FAKULTA APLIKOVANÝCH VĚD
ZÁPADOČESKÉ UNIVERZITY
V PLZNI

# Identifikace spamu naivním bayesovským klasifikátorem

Semestrální práce KIV/PC

Štěpán Faragula
A21B0119P

2. ledna 2023

# Obsah

# Kapitola 1

# Zadání

Při volbě zadání semestrální práce jsme měli na výběr z následujících možností:

1. Hledání kořenů rovnice
2. Identifikace spamu naivním bayesovským klasifikátorem
3. Celočíselná kalkulačka s neomezenou přesností

V této práci je popsáno řešení práce **číslo 2**.

## 1.1 Detaily zadání

Naprogramujte v ANSI C přenositelnou **konzolovou aplikaci**, která bude **rozhodovat, zda úsek textu** (textový soubor předaný jako parametr na příkazové řádce) **je nebo není spam**.

Program bude přijímat z příkazové řádky celkem **sedm** parametrů: První dva parametry budou vzor jména a počet trénovacích souborů obsahujících nevyžádané zprávy (tzv. **spam**). Třetí a čtvrtý parametr budou vzor jména a počet trénovacích souborů obsahujících vyžádané zprávy (tzv. **ham**). Pátý a šestý parametr budou vzor jména a počet testovacích souborů. Sedmý parametr představuje jméno výstupního textového souboru, který bude po dokončení činnosti Vašeho programu obsahovat výsledky klasifikace testovacích souborů.

Program se tedy bude spouštět příkazem

`spamid.exe` ⟨`spam`⟩ ⟨`spam-cnt`⟩ ⟨`ham`⟩ ⟨`ham-cnt`⟩ ⟨`test`⟩ ⟨`test-cnt`⟩ ⟨`out-file`⟩ ↵

Symboly ⟨`spam`⟩, ⟨`ham`⟩ a ⟨`test`⟩ představují vzory jména vstupních souborů. Symboly ⟨`spam-cnt`⟩, ⟨`ham-cnt`⟩ a ⟨`test-cnt`⟩ představují počty vstupních souborů. Vstupní soubory mají následující pojmenování: `vzorN`, kde `N` je celé číslo z intervalu $\langle 1;N \rangle$.

Přípona všech vstupních souborů je `.txt`, přípona není součástí vzoru. Váš program tedy může být během testování spuštěn například takto:

```
spamid.exe spam 10 ham 20 test 50 result.txt ⏎
```

Výsledkem činnosti programu bude textový soubor, který bude obsahovat seznam testovaných souborů a jejich klasifikaci (tedy rozhodnutí, zda jde o spam či neškodný obsah – ham).

Pokud nebude na příkazové řádce uvedeno právě sedm argumentů, vypište chybové hlášení a stručný návod k použití programu v angličtině podle běžných zvyklostí (viz např. ukázková semestrální práce na webu předmětu Programování v jazyce C). **Vstupem programu jsou pouze argumenty na příkazové řádce – interakce s uživatelem pomocí klávesnice či myši v průběhu práce programu se neočekává.**

Hotovou práci odevzdejte v jediném archivu typu ZIP prostřednictvím automatického odevzdávacího a validačního systému. Postupujte podle instrukcí uvedených na webu předmětu. Archiv nechť obsahuje všechny zdrojové soubory potřebné k přeložení programu, **makefile** pro Windows i Linux (pro překlad v Linuxu připravte soubor pojmenovaný `makefile` a pro Windows `makefile.win`) a dokumentaci ve formátu PDF vytvořenou v typografickém systému TEX, resp. LATEX. Bude-li některá z částí chybět, kontrolní skript Vaši práci odmítne.

# Kapitola 2

# Analýza úlohy

V úloze máme za úkol **zařadit soubory** do jedné ze dvou tříd – **spam** či **ham**. Je nám výrazně doporučeno použít **naivní bayesovský klasifikátor**, není tedy důvod rozebírat, jaký způsob klasifikace zvolíme.

## 2.1 Naivní bayesovský klasifikátor

Algoritmus, který má dvě fáze – **fáze učení** a **fáze klasifikace**.

### 2.1.1 Fáze učení

V této fázi vycházíme z předpokladu, že máme k dispozici **trénovací soubory** obsahující pouze slova označena jako spam nebo ham. U každého přečteného **slova** budeme uchovávat informaci o jeho **počtu výskytu** v souboru a jeho **podmíněnou pravděpodobnost výskytu**. Tímto způsobem přečteme každý trénovací soubor a všechna slova spojíme do **slovníku klasifikátoru**. Takto vytvořený **slovník** by měl ještě obsahovat **apriorní pravděpodobnost**, která analýzou vstupních souborů stanoví **výchozí pravděpodobnost výskytu spamu či hamu**.

### 2.1.2 Fáze klasifikace

**Testovací soubory** budeme klasifikovat podle jeho obsažených slov. Průběh klasifikace je takový, že každému slovu vyskytujícímu se v **testovacím souboru a zároveň ve slovníku klasifikátoru** přiřadíme **podmíněnou pravděpodobnosti výskytu** slova spamu a hamu. Dále sečteme logaritmy všech **podmíněných pravděpodobností** slov a přičteme logaritmus **apriorní pravděpodobnosti** třídy. Soubor zařadíme do té třídy, která bude mít **vyšší výsledek**. Klasifikace souboru je popsána rovnicí 2.1 na straně 5 kde:

- $C$ – množina všech tříd

- $c_i$ – označení třídy (spam, ham)
- $P(c_i)$ – apriorní pravděpodobnost třídy
- $P(\langle \text{word}_k | c_i \rangle)$ – podmíněná pravděpodobnost výskytu slova

$$c = \arg\max_{c_i \in C} \left( \log(P(c_i)) + \sum_{k \in \textbf{text}} \log(P(\langle \text{word}_k | c_i \rangle)) \right) \tag{2.1}$$

## 2.2 Definice problému

Klasifikátor potřebuje ke své činnosti **slovník**. **Slovník** můžeme vnímat jako **datovou strukturu**, která dokáže vrátit **položku** podle hledaného výrazu (řetězce). Musíme vnímat **rozdíl mezi položkou a slovníkem**. **Položka slovníku** je slovo reprezentováno řetězcem.

Dále potřebujeme vyřešit samotný **způsob klasifikace souborů**.

## 2.3 Volba datové struktury pro slovník

Slovník bude obsahovat informace o všech **unikátních slovech** ze skupiny souborů. Ve slovníku **nepotřebujeme mazat** položky. Pro naši potřebu nám tedy stačí pouze operace **přidání** a **hledání**. Chceme, aby tyto funkce pracovaly **co nejrychleji**.

### 2.3.1 Spojový seznam

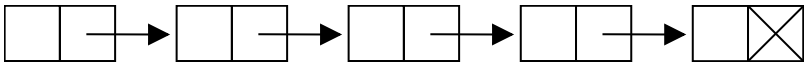
Jedná se o **seznam proměnné délky**, kde si každý prvek uchovává referenci na další prvek v seznamu. Spojový seznam je velmi jednoduchý na implementaci a zajišťuje **snadné přidání** prvku do seznamu v čase *O(1)*. Jeho obrovskou nevýhodou je, že neumožňuje přístup k prvku na konkrétním indexu. Kdybychom chtěli najít prvek podle nějakého klíče, museli bychom projít celý seznam a vždy porovnávat hodnotu s klíčem. **Časová složitost vyhledávání** je tedy *O(n)*.

Spojový seznam má sice rychlý čas vložení prvku, avšak kvůli dlouhému vyhledávání to **není** vhodná struktura pro naši úlohu.

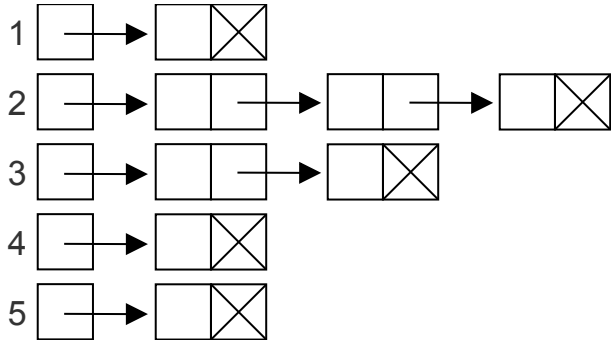
Na obrázku 2.1 na straně 6 je vidět grafické znázornění spojového seznamu.

### 2.3.2 Tabulka s rozptýlenými hodnotami

**Tabulka s rozptýlenými hodnotami**, tzv. *hash tabulka*, je datová struktura, která umožňuje rychlou manipulaci prvků podle **klíče**. Ke své činnosti využívá **hashovací funkci**, pomocí které získáme přístup k indexu položek. Kvalita této funkce se odrazí na celkové kvalitě tabulky. Chceme, aby funkce generovala vždy **stejné číslo pro stejný**



Obrázek 2.1: Spojový seznam




Obrázek 2.2: Tabulka s rozptýlenými hodnotami (hash tabulka)

**vstup**. Poté je důležité, aby funkce přiřazovala novým položkám indexy **rovnoměrného rozdělení**. Je také důležité se co nejvíce **vyhýbat kolizím**. Právě kvůli kolizím je ve skutečnosti na každém indexu tabulky uložen spojový seznam. Pokud k nějaké kolizi dojde, jednoduše přiřadíme hodnotu do spojového seznamu. **Časová složitost přidání** prvku do tabulky je tedy *O(1)*. Při hledání prvku získáme index, na kterém se prvek nachází, v čase *O(1)*. Poté stačí hodnotu hledat ve spojovém seznamu. Pokud máme **velmi dobrou hashovací funkci**, na každém indexu tabulky by se nacházel právě jeden prvek. **Celková časová složitost hledání** by tedy byla *O(1)*. Při použití **mizerné hashovací funkce** bude docházet ke kolizím a v nejhorším případě se všechny prvky tabulky přiřadí právě na jeden index. V takovém případě by **celková časová složitost hledání** byla *O(n)*.

Pro řešení práce **byla využita** právě hash tabulka, jelikož je to pro náš účel ideální datová struktura. Při samotné implementaci však budeme muset dbát na vymýšlení co nejlepší hashovací funkce.

Na obrázku 2.2 na je vidět grafické znázornění tabulky s rozptýlenými hodnotami.

## 2.4 Způsob klasifikace

Zde se nabízí dvě možnosti: (i) Vytvoříme další **slovník** obsahující slova testovacího souboru, který následně **porovnáme se slovníkem klasifikátoru**, (ii) nebo jednotlivá slova budeme porovnávat tzv. *on the fly*, kde **postupným čtením slov** počítáme pravděpodobnost tříd a soubor klasifikujeme jako spam či ham **bez vytváření dalšího slovníku**. Výhoda přístupu (i) je uchování čitelnosti programu za cenu vyšší paměťové náročnosti. Výhoda přístupu (ii) je jednoduchost implementace bez nároků na paměť.

Při řešení této semestrální práce byl využit postup (ii).

# Kapitola 3

# Popis implementace

Program je implementován v **programovacím jazyce C** standardu **ANSI**.

## 3.1 Moduly programu

Modulem je myšleno **dvojice hlavičkového souboru** (přípona `.h`) a **zdrojového kódu** (přípona `.c`). Funkce jednotlivých modulů je následující:

- `hash_table` – datová struktura hash tabulky
- `hash_table_entry` – datová struktura položky pro hash tabulku
- `classifier` – bayesovský klasifikátor
- `error` – obsahuje chybové hlášky

Dále se v práci nachází soubor `main.c`, který **vykonává algoritmus** naivního bayesovského klasifikátoru a **spouští program**. Nakonec je v projektu soubor `config.h`, kde jsou **pomocné definice**.

## 3.2 Implementace hash tabulky

Implementace hash tabulky je rozdělena na **dvě části**, jednou je **položka tabulky**, druhou **samotná tabulka**. Každá část má svou **vlastní strukturu**.

### 3.2.1 Struktura položky tabulky

Jedná se o **přečtené slovo z trénovacího souboru**. Struktura položky je implementována jako **spojový seznam** a nazývá se `entry`.

Každá položka si uchovává svůj **unikátní klíč** (řetězec), **počítadla výskytu slova** v souborech spamu a hamu, **podmíněnou pravděpodobnost výskytu** v těchto souborech a **referenci na další položku**. Struktura obsahuje **vlastní funkce přidání a hledání položky**, které jsou **následně volány v hash tabulce**.

**Novou položku** vytvoříme funkcí `entry_create()`, která alokuje potřebnou paměť. Maximální délka řetězce je dána hodnotou `STRING_LENGTH` v souboru `config.h`.

**Vložení položky** do seznamu provedeme funkcí `entry_insert()`, která novou položku přidá a při vložení již existující navýší počítadlo.

**Hledání položky** provedeme funkcí `entry_find()`.

### 3.2.2 Struktura hash tabulky

Hash tabulka nám slouží pro **uložení slov trénovacích souborů**. Má vždy **stejnou velikost** podle hodnoty `TABLE_SIZE` v hlavičkovém souboru `hash_table.h`. Tabulka **ukládá položky** `entry`, které se mohou zřetězit a vytvořit tak **spojový seznam**. Do tabulky tedy ukládáme **reference na první prvky spojového seznamu**.

Tabulka obsahuje **reference na položky**, vypočítanou **apriorní pravděpodobnost tříd** a různá **počítadla**. Struktura je v kódu **řádně okomentována**, nemá zde smysl rozebírat, co které počítadlo počítá. Nazývá se `hash_table`.

Pro **vytvoření nové tabulky** voláme funkci `table_create()`, která alokuje paměť a nastaví reference na každém indexu na `NULL`.

Pro **vložení nové položky** voláme funkci `table_insert()`, která podle hashovací funkce `hash_function()` přiřadí index v tabulce podle převzatého klíče. Parametrem `flag` rozlišujeme, jestli se jedná o **spamovou** či **hamovou položku**. Samotné vkládání do spojového seznamu řeší volaná funkce `entry_insert()` ve struktuře `entry`.

**Položku hledáme** funkcí `table_find()`, která vypočítá index položky pomocí hashovací funkce a samotné hledání položky ve spojovém seznamu je opět řešeno voláním funkce `entry_find()` ve struktuře `entry`.

## 3.3 Implementace bayesovského klasifikátoru

**Klasifikátor** využívá strukturu `hash_table` pro **uložení slov z trénovacích souborů**. Samotný algoritmus řídí soubor `main.c`, jehož funkce je popsána pseudokódem 1 na straně 9. Algoritmus využívá zejména funkce modulu `classifier`:

- `classifier_train()` – načte trénovací soubory do tabulky
- `compute_probability()` – vypočítá pravděpodobnost spamu/hamu
- `classifier_test()` – načte a klasifikuje testovací soubory

**Algorithm 1** Algoritmus vykonaný souborem `main.c`

```
1:  procedure MAIN(argc, argv[])
2:      ▷ Funkce jednotlivých argumentů definována v souboru main.c
3:      spam_pattern, ham_pattern, test_pattern
4:      spam_count, ham_count, test_count
5:      output
6:      ▷ Definováno v souboru config.h
7:      FLAG_SPAM, FLAG_HAM
8:      ▷ Vytvoří prázdnou hash tabulku
9:      table ← hash_table table_create()
10:     ▷ Načte trénovací soubory do tabulky
11:     classifier_train(table, spam_pattern, spam_count, FLAG_SPAM)
12:     classifier_train(table, ham_pattern, ham_count, FLAG_HAM)
13:     ▷ Vypočítá pravděpodobnost výskytu slov v tabulce
14:     compute_probability(table)
15:     ▷ Klasifikuje testovací soubory
16:     classifier_test(table, test_pattern, test_count, output)
17:     ▷ Uvolní použitou paměť
18:     table_free(table)
```

U **načítání trénovacích souborů** předáváme parametr `FLAG_SPAM` pro spamový soubor nebo `FLAG_HAM` pro hamový soubor, které jsou definovány v souboru `config.h`.

**Klasifikace testovacích souborů** probíhá způsobem *on the fly* popsaným v sekci 2.4.

# Kapitola 4

# Uživatelská příručka

Úloha je zamýšlena pro **spuštění v konzoli**. Při běhu programu se neočekává žádná interakce s uživatelem.

## 4.1 Přeložení programu

Program se překládá pomocí nástroje `make`. V archivu jsou připraveny dvě verze souboru `makefile`, jedna pro operační systémy typu **UNIX**, druhá pro systémy **Windows**. Na přeložení je nutné mít **nainstalovaný kompilátor** `gcc`. Pokud se program povedlo přeložit, vytvoří se v adresáři **spustitelný soubor** `spamid`.

## 4.2 Spuštění programu

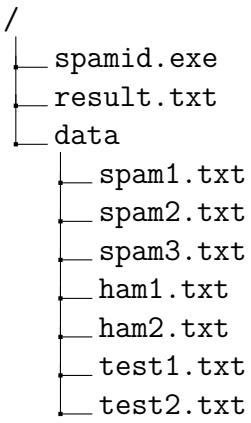
Spuštění probíhá v **příkazové řádce** operačního systému. Program očekává celkem 7 parametrů, při jejich nesprávném počtu nebo při zadání nevalidní hodnoty program ukončí činnost a vypíše způsob jeho spuštění i s příkladem. Spuštění je tedy následující:

`spamid.exe` ⟨`spam`⟩ ⟨`spam-cnt`⟩ ⟨`ham`⟩ ⟨`ham-cnt`⟩ ⟨`test`⟩ ⟨`test-cnt`⟩ ⟨`out-file`⟩ ↵

**Před spuštěním** se očekává, že v adresáři `data` jsou umístěny **trénovací a testovací soubory**. Ukázka adresářové struktury je vidět na obrázku 4.1 na straně 11. Cestu k adresáři s těmito soubory můžeme změnit v souboru `config.h`. Program umí pracovat pouze s textovými soubory (přípona `.txt`) obsahující **jeden řádek slov**, každé **oddělené mezerou**.

**Výsledkem programu** je výstupní soubor s názvem ⟨`out-file`⟩. Při zadávání názvu souboru musíme zahrnout i příponu (`.txt`). Výstup obsahuje na každém řádku název testovaného souboru a jeho klasifikaci (`S` pro spam, `H` pro ham) v následujícím formátu:

⟨`jméno souboru`⟩ ⟨`tabulátor`⟩ ⟨`klasifikace`⟩ ⟨`znak konce řádku`⟩




Obrázek 4.1: Ukázka adresářové struktury

```
test1.txt ⟶ S ⮐
test2.txt ⟶ S ⮐
test3.txt ⟶ H ⮐
test4.txt ⟶ S ⮐
test5.txt ⟶ H ⮐
```

Obrázek 4.2: Ukázka výstupního souboru

Na obrázku 4.2 je vidět ukázka konkrétního výstupu.

# Kapitola 5

# Závěr

Tato kapitola shrnuje výsledky celé práce.

## 5.1 Shrnutí

Klasifikátor dokáže identifikovat spamové či hamové soubory **s přesností 93 %**. Tato hodnota vyhovuje **minimální hranici přesnosti 90 %**, povedlo se nám tedy vytvořit **klasifikátor splňující zadání**. Při programování jsme využili **vlastní implementaci hash tabulky**, jelikož standard ANSI C nenabízí žádnou podobnou datovou strukturu. **Klasifikaci souborů** jsme prováděli způsobem *on the fly*, jelikož je jednodušší na implementaci a nezabírá zbytečně paměť.

## 5.2 Zhodnocení

**Nejobtížnější část práce** bylo pochopit, jakým způsobem funguje **naivní bayesovský klasifikátor**. Dále jsem měl problémy s **korektním uvolňováním paměti**, s tím mi naštěstí pomohl nástroj `valgrind`. Práce jinak probíhala **bez větších potíží**. Kód jsem se snažil psát **systematicky** a zároveň dodržovat veškerá **pravidla standardu ANSI C**.

## 5.3 Možná vylepšení

Některé části programu by se mohly **více zobecnit**, například struktura `hash_table`, konkrétně její položka `entry`. Mohlo by se jednat o **abstraktní položku spojového seznamu**, která by si uchovávala **referenci na stávající položku**. Atribut `next` by v tomto případě **ukazoval na obal** položky. Tímto způsobem bychom zvýšili **znovupoužitelnost kódu**.