala

Informace pro tuto sekci jsou čerpány z [11] a [13]. Použití jazyka Scala pro souběžné zpracování je velmi podpořenou používáním konstantních objektů. Pouze používání konstantních objektů zcela odstraňuje chybu způsobenou souběhem dvou a více vláken, které se snaží či jinak upravit proměnnou. Při použití konstantních objektů toto zkrátka není možné. Další výhodou konstant je, že se programátor nemusí zatěžovat přemýšlením nad tím, zda se k tomuto objektu nemůže dostat i jiné vlákno. I kdyby se tak stalo, tak toto vlákno může danou hodnotu pouze přečíst.

Ve Scale existuje mnoho způsobů, jak využít plný potenciál více jádrového procesoru. Uvedu zde dva nejběžnější způsoby a to tzv. Future a poté framework Akka inspirovaný modelem aktérů z jazyku Erlang [3].

Future

Future poskytuje jednu z možných cest, jak přemýšlet o souběžném vykonávání kódu. Future představuje objekt, který, po svém vykonání, bude obsahovat výslednou hodnotu. Pro vykonání tohoto kódu je nutné použít exekuční kontext.

Exekuční kontext je zodpovědný o vykonání kódu obsaženého ve Future a přidělování výpočetního výkonu. Scala obsahuje implementaci exekučního kontextu, který má k dispozici stejný počet vláken, jako procesor, na kterém se daný program spouští, jader.

Future je označení pro pseudo monádu, která při svém vytvoření vykoná tělo funkce v ní vložené v jiném vlákně, přičemž s výsledkem lze dále pracovat i z vlákna, ve kterém byla daná Future vytvořena a to buď a to zcela bez blokování. Tuto monádu můžeme různě transformovat a upravovat. Tyto úpravy a změny dat uložených ve Future jsou taktéž provedeny v separátním vlákně.

Future může skončit ve dvou stavech. Pokud byl daný kód vykonán bez toho, aniž by skončil chybou, tak taková Future bude označena jako úspěšná (successful) a bude obsahovat výslednou hodnotu. Pokud však dojde při vykonávání kódu k chybě, tak tato Future skončí jako neúspěšná (failed) a bude obsahovat chybu, která způsobila přerušení vykonávání daného kódu.

Future obsahuje *map* a *flatMap* metody, díky kterým je možné řetězit za sebe vykonávání určitých příkazů a transformovat výslednou hodnotu. Pokud v jednom z těchto příkazů dojde k výjimce, tak se další příkazy již nevykonávají. Toto je ukázáno na příkladu 15.

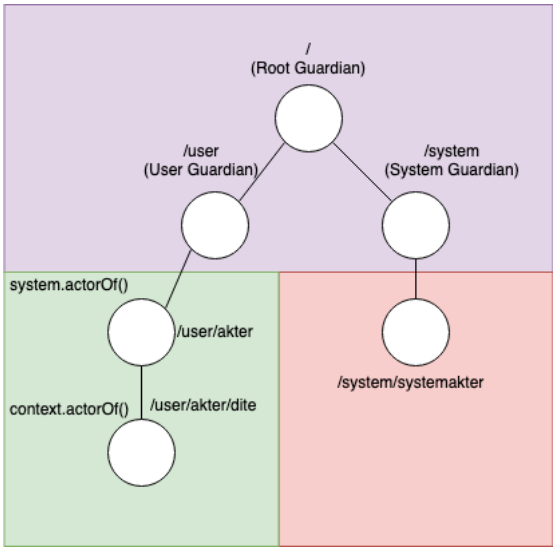
```scala
import ExecutionContext.global

val vysledek: Future[String] = Future(10+4)
  .map( cislo => cislo * 10)
  .flatMap(cislo =>
    // flatMap očekává Future libovolného typu
    Future
      .successful("Vysledek je: " + cislo)
  )

// vykonávání kódu v~tomto vlákně pokračuje
// bez jakéhokoli blokování.
```

Kód programu 15: Ukázka Future.

Akka

Akka je framework, který umožňuje přemýšlet o souběžném zpracování neblokujícím způsobem. K tomuto účelu je použitý model aktérů, kteří se mohou ovlivňovat pouze tím, že si posílají zprávy. Neexistuje žádný jiný způsob, jak jinak komunikovat s aktérem. Pokud to aktér nedovolí, a rozhodně by neměl, tak žádný další aktér nezná vnitřní hodnoty, které v sobě uchovává. Model aktérů je v podstatě objektově orientované paradigma dotažené do dokonalosti.

Akka zaručuje, že pokud aktér A posílá zprávu A_1 aktérovi B a následně pošle aktérovy B zprávu A_2, tak tyto zprávy dojdou přesně v tom pořadí, v jakém byly odeslány. Akka ovšem nezaručuje, že pokud aktér A pošle zprávu A_1 aktérovi B a následně aktér C pošle zprávu C_1 aktérovi B, tak zde již není pořadí doručení zaručeno. Může se tedy stát, že nejdříve přijde aktérovi B zpráva C_1 a následně A_1.



Obrázek 2.1: Základní hierarchie a unikátní názvy aktérů.

Na obrázku 2.1 je zobrazena základní hierarchie aktérů. Ve fialovém poli jsou kořenový aktéři, kteří jsou první a poslední instancí, která se stará o celý systém aktérů. V zeleném

poli jsou zobrazeni aktéři, které vytváří uživatel Akky a v červeném poli jsou zobrazeni aktéři, které používá systém aktérů pro vlastní účely.

Aktér s cestou /user je kořenovým aktérem pro všechny aktéry, které jsou vytvořeny pomocí systému aktorů. Systém aktorů je instance třídy ActorSystem, která se stará o všechny vytvořené aktéry. Tento systém je vždy vytvořený pouze jeden.

Mimo rozdělení do systémových a uživatelských aktérů je na tomto obrázku patrné i to, jak jsou jednotlivý aktéři v rámci systému označováni. Každý aktér má unikátní název tvořený cestou od kořenového aktéra až po svůj vlastní název. Každá tato cesta je validní URL[3] adresa.

Uživatel Akky vytváří aktéry pomocí metody *actorOf()* a to nejprve na nejvyšší úrovni pomocí samotné instance systému aktérů a poté v samotných aktérech pomocí vnitřní konstanty označené jako context a metody *actorOf()*.

Pokud si aktér vytvoří potomka, tak může specifikovat, co se stane, pokud se v tomto potomkovi objeví chyba. Akka je v základu nastavena tak, aby pokud se v jakémkoli aktérovi vyskytne neočekávaná výjimka, tak dojde k zahození právě zpracovávané zprávy a následně je daný aktér restartován a jeho vnitřní stav je vrácen do počáteční hodnoty. Po restartování začne tento aktér zpracovávat další zprávy, které má přítomné v poštovní schránce.

Každý aktér obsahuje referenci na svého rodiče.

Pro vytvoření předpisu aktéra je nutné vytvořit třídu, která bude rozšiřovat abstraktní třídu Actor a implementuje metodu *receive: Receive*. Vytvoření třídy a implementování metody *receive:Receive* je ukázáno na ukázce kódu 16.

```
import akka.actor.{Actor, Props}
objekt MujAkter {
    def apply(): Props[MujAkter] = Props(new MujAkter)
}
class MujAkter extends Actor {
    override def receive: Receive = {
        case "stop" =>
            context.stop()
        case zprava: String =>
            // ! == funkce tell
            sender() ! "Prijato: " + zprava
        case _ => ()
    }
}
```

Kód programu 16: Implementace aktéra.

Na ukázce kódu 16 je také ukázáno použití funkce tell, která slouží k posílání zpráv mezi aktéry. Zpráva je definována svým typem a pro rozlišení jednotlivých zpráv je použito porovnávání vzorů. Metoda *sender()* slouží k získání reference aktéra, od kterého daná zpráva přišla. Tomuto aktérovy by měla být vždy doručena alespoň nějaká odpověď. V případě, že do aktéra *MujAkter* přijde řetězec *stop*, tak dojde k zastavení daného aktéra. Jakékoli zprávy, které jsou přítomné v poštovní schránce budou přesměrovány na aktéra, který se stará o nedoručené zprávy. Tyto zprávy budou zničeny.

[3]Co znamená url se lze dočíst zde: https://en.wikipedia.org/wiki/URL

Na další ukázce kódu 17 je ukázáno vytvoření aktéra a následný synchronní volání pomocí funkce *ask* (nebo také ?).

```
import akka.actor.ActorSystem
import ExecutionContext.global
import scala.duration._
import scala.util.{Failure, Success}

object Main {
    def main(args: Array[String]){
        val system = ActorSystem("jmenoActorSystemu")
        val mujAkter: ActorRef = system.actorOf(MujAkter())

        (mujAkter ? "Zprava")(1 second)
          .map(println)
          .flatMap(_ => system.terminate)
    }
}
```

Kód programu 17: Vytvoření aktéra.

Metoda *ask* vytvoří Future, která bude obsahovat odpověď od aktéra *mujAkter*. Tato odpověď bude následně zobrazena na standardní výstup a poté dojde k ukončení aplikace. Metoda *ask* potřebuje nastavit časový limit, po který bude vyčkávat na odpověď. Pokud se do uplynutí této doby nepodaří získat odpověď, tak dojde k selhání dané Future, která bude poté obsahovat chybu TimeoutException.

Akka dále umožňuje vytvářet aplikace, které jsou schopné využívat výpočetní výkon více počítačů. V takovém případě hovoříme o tzv. clusteru.

Akka umožňuje vytvářet tzv. cluster jedináčky. Akka zajišťuje, že daná instance aktéra existuje v daném clusteru pouze jednou a v případě že uzel (jeden z počítačů, kteří tvoří cluster), na kterém je přítomna instance tohoto jedináčka (vždy je to ten nejstarší uzel), bude ukončen, tak dojde k vytvoření nové instance na druhém nejstarším uzlu, který je přítomný v clusteru bez toho, aby se výrazným způsobem narušil chod aplikace.

Akka dále umožňuje vytvářet tzv. směrovače. Směrovač pomocí různých strategií (Round-robin aj.) rozesílá zprávy cílovým aktérům, kteří tyto zprávy zpracovávají. Tento směrovač lze nastavit co do počtu vytvořených aktérů, tak také lze určit, na kterém uzlu mají být tyto instance drženy. To je zajištěno tím, že každému směrovači lze přiřadit určitou roli. Tuto roli poté určíme i v aplikačním nastavení na daném uzlu.

Každý směrovač a cluster jedináček může být označen rolí. Tato role poté souží k vybrání uzlu, na kterém budou aktéři označeni touto rolí vytvořeni.

Python

Pro souběžné zpracování se v Pythonu používá balíček multiprocessing nebo threading. Informace pro tuto kapitolu jsou čerpány z [8] a [7].

Threading

Balíček threading poskytuje základní primitiva nezbytná pro jakékoliv souběžné zpracování. Jeho rozhraní definuje zámky, semafory či bariéry, které se dále používají pro synchronizaci vláken. Třída Thread, která definuje právě jedno vlákno poskytuje rozhraní, které slouží k základnímu ovládání a identifikaci vlákna. Do tohoto rozhraní patří metoda *start()*, *run()* či *join()*. Nové vlákno se vytvoří instanciací třídy Thread. Třída Thread obsahuje parametr target, který přijímá funkci. Tato funkce se po spuštění vlákna metodou *start()* zavolá z metody *run()*. Parametr args definuje parametry, které budou této funkci předány při spuštění vlákna. Další možností pak je vytvořit třídu, která rozšíří třídu Thread, která bude implementovat metodu run.

Vytváření vláken je navíc omezeno globálním zámkem interpretu, který se stará o to, aby procesorový čas spotřebovávalo vždy pouze jedno vytvořené vlákno. Proto se vlákna v Pythonu používají pro blokující operace jako je čtení souborů z disku nebo provádění dotazů na vzdálené servery. Pro dosažení souběžného běhu se v Pythonu používá knohovna Multiprocessing.

Multiprocessing

Balíček multiprocessing je podobný balíčku Threading s tím rozdílem, že místo vláken vytváří procesy a proto není limitovaný globálním zámkem interpretu. Autoři se velice snažili, aby výsledné rozhraní vypadalo velice podobně, jako v balíčku threading a to se jim také podařilo. Proto konstruktor třídy Process, která reprezentuje jeden spuštěný proces, obsahuje stejné parametry jako konstruktor třídy Thread. Multiprocessing lze konfigurovat třemi způsoby a pouze jeden způsob je možné použít jak pro operační systém Windows, tak pro Unixové typy operačních systémů. Tyto způsoby se liší především v tom, která data budou přenesena i do nově vytvořeného procesu a která nikoli. Každý proces má totiž vlastní paměťový prostor. Jeden ze způsobů se jmenuje spawn. Spawn je možné použít v operačním systému Windows a Unixu. Při vytvoření procesu zkopíruje pouze nezbytná data, potřebná pro běh metody uvedené v konstruktoru třídy Process v parametru target. Vše ostatní, jako např. souborové deskriptory či jiné otevřené vstupně/výstupní cesty zkopírovány nejsou. Tento způsob vytváření procesů je na rozdíl od způsobu fork či forkserver dostupných pouze na platformě Unix pomalejší. Spawn se defaultně použije v operačním systému Windows. Další je fork, který lze použít pouze na operačních systémech Unixového typu. Tento způsob vytváření procesů je rozdílný od způsobu spawn v tom smyslu, že v momentě vytvoření procesu jsou oba procesy (rodič i dítě) zcela totožné. Kopíruje se tedy celý dosavadní stav rodičovského procesu. Vytváření procesů v multivláknové aplikaci je problematické. Pro tento způsob užití je doporučené používat forkserver. Fork je defaultně použitý na Unixové platformě. Poslední způsob se jmenuje forkserver a vyznačuje se tím, že při spuštění programu se vytvoří serverový proces, který se stará o vytváření všech dalších procesů a stejně jako u způsobu spawn se kopírují pouze ty zdroje, které jsou nutné pro spuštění dané funkce.

Pykka

Balíček pykka zde uvádím pouze pro zajímavost. Jedná se totiž o implementaci modelu aktérů napsanou v Pythonu. [4]

[4] Webové stránky pykka: https://www.pykka.org/en/latest/

R

R nabízí několik způsobů, jak využít více procesorů pro zrychlení běhu programu. Mezi tyto způsoby patří např. balíček parallel a balíček foreach, které umožňují procházet různé datové struktury souběžně. Množství vláken, které se využijí pro zpracování je dáno vstupním parametrem při inicializaci tzv. clusteru v případě balíčku parallel. Tento cluster se inicializuje i v případě balíčku foreach. Informace pro tuto sekci byly čerpány z [10].

Parallel

Balíček parallel umožňuje spouštění souběžné zpracování ve dvou základních typech. První typ se jmenuje FORK a druhý PSOCK.

Typ FORK je dostupný pouze pro unixové operační systémy, a proto programy napsané pomocí typu FORK nebudou fungovat na systému Windows. Typ PSOCK je možné spustit na jakémkoli operačním systému. FORK nabízí oproti PSOCK implicitní přejímání kontextu proměnných. Inicializace clusteru je již ze základu nastavena na typ PSOCK. Pro použití typu FORK je nezbytné toto explicitně vynutit tak, že vyplníme hodnotu parametru type hodnotou FORK ve funkci *makeCluster*. Po ukončení výpočtů je nezbytné daný cluster ukončit pomocí funkce *stopcluster()*, která na vstupu očekává referenci na rušený cluster.

Souběžné zpracování se poté používá pomocí funkce *parLapply()*, která na vstupu očekává inicializovaný cluster, data, na kterých bude provedena funkce, která je definovaná jako třetí parametr.

Pokud se pokusíme použít proměnnou definovanou v aktuálním rámci ve funkci, kterou předáme do paralelně zpracovávané funkce *parLapply*, kdy cluster je nastavený v režimu PSOCK, dostaneme chybu říkající, že daná proměnná není definovaná. Tento problém se řeší tím, že do daného clusteru exportujeme všechny proměnné, které chceme v daném clusteru používat. Toto platí i o balíčcích, které daná funkce používá. Tento případ je demonstrován na ukázce kódu 18. Na této ukázce je předvedeno i exportování knihoven pomocí funkce *clusterEvalQ()*.

```
exportovanaPromenna <- " Append"
clusterExport(cl, "exportovanaPromenna")

clusterEvalQ(cl, library(nejakaKnihovna))

parLapply(cl , c("a", "b", "c"),
              function(str)
              paste(str, exportovanaPromenna)
              )
```

Kód programu 18: Exportování proměnných a knihoven do clusteru.

Je důležité uvést, že změny proměnné provedené po exportování se neprojeví na objektu, který byl importován.

Foreach

Foreach je balíček, který obsahuje funkci *foreach()*, která umožňuje podobnou funkcionalitu jako funkce *lapply()* s tím rozdílem, že je možné ji používat jak pro sekvenční, tak

pro souběžný výpočet. Pro souběžné použití je nutné použít tzv. paralelní backend, který obsahuje mechanismus umožňující souběžný výpočet. Jde tedy o podobný princip jako v případě Scaly (exekuční kontext) 2.5. Pokud není žádný takový balíček použit, tak i souběžně spuštěný foreach je spuštěný pouze sekvenčně. Implementací souběžného backendu pro knihovnu Foreach je velké množství. Mezi tyto knihovny patří např. knihovna doParalel. Na ukázce kódu 19 je zobrazeno použití knihovny Foreach. I zde se vytváří cluster. Klíčové slovo %dopar% označuje požadavek na souběžný výpočet.

```
library(foreach)
library(doParrallel)

no_cores <- detectCores() - 1
cl <- makeCluster(no_cores)
registerDoParallel(cl)

append <- "Append"
foreach( variable = c("a", "b", "c"),
         .combine = c) %dopar%
            paste(variable, append)
```

Kód programu 19: Foreach knihovna a souběžný výpočet.

V tomto případě nebylo nutné exportovat proměnnou append, protože Foreach je spuštěný ve stejném kontextu jako je inicializovaná proměnná. Pokud ovšem funkci *foreach()* přesuneme do samostatné funkce, tak poté proměnná append již nebude ve stejném kontextu a díky tomu skončí program chybou. Funkce *foreach* nabízí možnost exportu proměnných stejně jako balíček Parallel. Exportovat proměnnou je možné pomocí parametru .export do kterého se přiřadí textový řetězec obsahující jméno proměnné, která má být exportována z aktuálního kontextu do kontextu funkce *foreach()*.

Pro exportování knihoven se používá podobný přístup, jako při exportování proměnných s tím rozdílem, že pro export knihoven se používá parametr .packages.

V případě synchronního přístupu s cílem zápisu k jedné proměnné je zapotřebí použít zámků, což zesložiťuje výsledný program.

2.6 Shrnutí

Na předchozích stránkách jsem porovnal způsoby, jakým se každý z trojice jazyků vypořádává s daným funkčním okruhem. Každý z trojice jazyků Scala, R a Python poskytuje nástroje, které se dají použít pro dané zaměření. Pro zobrazení dat Scala postrádá knihovnu, která by byla v tomto jazyku napsaná, ale je možné použít knihovny napsané v jazyce Java. To samé platí pro manipulaci s daty. Podpora Scaly pro strojové učení je dostačující. Knihovna SMILE pokrývá velmi širokou škálu různých typů algoritmů pro strojové učení. Scala se dále ukázala být velmi mocným nástrojem pro souběžné zpracování. Model aktérů velmi usnadňuje návrh, spravování a škálovatelnost systému, což je vhodné právě pro analýzu velkého množství dat, kdy výpočetní výkon pouze jednoho procesoru nestačí.

Podle srovnání stránky Stackshare[5], je jazyk Scala oblíbeným nástrojem díky tomu, že je kompilovaný, spustitelný na Java Virtual Machine, je silně a staticky typovaný, obsahuje porovnávání vzorů a má sobě vlastní podporu pro souběžného zpracování.

V případové studii se proto zaměřím na použití knihoven Akka a SMILE na kterých budu demonstrovat silné stránky jazyka Scala.

[5]Stack share srovnání jazyků Scala, R a Python: https://stackshare.io/stackups/python-vs-r-vs-scala

Kapitola 3

Případová studie

Jako případová studie byla zvolena úloha kategorizace e-mailových zpráv, protože umožňuje demonstrovat silné stránky jazyka Scala, což je souběžné zpracování. Cílem je navrhnout a implementovat systém pro takovou kategorizaci.

Tato kapitola nejprve obsahuje teoretický základ v podobně předzpracování textu pro klasifikaci a popis použitých klasifikačních modelů. V další části je popsán návrh a implementace samotného systému.

3.1 Předzpracování textu

Informace pro tuto část jsou čerpány z [9].

Předzpracování textu znamená, převést text do takové podoby, která je analyzovatelná a predikovatelná pro zvolený cíl. K tomuto účelu slouží různé metody. Mezi tyto metody se řadí stematizace, filtrování stop slov, normalizace a odstranění šumu.

Stematizace je metoda, která se pokouší převést slova do základního tvaru a to tak, že odsekne část konce slova. Např. slova trouble, troubling a troubled jsou převedena na stejné slovo troubl. Tato metoda nepřevádí slova přímo do základního tvaru, ale do tvaru, které se tomuto kořenu alespoň přibližuje. Mezi nejznámnější algoritmy určené pro stematizaci patří Porterův algoritmus [1].

Stop slova jsou dle [14], taková slova, která se ve psaném projevu vyskytují velmi často, ale jejich význam je čistě syntaktický. Nejčastěji jde o spojky, předložky atp. Tato slova tedy lze zcela vyfiltrovat bez toho, aby zmizel celkový význam dané věty.

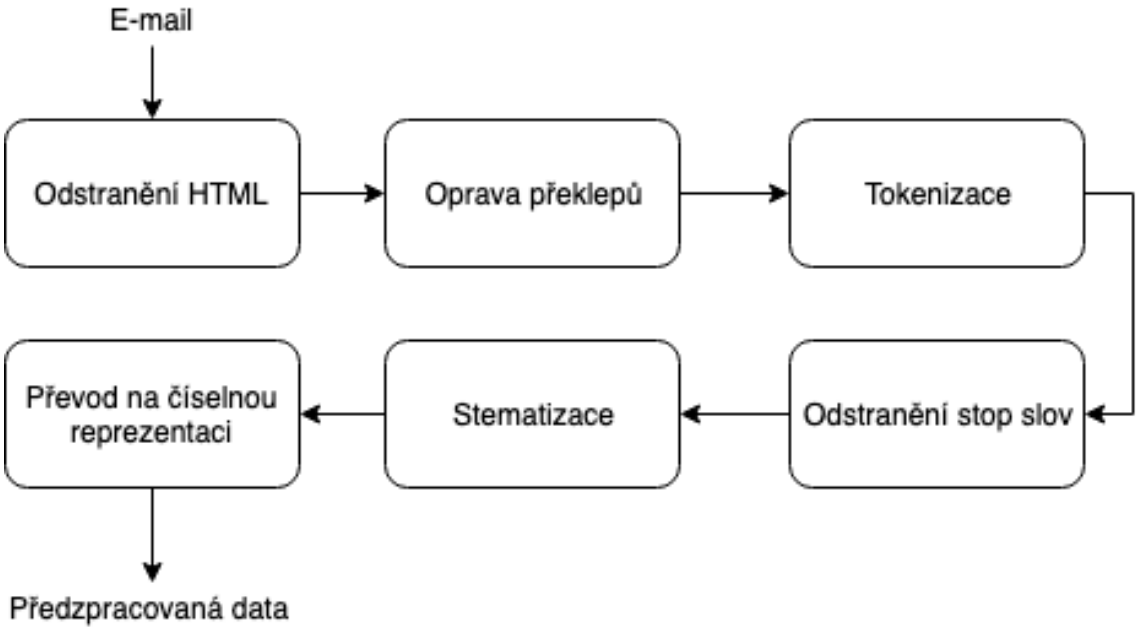
Normalizace označuje proces, kdy je slovo převedeno do standardního tvaru. Ne vždy se musí jednat o slovo. Např. emotikony mohou být normalizovány a převedeny na odpovídající slovo, čímž se zachytí další význam dané věty. Normalizace se také zabývá převodem slangových výrazů do standardního tvaru nebo převedením všech znaků na malá písmena.

Odstranění šumu je nezbytnou součástí předzpracování textu, protože text neobsahuje pouze slova, ale také různé řídicí znaky (čárky, tečky apod.), které, pokud zůstanou v textu ponechány, výrazným způsobem ztíží případnou predikci, ale i natrénování modelu. Tyto

[1] Porterův algoritmus: https://tartarus.org/martin/PorterStemmer/

znaky je proto vhodné odstranit. Dalším typem rušení mohou být různé formátovací značky např. HTML. Tyto značky je potřeba také odstranit, aby lépe vynikl samotný text.

V případě předzpracování e-mailových zpráv připadá do úvahy takové předzpracování, které je zobrazené na obrázku 3.1.



Obrázek 3.1: Postup předzpracování e-mailu do podoby určené ke klasifikaci/trénování.

- Odstranění HTML značek je nezbytným předpokladem pro vyčištění od nepotřebných značek (které jsou ve formě slov např. span), které jsou v HTML použity. Tyto značky by neblahým způsobem ovlivnily úspěšnost predikce.
- Oprava překlepů je nezbytná, protože se jedná o e-maily, které jsou psány lidmi a lidé dělají chyby. Součástí této fáze může být i převod na malá písmena.
- Tokenizace je jiný výraz pro rozdělení textu na jednotlivá slova nebo-li tokeny. Součástí této fáze bývá i odstranění řídících znaků.
- Stematizace slov do přibližného základního tvaru.
- Stematizovaná slova jsou poté převedena na číselnou reprezentaci. Způsob převedení na číselnou reprezentaci může být různý, ale často se používají principy typu Bag of Words (počet výskytů daného slova v textu dále jen BoW), či zda se dané slovo vyskytuje v použitém slovníku. Slovník často obsahuje slova, která jsou nejčastěji používaná pro každou klasifikační třídu. Pro aplikaci bude použita BoW reprezentace.

3.2 Klasifikace

Informace pro tuto sekci jsou čerpány z [19] a také z [22].

Klasifikace se soustředí na správné přiřazení třídy, do které patří klasifikovaná data. Klasifikační modely řadíme na modely s učitelem a bez učitele. Rozdíl mezi těmito dvěma

přístupy je ten, že pro modely s učitelem je nutné připravit nejprve množinu vstupních tj. trénovacích dat, které správně označíme třídou, do které náleží. Poté se tato data poskytnou modelu, který se z nich, podle specifického algoritmu, naučí rozlišovat jednotlivé třídy. Tyto modely se následně používají pro samotnou klasifikaci. Strojové učení bez učitele se zaměřuje převážně na hledání vzorů ve vstupních datech.

V této práce se zaměřím na metody s učitelem a klasifikační algoritmy použitelné při klasifikaci textových dokumentů. Tyto metody předpokládají, že existují vstupní data, která jsou správně zařazena do třídy, do které patří - tzv. trénovací data. Klasifikační model se na těchto trénovacích datech naučí, která data patří do které třídy. Po natrénování klasifikačního modelu je možné určit, do které třídy patří data s neznámou třídou. To jsou tzv. testovací data. Testovací data se používají pro validaci modelu. Jsou to data, u kterých známe třídu, do které patří. Tuto třídu posléze porovnáme s predikovanou třídou a na základě těchto dat je možné určit úspěšnost natrénovaného modelu.

Trénovací i testovací data by měla být před vstupem do klasifikátoru určitým způsobem předzpracována. Toto předzpracování se soustředí na odstranění různých rušivých jevů a tím je možné zvýšit úspěšnost klasifikátoru.

Přesnost klasifikátoru jde zlepšovat správným nastavením vstupních parametrů.

3.2.1 Křížová validace

Křížová validace je proces, kdy jsou trénovací data rozdělena na několik částí. Počet částí, na které jsou rozdělena trénovací data představuje počet cyklů křížové validace. V každém cyklu se vezmou trénovací data, která jsou ochuzena o testovací data, která byla vyčleněna při rozdělení pro každý cyklus. Trénovací data se použijí pro natrénování modelu a poté se na testovacích datech provede validace. Tímto způsobem se provedou i ostatní cykly. Výslednou přesnost klasifikačního modelu určíme pomocí průměru úspěšnosti v každém kole.

V dalších sekcích se zaměřím na popis klasifikátorů, které použiji v případové studii, a jejich podporu v knihovně SMILE.

3.2.2 Multinomiální naivní Bayes

Multinomiální naivní Bayes je klasifikační metoda s učitelem, ve které nezáleží na vzájemném vztahu jednotlivých slov v dokumentu. Právě díky tomuto přístupu je naivní, z čehož pramení velká síla a jednoduchost. Trénování tohoto modelu probíhá tak, že se nejprve vypočítá, kolik dokumentů, je v které třídě. Poté se vypočítá s jakou pravděpodobností se vyskytují jednotlivá slova, která jsou obsažena v trénovacích datech, v dané třídě. Predikce potom probíhá tak, že se pro každou třídu v natrénovaném modelu vypočítá pravděpodobnost, s kterou predikovaná data patří do té či oné třídy. Výsledek s nejvyšší hodnotou je vybrán jako třída predikovaných dat. Výsledná třída dokumentu se vypočítá pomocí rovnice:

$$c_{map} = \underset{c \in C}{arg\ max}[log\ \widehat{P}(c) + \sum_{1 \leq k \leq n_d} log\ \widehat{P}(t_k|c)] \tag{3.1}$$

kde c_{map} je hledaná třída a index map je maximální hodnota posteriorní pravděpodobnosti. C je množina tříd. $\widehat{P}(c)$ je apriorní pravděpodobnost toho, kolik slov se vyskytuje v každé třídě c děleno počet slov ve slovníku (slova získaná z trénovacích dat). $P(t_k|c)$ je podmíněná pravděpodobnost toho, že se slovo t_k vyskytuje v dokumentech, které patří do třídy c. c je tedy třída, do které daná slova náleží. Počet těchto prvků reprezentuje n_d.

Pravděpodobnost $\widehat{P}(c)$ je vypočítána pomocí následující rovnice:

$$\widehat{P}(c) = \frac{N_c}{N}, \tag{3.2}$$

kde N_c je počet všech trénovacích dat, které patří do třídy C a N je počet všech tříd.

Pravděpodobnost $\widehat{P}(t_k|c)$ je vyjádřena počtem výskytů prvků t_k v trénovacích datech, které patří do třídy c. Pokud slovo t neexistuje v třídě c, tak nastane situace, kdy bude výsledná pravděpodobnost nulová, což znemožní jakoukoli predikci. Pro tyto účely se pro jakýkoli počet výskytů prvků pro danou třídu přidává implicitně číslo 1 (nebo jakékoli jiné číslo větší než nula). To je známo jako aditivní vyhlazování nebo též Laplaceovo vyhlazování.

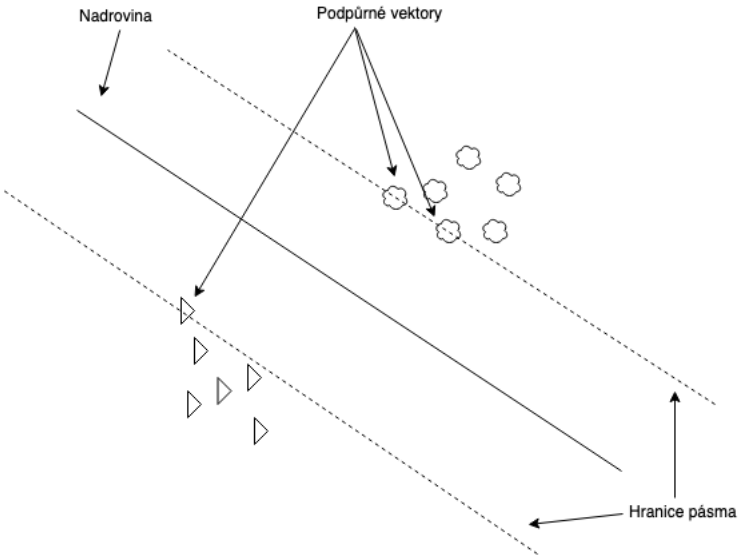
V knihovně SMILE je Multinomiální naivní Bayes dostupný jako třída NaiveBayes s konstruktorem `public NaiveBayes(Model model, int k, int p, double sigma)`, kde Model je výčet, který zastupuje klasifikační algoritmy. SMILE poskytuje implementace pro multionomiální, obecný, bernoulliho a polya urn model. Proměnná k představuje počet tříd, p představuje počet prvků použitých ve slovníku. Sigma je poté nenulové desetinné číslo větší než nula, které slouží k aditivnímu vyhlazování.

3.2.3 Support Vector Machines

Support Vector Machines nebo také SVM, je binární klasifikátor, jehož cílem je najít hranici, která je co možná nejvzdálenější od obou klasifikovaných tříd. Tato hranice má podobu nadroviny, kterou ve dvojrozměrném prostoru představuje přímka a ve trojrozměrném prostoru rovina. Tato rovina je znázorněna na obrázku 3.2 a tvoří ji dokumenty v podobě bodů $\vec{x}$, které splňují následující rovnici:

$$\vec{w}^T \vec{x} = -b, \tag{3.3}$$

kde b je konstanta a $\vec{w}$ je normálový vektor. Předpokládáme dále, že trénovací data



Obrázek 3.2: Rozdělení dat pomocí nadroviny.

$D = \{(\vec{x}_i, y_i)\}$, kde každý člen trénovacích dat je dvojice, kde první z dvojice je dokument $\vec{x}_i$ a druhý z dvojice je třída do které náleží y_i, a protože SVM je binární klasifikátor,

tak $y_i \in \{+1, -1\}$. Na obrázku 3.2 jsou také vidět podpůrné vektory a hranice tvrdého pásma. Tvrdé pásmo je prostor uvnitř hranic. Toto pásmo se nazývá tvrdé z toho důvodu, protože se v něm nesmí vyskytovat žádný bod x_i z trénovacích dat D. Pokud se v tomto pásmu vyskytne, tak je nutné posunout nadrovinu a spolu s ní i hranice pásma. Lineární klasifikátor získáme pomocí následující rovnice:

$$f(\vec{x}) = sign(\vec{w}\vec{x} + b). \tag{3.4}$$

Pro nalezení hodnot $\vec{w}$ a b je nezbytné aby:

- $\frac{1}{2}\vec{w}^T\vec{x}$ bylo minimalizováno, a
- pro všechna $\{(\vec{x}_i, y_i)\}, y_i(\vec{w}^T\vec{x}_i + b) \geq 1$,

pokud daná data nelze lineárně rozdělit na dvě části pomocí nadroviny ve dvojrozměrném prostoru, tak je možné přenést problém do trojrozměrného prostoru a pokusit se rozdělit data tam, popřípadě použít tzv. měkké pásmo, které dokáže tolerovat určité atypické případy, kdy se např. dokument s třídou 1 objevuje na opačné straně nadroviny než by bylo vhodné. Pro další informace o tomto problému viz [19].

Knihovna SMILE poskytuje implementaci SVM algoritmu s následujícím konstruktorem. `public SVM(MercerKernel<T> kernel, double Cp, double Cn)`, kde MercerKernel je interface (obdoba traitu v Javě), který má velké množství implementací mimo jiné i lineární. T je obecný typ vstupních dat. Cp je penalizace na pozitivní stranu za výskyt v opačném prostoru a Cn je penalizace na negativní stranu za výskyt v opačném prostoru.

3.2.4 AdaBoost

AdaBoost je zkratka pro slova adaptive boosting (adaptivní zesilování). Tento algoritmus využívá mnoho slabých prediktorů pro vytvoření modelu silného. Adaptivní zesilování se velmi často používá v kombinaci s rozhodovacími stromy o jednotkové výšce. AdaBoost byl vytvořen pro binární klasifikaci. Pro klasifikaci více než dvou tříd je nezbytné tyto třídy rozdělit do skupin po dvou. Tím, že je AdaBoost binární klasifikátor je zajištěno, že po průchodu každým rozhodovacím stromem bude výsledek buď -1 nebo 1 tzn., že bude vždy jasné, ke které ze dvou klasifikovaných tříd se slabý prediktor přiklání. Slabý prediktor je takový prediktor, který dokáže predikovat výslednou třídu pouze o trochu lépe než náhodný prediktor. Trénování modelu probíhá tak, že se nejprve inicializují váhy pro jednotlivé dokumenty ve vstupních datech na stejnou hodnotu (1/počet dokumentů). Poté je nutné najít takový slabý prediktor, který má nejmenší klasifikační chybovost na vstupních datech. Když je takový prediktor nalezen, tak dojde k vypočítání jeho váhy a následně se aktualizují váhy všech dokumentů v datech. Pokud byl dokument vyhodnocen správně, tak se jeho váha sníží a pokud byl vyhodnocený špatně, tak se jeho váha zvýší. Tím vyniknou dokumenty, které je obtížné klasifikovat a v dalším cyklu algoritmu se na ně bude brát větší zřetel. Na algoritmu 1 je formálně popsáno trénování modelu.

Na vstupu algoritmu je množina dvojic, kde x_i jsou samotná data náležící do domény $\mathcal{X}$ a y_i je třída daných dat, která náleží do množiny $\{-1, +1\}$. Na začátku se distribuce D_t, která značí váhy jednotlivých prvků, pro každý vstupní prvek inicializuje podle $D_1 = 1/t$, kde t je pořadí vstupního prvku v trénovací sadě. T je cílový počet slabých prediktorů, které tvoří tzv. silný prediktor. h_t je slabý prediktor jehož cílem je najít slabou predikci s co možná nejmenší váženou chybou ϵ vztahující se k distribuci D_t. α_t je váha daného slabého klasifikátoru. Po každém průchodu se tyto váhy upraví podle toho, zda byla predikce daného

Je dáno: $(x_1, y_1), ..., (x_m, y_m)$ kde $x_i \in \mathcal{X}, y_i \in \{-1, +1\}$
Inicializujeme $D_1(i) = 1/m$ pro $i = 1, \ldots, m$
Pro $t = 1, \ldots, T$:

- Natrénujeme slabý prediktor pomocí distribuce D_t.
- Získáme slabou predikci $h_t : \mathcal{X} \to \{-1, +1\}$.
- Zpřesnění: h_t s nejmenší váženou chybou:

$$\epsilon_t = Pr_{i \sim D_t}[h_t(x_i) \neq y_i]. \tag{3.5}$$

- Zvolíme: $\alpha_t = \frac{1}{2} ln(\frac{1-\epsilon_t}{\epsilon_t})$.
- Aktualizujeme pro $i = 1, \ldots, m$:

$$D_{t+1}(i) = \frac{D_t(i) exp(-\alpha_t y_i h_t(x_i))}{Z_t}, \tag{3.6}$$

- kde Z_t je normalizační faktor zvolený tak, aby D_{t+1} byla distribuce.

Výsledná funkce klasifikátoru:

$$H(x) = sign(\sum_{t=1}^{T} \alpha_t h_t(x)). \tag{3.7}$$

Algoritmus 1: Pseudokód pro AdaBoost [22]

slabého prediktoru správná či nikoli. Výsledná predikce $H(x)$ je dána volbou většiny. Tato volba je ovlivněna váhou α, kterou má každý slabý prediktor. Proces učení končí dosažením požadovaného počtu slabých prediktorů nebo pokud byla dosažena požadovaná přesnost.

SMILE poskytuje podporu pro klasifikační model AdaBoost pomocí třídy AdaBoost, která nabízí tento konstruktor `public AdaBoost(double[][] x, int[] y, int ntrees, int maxNodes)` , kde x jsou trénovací data, y jsou třídy přiřazené pro každá jednotlivá data. Parametr ntrees je cílový počet slabých prediktorů a maxNodes představuje počet listů, které bude mít výsledný slabý klasifikátor.

3.2.5 Metoda nejbližších sousedů

Metoda nejbližších sousedů nebo také k-NN je klasifikační metoda, která přiřadí klasifikovaný dokument té třídě, jejíchž nejvyšší počet k-členů, je ke klasifikovanému dokumentu nejblíže. Výběr správného čísla k je důležité, protože pokud zvolíme např. číslo 1, může dojít k tomu, že klasifikovaný dokument bude špatně zařazen z důvodu výskytu určitého atypického dokumentu, tedy v jeho nejbližší blízkosti. Proto je vhodné volit takové k, které je větší než 1. Dále je vhodné volit k liché, aby nedocházelo k situacím, kdy má vstupní dokument stejný počet nejbližších sousedů z různých tříd. Zvolení příliš velkého k také není vhodné, protože poté může dojít k zastínění třídy, která má malé množství dokumentů a tudíž bude zcela vyřazena z klasifikace. Nejčastěji jsou vybírána čísla 3, 5 nebo 7, ale výjimkou nejsou ani čísla mezi 50 a 100.

SMILE pro klasifikační algoritmus k nejbližších sousedů poskytuje následující statickou metodu třídy KNN, která daný model vytvoří. `public static KNN<double[]> learn(double[][]`

`x, int[] y, int k)`, kde x představuje množinu trénovacích dat, y obsahuje třídy, do kterých data v parametr x náleží a k je číslo, které určuje počet nejbližších sousedů. Pro výpočet vzdálenosti mezi jednotlivými sousedy je použitý algoritmus pro výpočet euklidovské vzdálenosti.

3.3 Analýza problému

Jako data jsem vybral dataset[2] obsahující 75 419 e-mailů, z toho 25 220 je označeno jako běžná komunikace a 50 199 e-mailů je označeno jako spam. E-maily jsou umístěny v adresáři *data* (lze nalézt v přiloženém datovém médiu), kde každý e-mail je v samostatném souboru. Dalším adresářem je adresář *full*, ve kterém je soubor *index*, který obsahuje množinu dvojic. První z dvojice je typ daného e-mailu, kde nevyžádaný e-mail je označený jako spam a e-mail z běžné komunikace jako ham. Druhý z dvojice představuje cestu k souboru.

3.3.1 Formát e-mailu

E-maily, které jsou obsaženy ve vybrané kolekci dat, mají různý formát. Některé z nich jsou v nekonzistentním stavu, s čímž si bude muset umět demonstrační aplikace poradit. E-maily tudíž nemusejí být vždy ve správném formátu. Pro načtení e-mailu do paměti programu jsem zvolil nástroj vytvořený nad knihovnou *mime4j* a to *email-mime-parser*, který umožňuje pracovat s e-maily, které splňují jak formát definovany v RFC 822, tak i pozdější revize RFC 2822, RFC 5322 a další.

3.3.2 Řídkost dat

Lidé, kteří tvoří spam zprávy se různými způsoby snaží dostat skrz stále lepší filtry, které se spam snaží rozeznávat a filtrovat je. Jedním ze způsobů, jak takový filtr oklamat je rozdělit slovo na jednotlivá písmena např. slovo "viagra"rozdělíme na "vi a g r a". Tím se takové slovo stane pro filtr neviditelným, protože jakmile bude profiltrováno přes stop slova, tak buď zcela zmizí nebo z něj zůstane pouze část, která ztratila svou hodnotu. Tento problém budu řešit vlastními silami a to tak, že vytvořím metodu na spojení těchto znaků. Počet znaků, které budou spojovány bude možné konfigurovat z konfiguračního souboru.

3.3.3 Pravopisné chyby

Výskyt pravopisných chyb v textu je velký problém, protože pro počítač jsou slova "goad"a "goat"naprosto odlišné, kdežto člověk dokáže snadno rozpoznat správné slovo i s překlepem. Proto je nutné taková slova opravit a provést to pokud možno co nejefektivněji. Pro tento problém naštěstí existují velmi dobrá řešení. Jedním z nich je knihovna SymSpell původně napsaná v programovacím jazyce C#, ale existují i řešení pro Javu a tím pádem i pro Scalu.

3.3.4 HTML

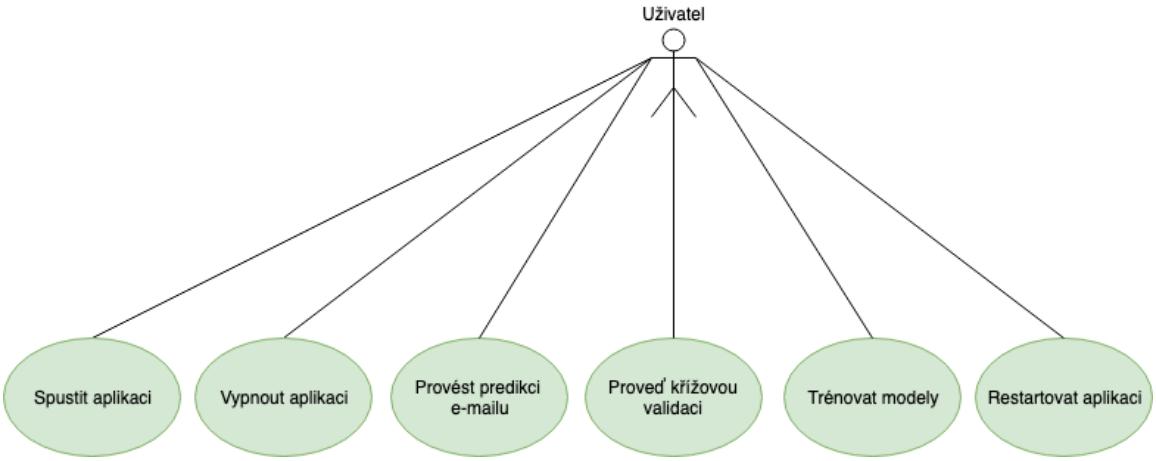
Mnoho e-mailů je dnes napsáno pomocí HTML (Hypertext Markup Language). V HTML se používá velké množství značek, které musejí být před samotným trénováním odfiltrovány. Na počátku jsem myslel, že použiji značky jako metadata, které mi pomohou v rozlišení správné kategorie. Cílem bylo získat HTML značky společně s tím, jak dlouhé slovo obalovaly. Např.

[2] 2007 TREC Public Spam Corpus: https://plg.uwaterloo.ca/~gvcormac/treccorpus07/

pokud by některý e-mail obsahoval 20 HTML *span* značek, které obalovaly vždy slovo o délce 1, tak by bylo celkem jisté, že se jedná o spam. Od tohoto záměru jsem nakonec upustil, protože jsem nenašel dostatek zpráv, ve kterých by bylo HTML použito takovým způsobem.

3.4 Požadavky na systém

Výsledný systém bude umožňovat všechny operace, které jsou popsány na obrázku 3.3 use case diagramem.



Obrázek 3.3: Use case diagram popisující použití aplikace.

V systému bude možné:

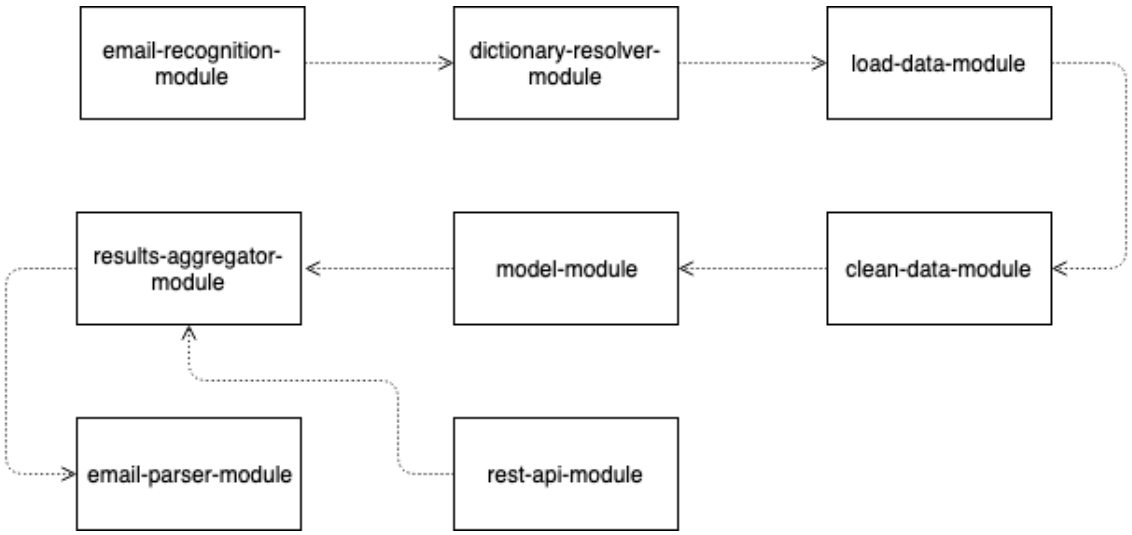
- stustit aplikaci tj. připravit ji na trénování a predikci.
- vypnout aplikaci.
- predikovat e-mail na natrénovaných modelech.
- provádět křížovou validaci modelů.
- natrénovat modely.
- restartovat aplikaci a uvést ji do původního stavu.

3.5 Návrh systému

Celý systém bude postavený na modelu aktérů implementovaném ve frameworku Akka s tím, že bude možné tento systém spouštět jako cluster (seskupení 2 a více počítačů, které jsou provázány a pracují spolu na stejném úkolu). Pro tuto vlastnost bude potřeba použít balíček z Akka nazvaný akka-cluster, který obsahuje podporu pro spuštění aplikace v clusteru. Předtím, než jsem se rozhodl pro řešení, které se spoléhá pouze na Akku, jsem plánoval použít injektor závislostí Guice od společnosti Google, ale nakonec jsem zjistil, že nebude potřeba a všechny problémy se závislostmi bude možné vyřešit pomocí Akky a sbt, což je nástroj určený pro zjednodušení kompilace zdrojových souborů napsaných ve Scale.

3.5.1 Logická struktura

Systém jsem rozdělil do osmi modulů. Load-data-module bude zodpovědný za načtení dat. Clean-data-module bude zodpovědný za čištění dat a jejich převedení do takového tvaru, který bude vhodný pro trénování a predikci. Model-module se bude zabývat trénováním modelů a následnou predikcí. Results-aggregator se bude starat o sběr informací tzn., že např. při křížové validaci bude sbírat výsledky z daného cyklu a po dokončení tohoto cyklu je zapíše do nakonfigurovaného adresáře. Email-recognition-module se bude starat o samotnou orchestraci a řízení celého systému. Rest-api-module bude poskytovat REST rozhraní pro ovládání celého systému. Bude také obsahovat metodu, pomocí které bude možné vyhodnocovat, zda daný e-mail je spam, či nikoli. Dictionary-resolver-module se bude starat o načtení nebo vytvoření slovníku, který bude následně použitý. Email-parser-module bude obsahovat metody nezbytné pro načtení e-mailu do paměti. Na obrázku 3.4 jsou zobrazeny jednotlivé moduly a jejich vzájemná závislost.



Obrázek 3.4: Graf závislostí modulů.

3.5.2 Konfigurace

Pro konfugraci jsem použil knihovnu *config*[3] od společnosti *Lightbend* založená samotným tvůrcem Scaly Martinem Oderskym.

Tato knihovna umožňuje definovat specifickou konfiguraci pro jednotlivé moduly ve formátu HOCON (Human-Optimized Config Object Notation to v překladu přibližně znamená Konfigurační popis optimalizovaný pro lidi). Této možnosti jsem využil, a proto má každý modul definovánu vlastní konfiguraci. V každém modulu budou popsány i možnosti konfigurace. Konfigurace *Akky* zde nebudu uvádět. Tyto možnosti mohou být dohledány na dokumentačních stránkách [4].

[3]Odkaz na knihovnu config: https://github.com/lightbend/config
[4]Akka - dokumentace: https://akka.io/docs/

3.5.3 Rozhraní systému

Pro celkové ovládání systému slouží RESTové rozhraní a to proto, že se bude jednat o systém v clusteru. Aplikace je navíc koncipována tak, aby bylo možné ji nechat puštěnou neomezeně dlouhou dobu, takže je nezbytné, aby se její chování dalo průběžně upravovat. REST rozhraní je navíc vhodné pro správnou propagaci příkazů a navíc je dostupné na všech uzlech v daném clusteru. Před popisem návrhu rozhraní bude stručně popsáno a vysvětleno RESTové rozhraní.

REST

Informace pro tuto část byla čerpána z tohoto zdroje [5].

REST je zkratka vytvořená ze slov Representational State Transfer a je to typ softwarové architektury, která je dnes velmi hojně používána dohromady s *HTTP* protokolem. *REST* představuje šest restrikcí, kdy v případě dodržení všech z nich hovoříme o tzv. RESTful rozhraní.*REST* používá metody definované protokolem *HTTP* (GET, POST, PUT atd.). Omezení jsou následující:

Klient-server architektura představuje první ze šesti omezení definovaných v RESTu. Idea tohoto omezení je rozdělení zodpovědností mezi klientem (tj. ten, který žádá) a serverem (tj. ten, který poskytuje služby nebo data). Tím je zajištěna škálovatelnost, jednotnost rozhraní a zjednodušení serverové části.

Bezstavost je druhé omezení. Toto omezení říká, že server si neudržuje žádný kontext o provedených dotazech. Tím je zajištěno, že každý dotaz odeslaný na server obsahuje všechny informace potřebné k jeho vyřízení. Tento kontext a data obsažená v dotazu, ale mohou být držena v jiné službě, typicky v databázi.

Kešovatelnost je schopnost držet si na straně klienta a to po určitou dobu odpovědi serveru v paměti. Každá taková odpověď musí obsahovat informaci o tom, zda je možné ji uložit. Tím je dosaženo toho, že některé dotazy na server nemusejí vůbec proběhnout.

Vrstvený systém představuje stav, kdy klient neví, zda server, kterého se dotazuje pouze jeden nebo se tento server dotazuje dalších služeb.

Kód na vyžádání je nepovinné omezení, které představuje možnosti, kdy server může rozšířit nebo upravit funkcionalitu klienta tím, že v odpovědi předá spustitelný kód napsaný např. v JavaScriptu.

Jednotné rozhraní je nejdůležitější omezení pro implementaci skutečného RESTful rozhraní. Toto omezení zjednodušuje architekturu a to takovým způsobem, kdy se mohou jednotlivé části systému vyvíjet zcela nezávisle na sobě. Toho je docíleno čtyřmi omezeními, která jsou následující:

- *Identifikace zdrojů v dotazech* představuje část *URL*, která reprezentuje určitý zdroj. Např. "http://localhost:8080/accounts/idAccount/cards"je *URL*, kde jsou zdroje slova *accounts* a *cards*. Každý tento zdroj definuje, které dotazy je možné provést. Samotná reprezentace dat v odpovědi nezávisí na tom, jak jsou daná data uložená. To tedy

znamená, že odpověď může být v jakémkoli formátu, který je definovaný v hlavičce *HTTP* zprávy.

- *Manipulace se zdroji* říká, že pokud klient drží reprezentaci daného zdroje včetně všech přiložených metadat, tak má dostatek informací k tomu, aby mohl daný zdroj upravit nebo smazat.
- *Sebe popisné zprávy* je další omezení, které jednotné rozhraní definuje a souvisí s tím, že každá zpráva obsahuje dostatek informací k tomu, aby klient věděl, jakým způsobem danou zprávu zpracovat.
- *Hypermédia jako aplikační stav* nebo také *HATEOAS* umožňuje klientovi získat veškeré informace o rozhraní pouze na základě znalosti kořenového zdroje. Při dotazu na tento zdroj dostane klient veškeré akce, které jsou dostupné v podobě *URL*.

Na obrázku 3.5 jsou popsány jednotlivé REST metody, které slouží pro samotné ovládání aplikace. Všechny metody se zdrojem controls slouží k ovládání aplikace. Metoda se zdrojem predictions, slouží pro predikci e-mailu poskytnutého v těle volání. Tento e-mail musí být kódován v BASE64. Metodu pro predikci je možné použít poté, co byla použita metoda !train-models.

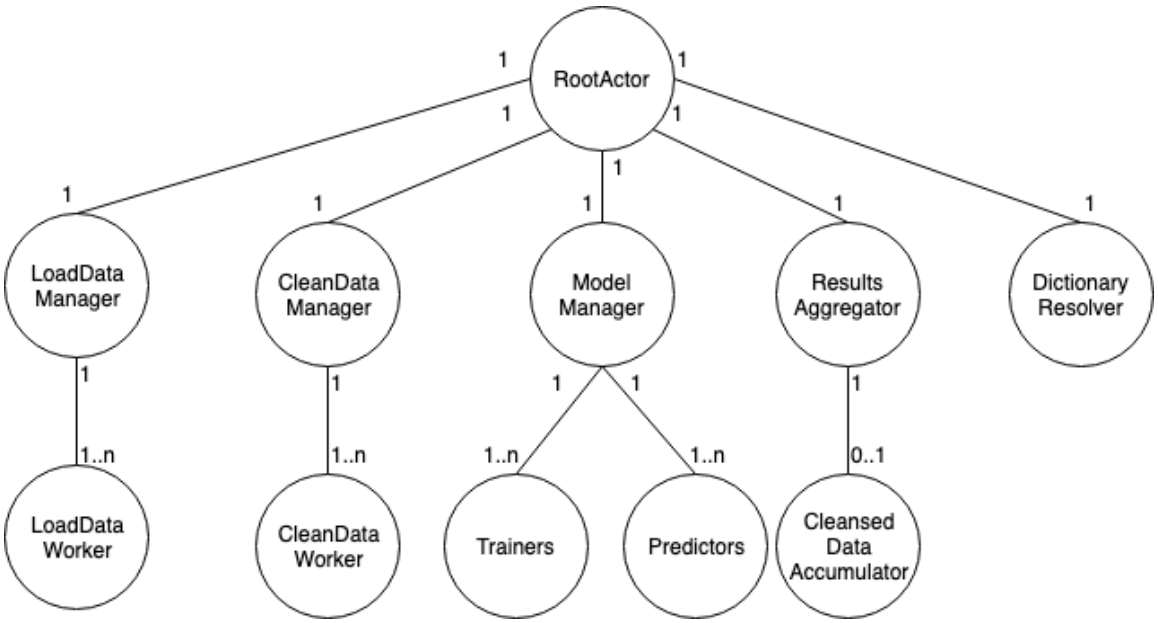
REST rozhraní	
POST	/controls/!start • po provolání této metody dojde k získání slovníku • poté, co je slovník získán, je aplikace připravena k použití
	/controls/!terminate • po provolání této metody dojde k vypnutí uzlu v clusteru
	/controls/!restart • po provolání této metody dojde k restartování všech aktérů, což je uvede do stavu, který nastane po provolání metody !start
	/controls/!train-models • tato metoda spustí process, který natrénuje modely pomocí trénovacích dat, které má aplikace k dispozici
	/controls/!cross-validation • tato metoda spustí process křížové validace
	/predictions/!predict • tato metoda umožňuje určit, zda poskytnutý e-mail patří do vyžádané či nevyžádané pošty • samotný e-mail musí být kódovaný v BASE64

Obrázek 3.5: REST rozhraní aplikace.

3.5.4 Hierarchie aktérů

Akka umožňuje vytvářet hierarchie aktérů viz. obr. 3.6 a já tuto vlastnost využiji pro delegaci práce, kdy pro každou část vytvořím aktéra, který bude tzv. manažerem a bude mít zodpovědnost za např. načtení dat. Takový manažer může mít 1 až N pracovníků, kterým bude delegovat práci. Tito pracovníci mohou být rozmístěni na více počítačů a přitom využít plný potenciál vícejádrového procesoru. Aktér RootActor je situován v modulu

email-recognition-module. Aktér LoadDataManager a všichni jemu podřízení herci patří do modulu load-data-module. Aktér CleanDataManager a jemu podřízení aktéři tvoří modul clean-data-module. Model-module obsahuje aktéra ModelManager a jemu podřízené aktéry. ResultsAggregator náleží do modulu results-aggregator-module a DictionaryResolver patří do dictionary-resolver-module.



Obrázek 3.6: Hierarchie aktérů.

3.6 Implementace REST rozhraní

V této sekci popíši reakci systému a jednotlivých aktérů na všechny metody, které jsou zobrazeny na obrázku 3.5. Na jednotlivých obrázcích komunikace aktérů jsou zobrazeny pouze nejdůležitější zprávy. Celkový přehled zpráv a další detaily jsou popsané v kapitole 3.7.

3.6.1 Controls

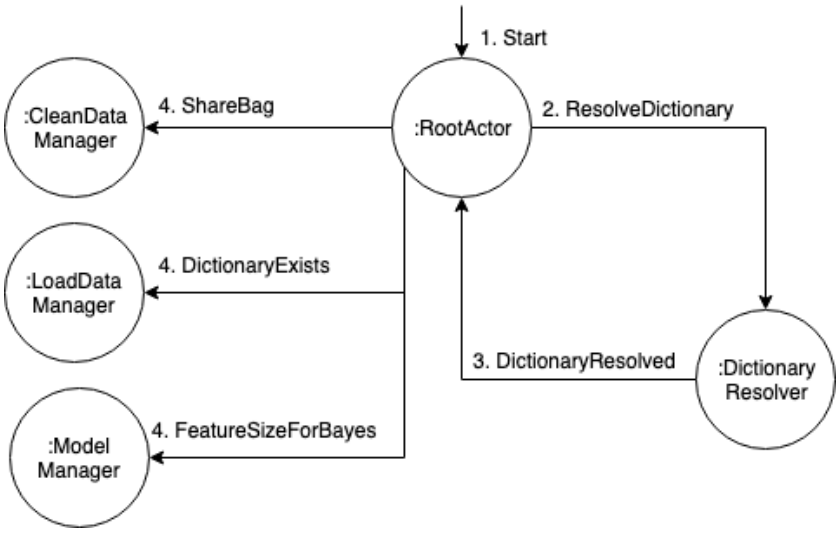
V této části bude popsáno chování systému po zavolání metod, určených k ovládání aplikace.

!start

Před použitím této metody se očekává, že jsou všechny uzly v clusteru navzájem spojeny. Pokud toto nebude dodrženo a některý uzel se připojí až po zavolání metody start, tak je nezbytné provést restart aplikace.

Na obrázku 3.7 je stručný popis komunikace aktérů, při připravování aplikace k provozu. Na obrázku je aplikace ve stavu, kdy zpráva o startu aplikace dorazila do aktéra RootActor.

1. Start zpráva pochází z REST rozhraní.
2. Druhá zpráva způsobí, že se aktér DictionaryResolver pokusí získat slovník, ze kterého vytvoří instanci třídy `public class Bag<String>`. Tato třída slouží k převodu



Obrázek 3.7: Stručný popis komunikace jednotlivých aktérů při startu.

slov do číselné reprezentace. Získání slovníku probíhá tak, že aktér nejprve zjistí, zda je nějaký slovník nakonfigurován, pokud zjistí, že takový slovník existuje, tak z něj vezme nakonfigurovaný počet slov a vytvoří instanci třídy Bag. Pokud není nakonfigurovaný žádný slovník, tak se začnou vytvářet slovníky ze všech poskytnutých dat. Jeden pro vyžádanou poštu a druhý pro nevyžádanou. Tyto slovníky jsou uloženo do nakonfigurovaného adresáře. Poté se z nich vytvoří instance třídy Bag.

3. Tato zpráva obsahuje serializovanou instanci třídy Bag společně s celkovým počtem použitých slov.

4. Tyto zprávy slouží k uvedení všech aktérů do stavu, kdy jsou připraveni k práci. Tzn., aktéři pro čištění dat jsou připraveni pro převod textu na číselnou reprezentaci, LoadDataManager je uvědoměn o tom, že má změnit své chování, aby mohl reagovat na další příkazy a ModelManager po přijetí zprávy vytvoří aktéry pro trénování modelů a aktéry pro predikci. Tato zpráva je nezbytná pouze pro klasifikátor naivního Bayese.

!terminate

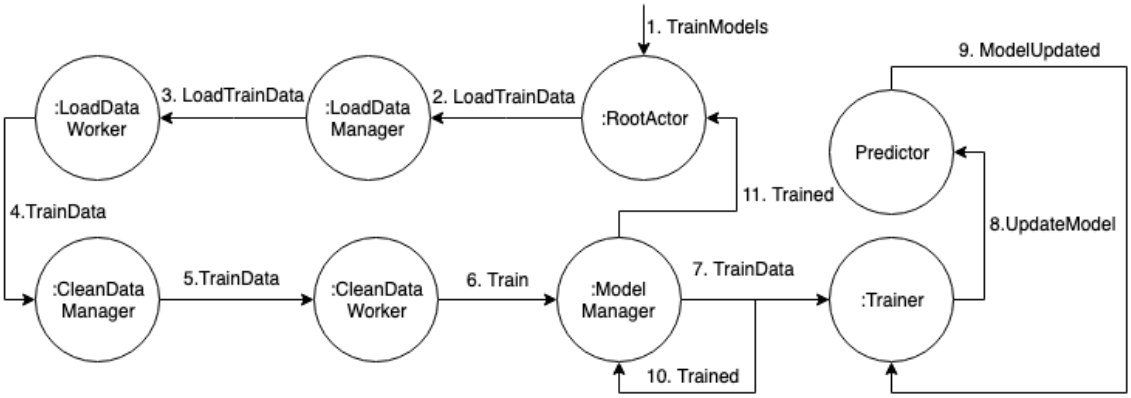
Tato zpráva odpojí uzel, na který byla zaslána, z clusteru. V závislosti na konfiguraci minimálního počtu uzlů může dojít k zastavení celé aplikace a to do té doby, než bude tento počet zase dostupný.

!restart

Tato zpráva spustí restartování všech aktérů. Restartování hierarchie aktérů je zjednodušené samotným frameworkem Akka, protože dojde-li k restartování aktéra rodiče, tak dojde k restartování všech jeho potomků. Aktér RootActor tedy po přijetí zprávy přikazující restart odešle zprávu, která přikazuje restartování všem jeho potomkům.

!train-models

Proces trénování modelů je zobrazen na obrázku 3.8. V tomto případě je jako trénovací data použitý pouze jeden e-mail. V případě použití více trénovacích dat je zpráv s číslem 3, 4, 5 a 6. tolik, kolik je trénovacích dat. 7. zpráva poté obsahuje všechna naakumulovaná předzpracovaná data, ze kterých se bude klasifikační model učit. Takto natrénovaný model je serializován a odeslán všem prediktorům. Poté je odeslána zpráva Trained aktérovi ModelManager (pokud je použitý více než jeden klasifikační model, tak aktér ModelManager čeká na natrénování všech modelů) a ten tuto zprávu propaguje aktérovi RootActor. Poté je možné začít predikovat e-maily poskytnuté přes REST rozhraní.



Obrázek 3.8: Trénování modelů.

!cross-validation

Křížová validace využívá jak procesu učení, který je zobrazen na obrázku 3.8, tak predikce, který je zobrazen na obrázku 3.9. V případě predikce je rozdíl v tom, že data pro predikci nepocházejí při křížové validaci z RESTového rozhraní, ale z nakonfigurovaných dat a výsledek se odesílá i aktérovi RestulsAggregator, který po každém cyklu zapíše výsledky do nakonfigurované složky. Po dokončení všech kol křížové validace je poté vytvořen i graf s jak správně klasifikovanými e-maily, tak také se špatně klasifikovanými pro každý model v každém kole.

3.6.2 predictions

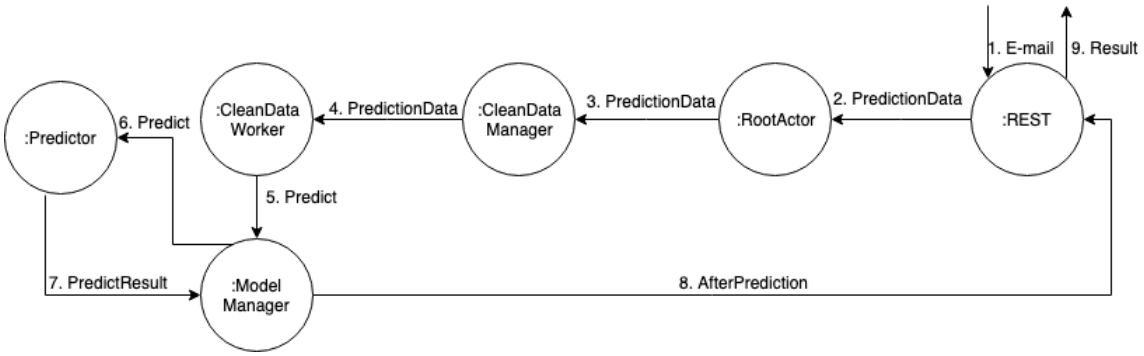
Tento zdroj obsahuje pouze jednu metodu a tou je metoda !predict.

!predict

Na obrázku 3.9 je zobrazen průchod zprávy z RESTového rozhraní. Obsah této zprávy je e-mail, kterému natrénované modely určí třídu, do které s nejvyšší pravděpodobností patří. Aplikace musí mít natrénované modely, jinak takový dotaz skončí s chybovou hláškou.

1. Zpráva v JSON formátu, která obsahuje e-mail zakódovaný v BASE64. Po přijetí je tento e-mail dekódován.

2. Aktér RootActor tuto zprávu pouze předá dál společně s referencí na odesílatele.



Obrázek 3.9: Průchod zprávy určené pro predikci.

3. Aktér CleanDataManager deleguje zprávu stejným způsobem jako aktér RootActor a to na jednoho z aktérů typu CleanDataWorker.

4. CleanDataWorker provede předzpracování e-mailu a odešle ho, společně s referencí na původního odesílatele aktérovi ModelManager.

5. ModelManager deleguje přijatou zprávu na všechnu dostupné aktéry určené pro predikci (na obrázku je zobrazen pouze jeden, ale může jich být více v závislosti na použitých modelech).

6. Prediktor použije natrénovaný model a pomocí něj predikuje třídu e-mailu.

7. Po obdržení predikce odešle výsledek původnímu odesílateli.

8. Tato zpráva je již pouze převedena do JSON formátu.

3.7 Moduly systému

V této sekci budou popsány jednotlivé moduly a to z pohledu implementace jednotlivých tříd, tak z pohledu jejich konfigurace pomocí konfiguračního souboru.

Zpráva je vždy reprezentována instancí datové třídy. V následujícím textu budou tyto pojmy volně zaměňovány.

3.7.1 email-recognition-module

Toto je kořenový modul, který obsahuje metodu *main(args: Array[String])*, která slouží ke spuštění aplikace. Dále obsahuje třídu aktéra RootActor.

Třídy

RootActor

RootActor je aktér, který se stará o vytvoření všech aktérů, kteří jsou v hierarchii 3.6 pod ním. Stará se také o distribuci důležitých informací a dat skrze systém. Tento aktér se stará o přijímání a přeposílání zpráv, které přichází z REST rozhraní. Tento aktér je také zodpovědný za správnou konzistenci aplikace.

Tento aktér je definován následujími stavy.

- *ApplicationStart* představuje chování aktéra, které slouží k přípravě všech aktérů do použitelného stavu. Přijímá tyto zprávy:
  - StartApplication je typ zprávy, která spustí proces pro získání slovníku.
  - DictionaryResolved je zpráva přijatá po získání slovníku, která obsahuje jak třídu Bag, tak celkový počet použitých slov. Tato data jsou distribuována na podřízené aktéry a následně se aktér RootActor přepne do stavu *waitingForOrders*.
- *waitingForOrders* je chování, kdy je systém již připravený na trénování modelů a křížovou validaci a je možné systém dále ovládat přes RESTové rozhraní. Podporuje příjem těchto zpráv:
  - RestartActors je příkaz způsobí kompletní vypnutí a zapnutí všech aktérů, kteří jsou na nižší pozici v hierarchii než-li RootActor. Tento příkaz způsobí ztrátu natrénovaných modelů a všech vnitřních stavů jednotlivých aktérů.
  - TrainModel je příkaz, který spustí trénování modelů. Z pohledu aktéra RootActor to znamená přepnout své chování na *trainModels* a poslat sobě samému zprávu, která spustí samotné trénování.

    V průběhu trénování modelů není možné jakkoli ovládat běh aplikace.
  - StartCrossValidation představuje zprávu, kdy po jejím přijetí aktér RootActor přepne své chování na chování *crossValidation* a následně si pošle zprávu, která zaktivuje samotný proces křížové validace. Po dokončení křížové validace se aktér přepne do chování *waitingForOrders*.
- *crossValidation* je chování, které se stará o správný průběh křížové validace.
  - StartCrossValidation spustí křížovou validaci a připraví každého aktéra pro tento proces.
  - LastPredictionMade představuje zprávu, která ukončuje jednotlivé cykly křížové validace.
  - Trained je zpráva od aktéra ModelManager, která spustí predikční část křížové validace.
  - CrossValidationDone je zpráva, která říká, že je proces křížové validace zcela dokončen. Aktér ModelManager je následně restartován.
- *trainModels* je chování, které slouží k řízení procesu trénování modelů a definuje tyto zprávy.
  - TrainModel je zpráva, která spustí trénování modelů. Aktérovi LoadDataManager je odeslána zpráva LoadTrainData.
  - Trained je zpráva, která říká, že jsou všechny modely natrénovány a připraveny k predikci,

3.7.2 load-data-module

Tento modul je zodpovědný za nahrání e-mailů z datasetu do paměti. Je označený rolí load-data. Při spuštění v clusteru musí být vybrán právě jeden uzel jako load-data. Na tomto uzlu musí být přítomna testovací data.

Třídy

LoadDataManager

Tento aktér se stará o instanciaci aktérů, kteří načítají e-maily do paměti, a disponuje čtyřmi stavy chování. *waitingForOrders*, *creationOfDictionary*, *crossValidation* a *loadTrainData*.

- *creationOfDictionary* je základní chování aktéra po instanciaci a umožňuje vykonat tyto příkazy.
  - DictionaryExists říká aktérovi, že slovníkový korpus je již vytvořený. Po přijetí této zprávy přejde aktér do stavu *waitingForOrders*
  - CreateDictionaryFromData přikazuje aktérovi získat všechny soubory, které jsou k dispozici v poskytnutých e-mailech a rozeslat je svým podřízeným aktérům ke zpracování. Jakmile je aktér upozorněn na existenci slovníku, změní své chování na *waitingForOrders* a čeká na příkazy od aktéra RootActor.
- *waitingForOrders* chování podporuje tyto příkazy:
  - Done restartuje aktéra a převede ho do základního stavu chování tj. *creationOfDictionary*.
  - LoadTrainData načte data určená pro natrénování modelů, přičemž se snaží vždy zachovat vyváženost trénovací sady. Množství dat je možné regulovat na základě konfigurace. Více informací o konfiguraci lze nalézt v sekci věnující se konfiguraci, která je dále v této kapitol.
  - StartCrossValidation změní aktérovo chování na *crossValidation* a následně si aktér LoadDataManager pošle zprávu stejného typu, která zahájí křížovou validaci.
- *crossValidation* umožňuje přijímat tyto tři typy zpráv:
  - StartCrossValidation zahájí křížovou validaci a načte množinu e-mailů specifikovanou v konfiguraci pro trénování modelů. Tuto množinu rozdělí do částí, jejichž počet je také konfigurovatelný. Pro každý cyklus vyhradí data pro trénování a data pro validaci. Po posledním cyklu ohlásí aktérovi RootActor, že křížová validace byla dokončena.
  - ContinueCrossValidation přikazuje aktérovi, aby načetl testovací data a rozeslal je ke zpracování.
  - Done restartuje aktéra.

LoadDataWorker

Tento aktér je podřízený aktérovi LoadDataManager a definuje pouze jedno chování. Toto chování akceptuje tyto zprávy.

- LoadDataForDictionary je zpráva obsahující instanci třídy File a objekt typu EmailType. Objekt typu File zastupuje soubor, který reprezentuje umístění e-mailu a EmailType typ daného e-mailu.

  Po přijetí této zprávy použije aktér objekt EmailParser a jeho metodu `def parse(file: File): Email`, která vytvoří objekt datové třídy Email, která obsahuje typ e-mailu

(spam/ham), identifikační číslo a kolekci jednotlivých částí v podobě datové třídy BodyParts. Datová třída BodyParts v sobě definuje atribut type, který nese informaci o typu daného e-mailu (zda je typu HTML či zda je to pouhý text) a poté samotné tělo dané části. Po načtení e-mailu je odeslán k dalšímu zpracování aktérovi CleanDataManager v podobě datové třídy CleanDataForDictionary. Reference na odesílatele této zprávy je předána dále. Tím je zajištěno, že všechny zpracované e-maily dostane zpět aktér CleansedDataAccumulator.

- LoadTrainData je zpráva, která obsahuje stejné atributy jako datová třída LoadDataForDictionary. Aktér se chová téměř stejně jako v případě zprávy LoadDataForDictionary s tím rozdílem, že nepředává dále odesílatele této zprávy. Další rozdíl je v tom, že načtená data jsou odeslána v datové třídě TrainData.

- LoadPredictionData je zpráva, která má stejné atributy jako třída popsaná výše. Chování aktéra po přijetí této zprávy je také stejné, pouze se změní třída ve které je obalený výsledek. Výsledná zpráva poslaná aktérovi CleanDataManager je typu PredictionData.

Konfigurace

Konfiguraci tohoto modulu obsahuje konfigurační objekt *load-data*, který obsahuje následující konfigurační možnosti:

- *number-of-workers* umožňuje nastavit počet aktérů, kteří slouží pro samotné nahrání dat do paměti. Tento počet určuje množství aktérů na jeden uzel v clusteru.

- *data* je adresář ve kterém se nacházejí soubory obsahující e-maily. Současná implementace předpokládá, že jeden soubor obsahuje právě jeden e-mail.

- *labels* je soubor ve kterém jsou obsaženy dvojice, kde první člen obsahuje slovo určující typ daného e-mailu (ham nebo spam) a druhý člen dvojice představuje relativní cestu k souboru obsahující daný e-mail. Jako oddělovač těchto dvojic je použita mezera. Tyto dvojice jsou od sebe odděleny znakem nového řádku.

- *split-to* definuje počet částí, do kterých budou rozdělena data při křížové validaci. Je vyžadováno přirozené číslo.

- *train-size* představuje procentuální velikost dat použitých pro natrénování modelů při použití RESTové metody !train-models.

- *from-each-group* představuje počet e-mailů, z každé skupiny, použitých pro křížovou validaci. Tato konfigurace je zde pouze pro případ, kdy není možné z kapacitních důvodů použít celou sadu dat. V případě, že toto nastavení nebude přítomno, budou použita všechna data.

Modul load-data-module definuje vlastní dispatcher (synchronizační prvek určený pro přiřazování vláken aktérům) určený pouze pro aktéry typu LoadDataWorker. Výchozí počet vláken toho dispatcheru je nastaven na dvě vlákna pro daný uzel v clusteru.

3.7.3 clean-data-module

Tento modul se stará o předzpracování načtených dat.

Třídy

CleanDataManager

Tato třída implementuje chování aktéra, který je zodpovědný za řízení čištění dat a definuje dva typu chování a to: *withDictionary* a *withoutDictionary*. Delegování zpráv na potomky zohledňuje velikost schránky jednotlivých potomků. Zpráva je tedy delegována tomu, kdo má nejmenší počet zpráv ke zpracování.

- *withoutDictionary* je výchozí chování a umožňuje přijímat následující typy zpráv.
  - CleanDataForDictionary je zpráva obsahující nezpracovaný e-mail. Tato zpráva je ihned po přijetí přeposlána jednomu z aktérů CleanDataWorker pro zpracování.
  - ShareBag je instance datové třídy, která v sobě obsahuje serializovanou třídu Bag. Okamžitě po přijetí se tato zpráva rozešle všem instancím třídy CleanDataWorker a aktér se přepne do stavu *withDictionary*.
  - Done zpráva restartuje aktéra a všechny jeho potomky.
- *withDictionary* definuje chování aktéra, kdy každý podřízený aktér typu CleanDataWorker má k dispozici třídu Bag. Je možné přijímat tyto zprávy.
  - PredictionData je instance datové třídy obsahující nezpracovaný e-mail. Tato zpráva je delegována na jednoho z aktérů typu CleanDataWorker.
  - TrainData je zpráva obsahující předzpracovaný e-mail určený k natrénování modelů. Tato zpráva je stejně jako předešlá ihned přeposlána jednomu z podřízených aktérů ke zpracování.
  - Done restartuje aktéra a všechny jeho potomky.

CleanDataWorker

Tato třída implementuje aktéra, který se stará o samotné předzpracování dat pro klasifikační modely. Tento aktér obsahuje dva typy chování a to *withBagOfWords* a *withoutBagOfWords*.

- *withoutBagOfWords* je chování, které umožňuje přijmout pouze zprávy CleanDataForDictionary a ShareBag. CleanDataForDictionary slouží pro předzpracování textu pro slovník vytvořený z e-mailů. Proto projde pouze předzpracováním textu, které neobsahuje převod do bag of words reprezentace a následně se odešle aktérovi *CleansedDataAccumulator*. Zpráva ShareBag v sobě obsahuje serializovanou instanci třídy Bag. Po přijetí této zprávy se aktér přepne do *withBagOfWords*.
- *withBagOfWords* chování podporuje příjem zprávy TrainData a PredictionData. V obou těchto případech se provede předzpracování textu. Liší se pouze typ odchozí zprávy pro aktéra ModelManager. V případě TrainData je odchozí zpráva typu Train a pro PredictionData je to zpráva typu Predict.

Konfigurace

Tento modul definuje konfigurační objekt *clean-data*, který obsahuje následující možnosti.

- *stemmer* určuje, který ze dvou stemmerů, porter nebo lancaster, bude použitý.

- *stop-words* je nastavení, které určuje, která množina stop slov bude použita. I zde je na výběr z více možností. Výchozí nastavení používá výchozí (default) stop slova definovaná knihovnou SMILE. SMILE obsahuje tyto množiny stop slov: *comprehensive, google, mysql* a *default*[5]. Je možné použít vlastní stop slova. Použitá slova musejí být oddělena čárkou.

- *number-of-workers* představuje počet aktérů LoadDataWorker spuštěných v jednom uzlu.

- *symspell-dictionary* představuje cestu ke slovníku, nezbytného pro opravu slov v knihovně SymSpell. Tento slovník je obsažen v samotné aplikaci a byl převzat z oficiální stránek knihovny SymSpell.

- *concatenate-chars* představuje maximální počet samostatně stojících znaků, které budou spojeny do jednoho slova. Očekávají se pouze přirozená čísla. Pro více informací viz kapitolu 3.3.2.

I tento modul používá vlastní dispatcher určený pouze pro aktéry typu CleanDataWorker. Ve výchozím nastavení má tento dispatcher k dispozici 4 vlákna. Pro použití více vláken je nezbytné redefinovat *clean-dispatcher*. Tento modul je ze všech ostatních modulů nejvytíženější z pohledu výpočetního výkonu, proto je vhodné přiřadit mu pokud možno co nejvyšší počet vláken.

3.7.4 model-module

Tento modul slouží pro samotné trénování modelů a pro predikci e-mailů.

Třídy

ModelManager

ModelManager je aktér, který se stará o inicializaci a správu aktérů, určených pro trénink a predikci. Protože tento aktér slouží jak pro křížovou validaci, tak pro samotnou predikci, je nutné zajistit, aby se při křížové validaci RootActor dozvěděl o přijetí poslední zprávy obsahující trénovací data, a nebo poslední zprávy určené k predikci a zároveň, aby se žádná taková zpráva neodeslala ve chvíli, kdy je aplikace určená pouze pro predikci. Toho jsem dosáhl časovou zprávou, která bude o těchto událostech informovat aktéra RootActor a to po vypršení předem definovaného časového úseku. Bez tohoto upozornění by se aplikace dostala do nekonzistentního stavu a nebylo by možné pokračovat v křížové validaci.

ModelManager je definován čtyřmi typy chování a to *startingState*, *predictState*, *shiftState* a *trainState*.

- *startingState* přijímá pouze jednu zprávu a to FeatureSizeForBayes, která obsahuje počet slov z vytvořeného slovníku. Tento počet je potřebný pro vytvoření Naive Bayes modelu. Po přijetí této zprávy dojde k vytvoření všech trenérů i prediktorů a ModelManager přejde do *trainState*.

- *trainState* je chování, ve kterém ModelManger přijímá pouze zprávy určené pro natrénování modelů.

[5]Stop slova ve SMILE:https://github.com/haifengl/smile/tree/master/nlp/src/main/resources/smile/nlp/dictionary

  - Train je zpráva obsahující data potřebná k natrénování modelu. Zprávy tohoto typu jsou v tomto aktérovi akumulovány pro další použití. Přijetí této zprávy resetuje časovou zprávu.
  - SetShiftMessage je příkaz poslaný od aktéra *RootActor*, který způsobí, že si ModelManager vytvoří časovou zprávu.
  - TrainModels je zpráva, která spustí samotný trénink všech modelů a to po přijetí časové zprávy. Všechny akumulované zprávy odešle trenérům pro jednotlivé modely. Pro zajištění bezpečnosti vnitřního stavu aktéra se po zpracování této zprávy změní chování aktéra na *shiftState*.
  - Done restartuje aktéra a všechny jeho potomky.

- *shiftState* je přechodné chování aktéra, ve kterém jednotliví trenéři vytvářejí své modely. Toto chování umožňuje přijmout pouze dva typy zpráv.
  - Done slouží pro restartování aktéra.
  - Trained je zpráva, která oznamuje, že se podařilo natrénovat další model. Aktér ModelManager zná celkový počet modelů a po přijetí každé zprávy Trained si vnitřně zvyšuje počet úspěšně natrénovaných modelů. Jakmile dojde k natrénování a distribuci všech modelů prediktorům, tak se chování aktéra změní na *predictState*.

- *predictState* je chování, které umožňuje určovat třídu poskytnutého e-mailu a poskytuje komunikační rozhraní definované pomocí následujících zpráv.
  - Predict je zpráva obsahující předzpracovaný e-mail. Tato zpráva je asynchronně delegována na aktéry určené pro predikci. Po obdržení výsledků, je vybrána predikovaná třída daného e-mailu pomocí sečtení všech vah pro každou výslednou třídu (každý model může mít jinou váhu). Třída, která má největší váhu je poté zvolena jako výsledek.
  - WriteModels přikazuje aktérovi, aby serializoval všechny natrénované modely do předem konfigurovaného adresáře. Poté aktér změní své chování na *trainState* a očekává další trénovací data.
  - Done restartuje aktéra a všechny jeho potomky.
  - SetShiftMessage nastaví časovou zprávu, která po svém vypršení odešle zprávu LastPredictionMade aktérovi RootActor o provedení poslední predikce. Tato časová zpráva se nastavuje pouze v případě křížové validace.

Trainer

Trainer je typový trait, který v sobě definuje chování všech tříd zastupujících jakéhokoli trenéra modelu. Tento trait v sobě deklaruje metodu a atributy, které musejí být ve třídě, která tento trait rozšiřuje, definovány. Každý trenér musí mít prediktory, kterým bude sdílet natrénovaný model. Dále musí mít definovánu cestu k adresáři, do kterého bude možné ukládat natrénované modely. Každý trenér také obsahuje jméno modelu, který má za cíl natrénovat a také funkci, která po každém zavolání vytvoří nový model. Tato funkce má následující předpis `def trainModel: (Array[Array[Double]], Array[Int]) => T`, kde T je generický typ a zastupuje výsledný model. První vstupní parametr představuje trénovací data a druhý parametr představuje třídy, do kterých trénovací data patří. Tento

trait rozšiřuje abstraktní třídu Actor a implementuje v sobě chování, které je stejné napříč všemi trenéry. Toto chování přijímá tyto zprávy.

- WriteModels přikazuje aktérovi, aby serializoval natrénovaný model a následně ho uložil do adresáře, který byl předán v konstruktoru třídy, která implementuje tento trait.
- TrainData je instance datové třídy obsahující trénovací data. Po přijetí této zprávy se zavolá funkce `def trainModel: (Array[Array[Double]], Array[Int]) => T` a stávající model bude nahrazen za nový. Následně je tento nový model předán všem prediktorům a rodiči je předána zpráva o natrénování modelu.

V současné chvíli existují čtyři druhy tříd, které implementují daný trait. Jsou to NaiveTrainer, SVMTrainer, KNNTrainer a AdaBoostTrainer. Každá tato třída v sobě implementuje metodu určenou k natrénování modelu daného typu.

Predikční modely

V celé aplikaci existuje pouze jedna třída určená pro predikci. Tato třída je použita pro všechny použité modely a jmenuje se *GenericPredictor*. Každý klasifikátor v knihovně SMILE implementuje `interface Classifier<T>`, který obsahuje metodu pro predikci. Tento aktér obsahuje pouze jedno chováni.

- UpdateModel je instance datové třídy obsahující serializovaný model. Tento model je deserializovaný a použitý pro predikci.
- CleansedEmail představuje zprávu, která obsahuje předzpracovaná data pro predikci.

Konfigurace

Nastavení tohoto modulu obsahuje konfigurační objekt *model* s následujícími možnostmi:

- *models* je pole polí, kde první prvek vnitřního pole je textový řetězec obsahující název modelu a druhý prvek je číslo v rozmezí 0 až 100 včetně, které určuje váhu, který daný model má. Je možné použít následující modely: SVM (3.2.3), AdaBoost (3.2.4), NaiveBayes (3.2.2) a KNN (3.2.5). Na velikosti písmen nezáleží.
- *write-model-to* je cesta k adresáři, do kterého budou zapsány natrénované modely.
- *number-of-predictors* představuje počet prediktorů běžících v jednom uzlu pro každý klasifikační model.

Tento modul definuje vlastní dispatcher určený pro aktéry, kteří se starají o predikci a trénování modelů. Tento dispatcher se nazývá *model-dispatcher* a výchozí nastavení mu přiděluje 2 vlákna. V ideálním případě by měl mít každý model přiřazeno jedno vlákno a to proto, aby mohlo být trénování modelů spuštěno souběžně.

Každý klasifikační model má definované vlastní nastavení, které bude popsáno v následujících částech.

Naivní Bayes

Nastavení Naivního Bayese obsahuje konfigurační objekt *naive-bayes*, který je obsažený v konfiguračním objektu *model*. Tento objekt slouží k nastavení následujících parametrů.

- *sigma* je desetinné číslo větší než nula, které je použito jako hodnota pro aditivní vyhlazování. Výchozí hodnota je nastavena na hodnotu 1,0.
- *model* představuje textový řetězec určující, který typ algoritmu bude použitý. SMILE poskytuje následující implementace: GENERAL, MULTINOMIAL, BERNOULLI a POLYAURN. Výchozí nastavení používá MULTINOMIAL.

Support Vector Machines

Tento algoritmus je možné nastavit v konfiguračním objektu *svm* obsaženém v konfiguračním objektu *model*.

- *kernel* umožňuje nastavit jádro SVM algoritmu. Je možné použít tato dvě jádra: GAUSSIAN a LINEAR. LINEAR je výchozí hodnota.
- *positive-margin-smoothing* představuje kladné desetinné číslo, které slouží pro penalizaci za výskyt v opačném (pozitivním) prostoru.
- *negative-margin-smoothing* představuje kladné desetinné číslo, které slouží pro penalizaci za výskyt v opačném (negativním) prostoru.
- *sigma* je pozitivní desetinné číslo, které je nutné nastavit pouze v případě použití jádra typu GAUSSIAN. Toto číslo představuje konstantu určenou pro zjemnění modelu.

AdaBoost

Tento model je možné nastavit v konfiguračním objektu *adaboost* obsaženém v konfiguračním objektu *model* a poskytuje následující možnosti.

- *number-of-trees* je přirozené číslo, které určuje počet rozhodovacích stromů, které vytvoří výsledný model. Výchozí hodnota je 1000 rozhodovacích stromů.
- *max-nodes* je přirozené číslo, které představuje počet listů každého rozhodovacího stromu. Výchozí hodnota je nastavena na 2.

K nejbližších sousedů

Tento model je možné nastavit v konfiguračním objektu *knn* obsaženém v konfiguračním objektu *model* a poskytuje následující možnosti.

- *k* je kladné číslo, které představuje počet nejbližších sousedů. Výchozí hodnota je číslo 5.

3.7.5 dictionary-resolver

Tento modul slouží pro získání slovníku. Pokud není aplikaci poskytnutý žádný slovník, tak se automaticky začne vytvářet z poskytnutých e-mailů, viz 3.7.2. Pro vytvoření slovníku je nezbytné poskytnout i soubor, který obsahuje rozdělení e-mailů do správné třídy, protože *DictionaryResolver* vytvoří slovník pro každou třídu.

Třídy

DictionaryResolver

Tato třída představuje implementaci aktéra, který se stará o obstarání slovníku. Tento aktér obsahuje pouze jedno chování, které přijímá následující zprávy.

- DataForDictionary je zpráva, která obsahuje kolekci předzpracovaných e-mailů s tím, že dané e-maily jsou ve formě dvojic - slovo, počet výskytů. E-maily jsou seskupeny podle třídy do které patří. Slova a jejich výskyty jsou poté sečteny pro každou třídu a seřazeny podle celkového počtu výskytů. Takto vytvořené slovníky jsou poté uloženy a z každého z nich je vybrán nakonfigurovaný počet slov. Tato slova si poté pošle sám sobě v datové třídě Dictionary.
- ResolveDictionary je zpráva, která spustí proces pro obstarání slovníku. Aktér se nejprve pokusí slovník získat z nakonfigurovaných souborů. Pokud nebyl poskytnutý žádný slovník, dojde k vytvoření aktéra *CleansedDataAccumulator*, který se dále postará o získání všech relevantních dat.
- Dictionary je zpráva, která obsahuje slova, která se použijí pro instanciaci třídy Bag. Po instanciaci je tento objekt serializován a odeslán aktérovi RootActor pro další distribuci.

CleansedDataAccumulator

Je implementace aktéra, která je zodpovědná za získání všech vyčištěných e-mailů. Tyto e-maily jsou následně použity pro tvorbu slovníku. Tento aktér obsahuje pouze jedno chování, které přijímá následující typy zpráv.

- CreateDictionary je instance datové třídy, která v sobě obsahuje referenci na LoadDataManager kterému aktér CleansedDataAccumulator pošle zprávu typu CreateDictionaryFromData.
- CleansedData je zpráva obsahující vyčištěná data, která budou použita pro vytvoření slovníku. CleansedDataAccumulator tyto zprávy akumuluje a neustále aktualizuje časovou zprávu, která slouží jako synchronizační prvek. Po přijetí poslední zprávy tohoto typu a vypršení časové zprávy je aktérovi CleansedDataAccumulator zaslána zpráva SendDictionary.
- SendDictionary je zpráva, která říká aktérovi CleansedDataAccumulator, aby odeslal naakumulovaná předzpracovaná data svému rodiči, což je aktér DictionaryResolver. Po odeslání zprávy se tento aktér ukončí.

Konfigurace

- *load-ham-dictionary-path* představuje cestu k souboru, který obsahuje slovníková data vytvořená z e-mailů, které patří do skupiny vyžádané pošty. Každé slovo v takovém souboru zabírá vlastní řádek. Pro použití výchozího slovníku je nutné zadat tuto hodnotu "/ham-dictionary.txt".
- *load-spam-dictionary-path* představuje cestu k souboru, který obsahuje slovníková data vytvořená z e-mailů, které patří do skupiny nevyžádané pošty. Každé takové

slovo zabírá v daném souboru jeden řádek.Pro použití výchozího slovníku je nutné zadat tuto hodnotu "/spam-dictionary.txt".

- *take-up-to* je přirozené číslo, které určuje počet slov, která budou z každého poskytnutého slovníku použita.
- *save-to* definuje cestu k adresáři, do kterého budou uloženy vytvořené slovníky.

Místo dvou slovníků je možné poskytnout pouze jeden. V tom případě nezáleží na tom, zda bude vyplněna cesta ke slovníku vyžádané či nevyžádané pošty.

3.7.6 results-aggregator

Třídy

ResultsAggregator

Tento aktér se soustředí na sběr výsledků predikce a obsahuje jedno chování, které umožňuje přijmout tyto zprávy.

- AfterPrediction je zpráva, která obsahuje výsledek predikce a to jak celkový výsledek, tak výsledky jednotlivých modelů. Tyto zprávy jsou akumulovány.
- WriteResults způsobí, že se všechny akumulované zprávy převedou do formátu JSON a zapíší se do nakonfigurovaného adresáře. Do toho adresáře se také zapíše celková úspěšnost predikcí (pokud je předem určeno, do které třídy daný e-mail patří).
- WriteGraph vytvoří grafy úspěšných a neúspěšných predikcí pro každý model a zapíše je do adresáře určeného pro výsledky.
- Done restartuje aktéra.

Konfigurace

Tento modul poskytuje konfigurační objekt s názvem *results-aggregator*, který obsahuje pouze možnost nastavit, do kterého adresáře se budou ukládat výsledky z křížové validace. Pro nastavení tohoto adresáře slouží atribut *results-directory*, který očekává cestu k adresáři.

3.7.7 email-parser-module

Tento modul obsahuje objekt (viz co je to objekt v jazyce Scala 3) EmailParser, který obsahuje metody pro načtení e-mailů z různých zdrojů (String, File, InputStream). Tento objekt je používán převážně v load-data-module a poté také v rest-api-module. Pro správné načtení e-mailů je použita knihovna Email mime parser.

3.7.8 rest-api-module

Třídy

V tomto modulu nejsou přítomni žádní aktéři. Místo toho je zde popsáno REST rozhraní pomocí knihovny akka-http a také třída, která se stará o zpracování požadavků. REST rozhraní je definované ve třídě *HttpServer* a třída *HttpServerHandler*, která se stará o zpracování požadavků. Tato třída zná referenci na aktéra *RootActor* a všechny požadavky jsou na něj delegovány s výjimkou příkazu o vypnutí.

Konfigurace

V tomto modulu je možné nastavit adresu a port http severu. Toto nastavení obsahuje konfigurační objekt *http*.

- *address* představuje textový řetězec, který obsahuje adresu, na které se spustí server. Výchozí hodnota je nastavena na hodnotu *localhost*.
- *port* je číselná hodnota v rozmezí 0 až 65535 včetně a určuje port, na kterém bude server naslouchat. Výchozí hodnota je nastavena na 4201.

Kapitola 4

Experimenty

Pro experimenty jsem použil podmnožinu, která čítala celkově 10 000 e-mailů[1]. Polovina z nich patří do vyžádané pošty a polovina do nevyžádané. Na těchto e-mailech jsem vždy provedl čtyři cykly křížové validace s různým nastavením jednotlivých klasifikátorů.

Použité slovníky jsem vytvořil z celé množiny dat, která obsahuje 75 419 e-mailů. Pro experimenty jsem použil sestavu čtyř klasifikátorů (SVM, kNN, NaiveBayes a AdaBoost). Sledovat budu jak výsledky jednotlivých klasifikátorů, tak výsledné predikce. Vyhodnocení výsledné predikce probíhá tak, že se seskupí výsledky jednotlivých klasifikátorů a výsledek, který má nejvyšší váhu je určen jako výsledný. Všechny klasifikátory budou trénovány na stejných datech. Nejprve se zaměřím na experimenty, ve kterých budu postupně zvyšovat počet použitých slov ze slovníků. Poté na základě výsledků vyberu nejlepší konfiguraci a upravím váhy jednotlivých klasifikátorů podle dosavadní úspěšnosti.

Pro vytvoření základního povědomí jsem provedl křížovou validaci s různým počtem slov z jednotlivých slovníků. Nejprve jsem použil 50, poté 100, 150, 200, 250, 500 a 1 000 slov z každého slovníku (pro vyžádanou a nevyžádanou poštu). Váhy byly pro každý klasifikátor nastaveny na stejnou hodnotu. Počet nejbližších sousedů metody KNN byl stanoven na 5. SVM klasifikátor byl nastaven tak, aby bylo použito lineární jádro a positivní i negativní penalizace byla nastavena na číslo 10. AdaBoost klasifikátor byl nastavený tak, aby sestával z 1000 slabých prediktorů. Výsledky jsou shrnuty v tabulce 4.1. Procenta v této tabulce představují celkovou úspěšnost výsledných predikcí.

Počet slov	1. Cyklus	2. Cyklus	3. Cyklus	4. Cyklus	Průměrná úspěšnost
50	93.237 %	93.04 %	93.88 %	93.679 %	93.459 %
100	95.12 %	95.6 %	95.679 %	95.518 %	95.48 %
150	96.16 %	96.198 %	96.32 %	96.48 %	96.29 %
200	96.679 %	95.68 %	96.2 %	95.72 %	96.07 %
250	98.199 %	97.96 %	97.76 %	97.68 %	97.9 %
500	97.2 %	98.44 %	97.999 %	97.76 %	97.85 %
1 000	98.6 %	98.48 %	97.999 %	98.2 %	98.32 %

Tabulka 4.1: Výsledky křížové validace při zachování základního nastavení klasifikátorů.

Nejvyššího výsledku dosáhly klasifikátory při použití 1000 slov z každého slovníku. V tabulce 4.2 jsou zobrazeny celkové počty špatně klasifikovaných e-mailů ve všech cyklech kří-

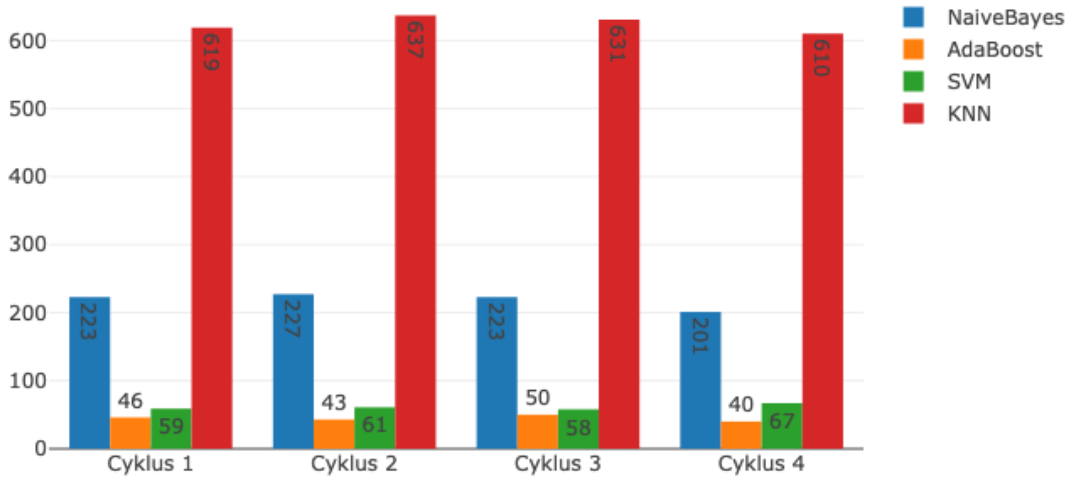
[1] 2007 TREC Public Spam Corpus: https://plg.uwaterloo.ca/~gvcormac/treccorpus07/

žové validace. Špatně klasifikovaný e-mail je takový, pro který byla dílčí klasifikace jiná než správná. Lze pozorovat zlepšení s počtem přidaných slov u AdaBoost, NaiveBayes a SVM klasifikátoru. KNN klasifikátor se s počtem použitých slov horší. Z výsledků je dále patrné, že AdaBoost je ze všech klasifikátorů nejlepší. Což může být způsobené vhodným nastavením. Provedu proto ještě jeden test s 1000 slovy, kdy zvednu počet slabých prediktorů na dvojnásobný počet, tedy 2000. Klasifikátor SVM si také vedl dobře, ale myslím, že by si mohl vést lépe. Pro další experiment zvýším pozitivní i negativní penalizaci na 17. Tuto hodnotu jsem zvolil čistě náhodně, abych viděl případný rozdíl a mohl se na základě tohoto rozdílu příště lépe rozhodovat. U klasifikátoru KNN zvýším počet nejbližších sousedů na 57 proto, abych viděl zlepšení či zhoršení klasifikace.

Počet slov	AdaBoost	NaiveBayes	SVM	KNN
50	733	1 799	667	662
100	524	1 560	514	663
150	436	1 643	438	727
200	441	1 649	481	700
250	224	1 465	464	762
500	235	1 053	288	960
1 000	172	880	243	1 099

Tabulka 4.2: Špatně určené třídy při použití různých počtů slov.

Na obrázku 4.1 je zobrazen počet špatně klasifikovaných e-mailů v každém kole křížové validace. Z tohoto experimentu je patrné, že zvýšením čísla k pro klasifikátor založený na nejbližších sousedech ničemu nepomohlo. Právě naopak. S celkovým počtem špatně vyhodnocených e-mailů rovných 2 497 se nepotvrdilo, že zvýšením k dojde ke zlepšení predikčních schopností. To může být způsobeno tím, že se při použití více slov začnou jednotlivé klasifikační kategorie překrývat. Klasifikátor SVM se v tomto případě zhoršil o 2 špatně vyhodnocené e-maily oproti dřívějšímu nastavení. AdaBoost klasifikátor špatně určil 179 e-mailů, což neznačí žádné zlepšení oproti tomu, kdy bylo použito pouze 1 000 slabých prediktorů namísto nynějších 2 000.



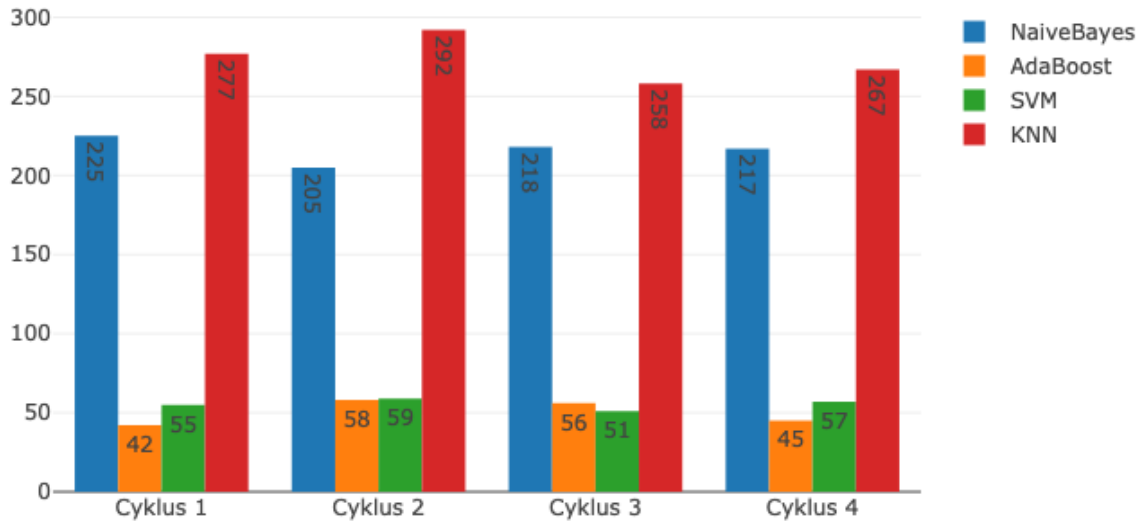
Obrázek 4.1: Počet nesprávně určených e-mailů pro každý klasifikátor v každém cyklu.

V dalším experimentu tedy použiji základní nastavení, neboť se ukázalo jako nejlepší, společně s 1 000 slov z každého slovníku. Rozdíl bude ve vahách, které bude každý jeden klasifikátor mít. Tyto váhy upravím podle poměru chybně klasifikovaných.

- AdaBoost klasifikátoru přidělím nejvyšší váhu, protože z dosavadních výsledků se mýlí nejméně. Váha tohoto klasifikátoru zůstane nezměněna a bude mít nejvyšší hodnotu.
- NaiveBayes klasifikátor je o více než 500 špatně predikovaných e-mailů horší než AdaBoost klasifikátor. Snížím proto jeho váhu na 50.
- SVM klasifikátor je pouze o 71 špatně určených e-mailů horší než AdaBoost a proto bude mít druhou nejvyšší váhu tj. 90.
- KNN klasifikátor je ze všech klasifikátorů nejhorší a proto bude mít nejmenší váhu tj. 20.

Na grafu 4.2 je vidět mírné zhoršení oproti stavu, kdy měl každý klasifikátor stejnou váhu. To je nejspíše způsobené načtením dat v jiném pořadí. Rozdíly nejsou nijak markantní.

- AdaBoost klasifikátor s celkovým počtem 202 špatně klasifikovaných e-mailů se zhoršil o 30 oproti předchozímu testu s použitím 1000 slov.
- SVM klasifikátor s počtem 222 špatně zařazených e-mailů dosáhl zlepšení o 21 e-mailů.
- NaiveBayes klasifikátor s počtem 865 špatně určených e-mailů je lepší než v případě předchozího testu s počátečními hodnotami (1 000 slov). Zlepšil se o 15.
- KNN klasifikátor se s celkovým počtem špatně klasifikovaných e-mailů 1094 zlepšil o 5 správně klasifikovaných e-mailů.



Obrázek 4.2: Počet nesprávně určených e-mailů při různých vahách.

Celková úspěšnost v jednotlivých kolech je zobrazena v tabulce 4.3. Je patrné, že úspěšnost klesla o 13 setin procenta oproti testu, kdy měl každý klasifikátor stejné váhy. Z toho vyplývá, že ponechání stejných vah je lepší než nastavit klasifikátorům různé váhy.

Počet slov	1. Cyklus	2. Cyklus	3. Cyklus	4. Cyklus	Průměrná úspěšnost
1 000	98,12 %	98.24 %	97.96 %	98.44 %	98.19 %

Tabulka 4.3: Úspěšnost při klasifikaci s různými váhami.

Cílem těchto experimentů nebylo nalézt optimální nastavení použitých klasifikátorů, nýbrž vytvořit ukázkovou aplikaci a ukázat její použití na několika typických experimentech, které souvisejí s vytvářením a použitím klasifikačních modelů.

Kapitola 5

Závěr

Z provedených experimentů a implementované aplikace vyplývá, že jazyk Scala je snadné a vhodné použít pro analýzu dat. Poskytuje veškeré nástroje, které jsou nezbytné pro analýzu dat a pokud tyto nástroje nejsou implementované přímo ve Scale, tak se dají použít nástroje implementované v jazyce Java. Interoperabilita mezi těmito dvěma jazyky přidává jazyku Scala velmi mnoho nástrojů.

S jazykem Scala se mi pracovalo velmi dobře. Po překonání počátečních problémů jsem zjistil, že je tento jazyk opravdu výborně navržený a další práce byla spíše zábavou. Nejvíce se mi líbí stručnost a možnosti, které nabízí např. typové třídy, implicitní parametry a možnost pracovat s funkcemi jako s běžnými proměnnými. Této možnosti jsem využil v případě rozesílání zpráv pro trenéry jednotlivých modelů a také pro predikce e-mailů.

Při experimentech dosahovala doba zpracování 100 e-mailů při použití 2 vláken a 2 aktérů CleanDataWorker rychlosti přibližně 100 e-mailů za 15 sekund. Při použití pouze 1 vlákna a 1 aktéra se doba zpracování zvýší na 25 sekund. Těchto výsledků jsem dosáhl na dvoujádrovém procesoru Intel core i5 7360U. Při použití dvou procesorů dosahuje rychlost zpracování přibližně 8-10 sekund. Druhý použitý procesor byl Intel core i3 4158U.

Literatura

[1] Berhane, F. : *Best packages for data manipulation in R.* [Online; navštíveno 20.3.2019]. Dostupné z: <https://www.r-bloggers.com/best-packages-for-data-manipulation-in-r/>

[2] Bronshtein, A. : *A Quick Introduction to the "Pandas" Python Library.* [Online; navštíveno 20.3.2019]. Dostupné z: <https://towardsdatascience.com/a-quick-introduction-to-the-pandas-python-library-f1b678f34673>

[3] contributors, W. : *Actor model.* [Online; navštíveno 15.4.2019]. Dostupné z: <https://en.wikipedia.org/w/index.php?title=Actor_model&oldid=891569665>

[4] contributors, W. : *Matplotlib.* [Online; navštíveno 20.3.2019]. Dostupné z: <https://en.wikipedia.org/w/index.php?title=Matplotlib&oldid=885147544>

[5] contributors, W. : *Representational state transfer.* [Online; navštíveno 5.4.2019]. Dostupné z: <https://en.wikipedia.org/w/index.php?title=Representational_state_transfer&oldid=890509010>

[6] contributors, W. : *R (programming language).* [Online; navštíveno 20.3.2019]. Dostupné z: <https://en.wikipedia.org/w/index.php?title=R_(programming_language)&oldid=888614452>

[7] Foundation, P. S. : *multiprocessing — Process-based parallelism.* [Online; navštíveno 15.4.2019]. Dostupné z: <https://docs.python.org/3/library/multiprocessing.html>

[8] Foundation, P. S. : *threading — Thread-based parallelism.* [Online; navštíveno 15.4.2019]. Dostupné z: <https://docs.python.org/3/library/threading.html>

[9] Ganesan, K. : *All you need to know about text preprocessing for NLP and Machine Learning.* [Online; navštíveno 25.4.2019]. Dostupné z: <https://www.kdnuggets.com/2019/04/text-preprocessing-nlp-machine-learning.html>

[10] Gordon, M. : *How-to go parallel in R.* [Online; navštíveno 15.4.2019]. Dostupné z: <https://www.r-bloggers.com/how-to-go-parallel-in-r-basics-tips/>

[11] Haller, P.; Prokopec, A.; Miller, H.; aj. : *Futures and promises.* [Online; navštíveno 15.4.2019]. Dostupné z: <https://docs.scala-lang.org/overviews/core/futures.html>

[12] Hjelle, G. A. : *The Ultimate Guide to Python Type Checking.* [Online; navštíveno 20.3.2019]. Dostupné z: <https://realpython.com/python-type-checking/>

[13] Inc., L. : *Akka documentation*. [Online; navštíveno 15.4.2019]. Dostupné z: <https://akka.io/docs/>

[14] Centrum zpracování přirozeného jazyka, F. M. : *Český Stop list*. [Online; navštíveno 5.4.2019]. Dostupné z: <https://nlp.fi.muni.cz/cs/StopList>

[15] JTableSaw : *Tablesaw documentation*. [Online; navštíveno 20.3.2019]. Dostupné z: <https://jtablesaw.github.io/tablesaw/>

[16] Kuhn, M. : *The caret available models*. [Online; navštíveno 20.3.2019]. Dostupné z: <http://topepo.github.io/caret/available-models.html>

[17] L, A. B. : *Top 8 programming languages every data scientist should master in 2019*. [Online; navštíveno 9.4.2019]. Dostupné z: <https://bigdata-madesimple.com/top-8-programming-languages-every-data-scientist-should-master-in-2019/>

[18] Li, M.; Paczuski, P. : *Top R Packages for Machine Learning*. [Online; navštíveno 20.3.2019]. Dostupné z: <https://www.kdnuggets.com/2017/02/top-r-packages-machine-learning.html>

[19] Manning, C. D.; Raghavan, P.; Schütze, H. : *Introduction to information retrieval*. Cambridge University Press, 2008. ISBN 0521865719.

[20] Petrov, C. : *Big Data Statistics 2019*. [Online; navštíveno 27.4.2018]. Dostupné z: <https://techjury.net/stats-about/big-data-statistics/>

[21] Plotly : *Plotly libraries*. [Online; navštíveno 20.3.2019]. Dostupné z: <https://plot.ly/graphing-libraries/>

[22] Schapire, R. E. : *Empirical Inference*. Springer, Berlin, Heidelberg, 2013. ISBN 978-3-642-41135-9.

[23] Venners, B.; Spoon, L.; Odersky, M. : *Programming in Scala*. Artima Press, 2016. ISBN 9780981531687.

[24] Wickham, H. : *ggplot2*. [Online; navštíveno 20.3.2019]. Dostupné z: <https://ggplot2.tidyverse.org/>

Příloha A

Obsah přiloženého paměťového média

Paměťové médium obsahuje zabalený soubor bp, který po rozbalení obsahuje:

- Adresář emailRecognition, ve kterém se nachází zdrojové soubory aplikace.
- Adresář trec07p obsahuje použitý dataset. V tomto adresáři se nachází adresář data, který obsahuje všechna data, adresář experiments, který obsahuje data použitá při experimentech a adresář full obsahuje soubor, který obsahuje přiřazení třídy k jednotlivým e-mailům.
- Adresář experiments obsahuje výsledky jednotlivých experimentů.
- Adresář compiled obsahuje adresáře: bin, conf a lib. V bin jsou skripty pro spuštění aplikace, conf obsahuje konfigurační soubory a lib obsahuje přeložené soubory.
- Soubor readme.txt, který obsahuje informace potřebné pro správné nastavení aplikace.

Příloha B

Manuál

Pro překlad a spuštění aplikace je nezbytné mít nainstalovanou Javu. Je možné použít verzi 8 a vyšší.

Všechny následující postupy byly vyzkoušeny na linuxovém prostředí.

B.1 Překlad zdrojových souborů

Pro překlad zdrojových souborů je nezbytné otevřít terminál a přesunout se do adresáře emailRecognition. V tomto adresáři se nachází skript nazvaný sbt. Pro jeho spuštění je nezbytné mít k dispozici bash. Překlad se poté provede pomocí příkazu "sbt compile". Pro spuštění aplikace z přeložených souborů je poté potřeba použít příkaz "sbt run".

B.2 Spuštění přeložených souborů

Pro spuštění již přeložených souborů je nutné přesunout se v terminálu do adresáře compiled/bin, kde je připravený skript email-recognition. Tento skript doporučuji spouštět s argumentem -J-Xmx6G. Tento argument zvýší velikost haldy na 6 GB. Pro experimenty byl tento limit postačující. V adresáři bin je také skript (email-recognition.bat) pro spuštění v systému Windows.

B.3 Ovládání aplikace

Pro ovládání aplikace je nutné mít k dispozici nástroj pro vykonávání http volání. V následující sekci jsou připraveny příkazy pomocí nástroje curl.

B.3.1 !start

curl -d '{}' -H "Content-Type: application/json"-X POST 'http://localhost:4201/controls/!start'

B.3.2 !train-models

curl -d '{}' -H "Content-Type: application/json"-X POST 'http://localhost:4201/controls/!train-models'

B.3.3 !cross-validation

curl -d '{}' -H "Content-Type: application/json"-X POST 'http://localhost:4201/controls/!cross-validation'

B.3.4 !terminate

curl -d '{}' -H "Content-Type: application/json"-X POST 'http://localhost:4201/controls/!terminate'

B.3.5 !restart

curl -d '{}' -H "Content-Type: application/json"-X POST 'http://localhost:4201/controls/!restart'

B.3.6 !predict

Obsah atributu data musí být zakódován v base64.

curl -d '{"data": "BASE64ENCODED"}' -H "Content-Type: application/json"-X POST 'http://localhost:4201/predictions/!predict'