VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

GENEROVÁNÍ KÓDU Z TEXTOVÉHO POPISU FUNKCIONALITY

GENERATING CODE FROM TEXTUAL DESCRIPTION OF FUNCTIONALITY

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE **JÁN KAČUR**
AUTHOR

VEDOUCÍ PRÁCE **doc. RNDr. PAVEL SMRŽ, Ph.D.**
SUPERVISOR

BRNO 2021

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav počítačové grafiky a multimédií (UPGM) Akademický rok 2020/2021

Zadání bakalářské práce

Student: **Kačur Ján**
Program: Informační technologie
Název: **Generování kódu z textového popisu funkcionality**
Generating Code from Textual Description of Functionality
Kategorie: Umělá inteligence

Zadání:

1. Seznamte se se způsoby využití rozsáhlých jazykových modelů typu GPT-3 a jejich adaptace na doménových datech, např. archivu GitHub a StackExchange.
2. Zpracujte dostupná data komentovaných kódů tak, aby bylo možné průběžně vyhodnocovat výsledky vytvářených modelů.
3. Na základě získaných poznatků navrhněte a implementujte systém, který dokáže na základě hlaviček funkcí a prvotních komentářů navrhovat strukturu následného kódu.
4. Vyhodnoťte výsledky systému na reprezentativním vzorku dat, diskutujte závislost výsledků na intuitivní složitosti popisu.
5. Vytvořte stručný plakát prezentující práci, její cíle a výsledky.

Literatura:

- dle doporučení vedoucího

Pro udělení zápočtu za první semestr je požadováno:

- funkční prototyp řešení

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/

Vedoucí práce: **Smrž Pavel, doc. RNDr., Ph.D.**
Vedoucí ústavu: Černocký Jan, doc. Dr. Ing.
Datum zadání: 1. listopadu 2020
Datum odevzdání: 30. července 2021
Datum schválení: 30. října 2020

Abstrakt

Cieľom tejto práce bolo navhrnúť a implementovať systém na generovanie kódu z textového popisu funkcionality. Boli vypracované celkovo 2 systémy, prvý z nich slúžil ako kontrolný prototyp, a druhý ako reálny výstup práce. Zameral som sa na použitie nepredtrénovaných modelov s menšími rozmermi. Obidva systémy používali ako jadro model typu Transformer. Druhý systém využil na rozdiel od prvého syntaktický rozklad kódu aj textových popisov. Dáta pre obidva systémy pochádzali z projektu CodeSearchNet, cieľový jazyk pre generovanie bol jazyk Python. Druhý systém dosiahol lepšie číselné výsledky, ako prvý, s presnosťou predpovede slov 85%, zatiaľ čo prvý len 60%. Systém dokázal doplniť správny kód na dokončenie funkcie, s väčšou časovou odozvou. V tejto práci sa venujem takmer výlučne druhému systému.

Abstract

The aim of this thesis was to design and implement system for code generation from textual description of functionality. In total, 2 systems were implemented. One of them served its purpose as a control prototype, the second one was the main product of this thesis. I focused on using smaller non-pre-trained models. Both systems used Transformer type model as their cores. The second system, unlike the first, used syntactic decomposition of both code and textual descriptions. Data used in both systems originated from project CodeSearchNet. Targer programming language to generate was Python. The second system achieved better quantitative results than the first one, with accuracy of 85% versus 60%. The system managed to auto-complete correct code to finish the function definition, with bigger time delay. This thesis is almost exclusively dedicated to the second system.

Kľúčové slová

generovanie kódu, doplňovanie kódu, spracovanie prirodzeného jazyka, strojové učenie, transformer, gpt, tensorflow, abstraktný syntaktický strom, POS tagging

Keywords

code generation, code prediction, nlp, machine learning, transformer, gpt, tensorflow, abstract syntax tree, POS tagging

Citácia

KAČUR, Ján. *Generování kódu z textového popisu funkcionality.* Brno, 2021. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce doc. RNDr. Pavel Smrž, Ph.D.

Generování kódu z textového popisu funkcionality

Prehlásenie

Prehlasujem, že som túto bakalársku prácu vypracoval samostatne pod vedením pána doc. RNDr. Pavla Smrža, Ph.D. Uviedol som všetky literárne pramene, publikácie, a ďalšie zdroje, z ktorých som čerpal.

.....
Ján Kačur
29. júla 2021

Poďakovanie

Chcel by som poďakovať môjmu vedúcemu práce, pánovi doc. RNDr. Pavlovi Smržovi, Ph.D., za odbornú pomoc počas vypracovávania tejto práce.

Obsah

1 Úvod **3**

2 Koncepty v spracovaní prirodzeného jazyka a strojovom učení **5**
2.1 Transformer 5
2.2 POS tagging 9
2.3 GPT 9
2.4 Metriky v strojovom učení 10
2.5 Tokenizácia 10
2.6 Vzorkovacie heuristiky 12
2.7 Softmax 13

3 Zhrnutie doterajšieho postupu **14**

4 Dáta **15**
4.1 Dostupné datasety 15
4.2 CodeSearchNet 15

5 Návrh riešenia **17**
5.1 Spracovanie dát 17
5.2 Syntaktické spracovanie vstupu 18
5.3 Tokenizér 20
5.4 Model 22
5.5 Kontrola syntaxe 23
5.6 Vzorkovacie heuristiky 24
5.7 Užívateľské rozhranie 24

6 Implementácia **27**
6.1 Spracovanie dát 27
6.2 Tokenizácia 29
6.3 Detokenizácia kódu 38
6.4 Kontrola gramatiky 39
6.5 Model a tréning 40

7 Testovanie **41**
7.1 Prvá fáza testovania 41
7.2 Druhá fáza testovania 42

8 Záver **47**

Literatúra **48**

A Obsah pamäťového média **50**

B Plagát **53**

Kapitola 1

Úvod

Generovanie kódu je pomerne nová disciplína v oblasti strojového učenia. Všetci programátori vedia, aké je to písať repetitívny kód. Niekedy je jednoduchá myšlienka ťažko prevediteľná do samotného kódu, alebo to skrátka zaberie neprimerane dlhý čas. Preto sa rôzni výskumníci snažia navrhnúť systém, ktorý by dokázal takýto kód napísať na základe textového popisu, alebo doplniť na základe už napísaného kódu.

Práve sa nachádzame v čase, kedy takéto systémy dosahujú úrovne, pri ktorej má zmysel hovoriť o masovom komerčnom používaní. Tieto systémy používaju jazykové modely obrovských rozmerov, trénované na extrémnom množstve dát. Tieto dáta sú zväčša zozbierané z veľkých internetových repozitárov. Vďaka týmto skutočnostiam tieto modely potrebujú enormné množstvo výpočtového výkonu a pamäte.

V tejto práci sa budem zaoberať návrhom a implementáciou podobného systému. Cieľom je navhrnúť systém na generovanie kódu. Jeden zo zásadných rozdielov bude v tom, že sa pokúsim takýto systém navrhnúť v oveľa menšom merítku. To znamená menej dát, menší rozmer modelu, menšia výpočtová kapacita, atď. Pri takýchto obmedzeniach je potrebné vyťažiť z poskytnutých dát absolútne maximum.

Vo viacerých prácach v tomto odvetví prevláda ďalší trend. Okrem veľkých dát a veľkých modelov sa viacero tímov snažilo extrahovať z poskytnutého kódu viac informácií vďaka syntaktickej analýze. Cieľom tohoto prístupu je, aby sa model neučil len lexikálny význam jednotlivých slov, ale aj ten syntaktický, resp. gramatický. Programovacie jazyky sú totiž vysoko štruktúrované, dá sa teda očakávať, že premiestnenie záujmu z lexikálneho na syntaktický význam prinesie zlepšenie kvality modelov.

Jedným zo spôsobov, ako chcem dosiahnuť vyššiu kvalitu modelu, je okrem použitia syntaktického významu kódu podobne využiť aj syntaktický význam vety opisujúcej funkcionalitu tohoto kódu. Cieľom je zistiť, či model dokáže nájsť súvislosti medzi gramatickým významom prirodzeného jazyka a kódu. K tomuto problému sa staviam podobne, ako ku strojovému prekladu. Narozdiel od prekladu medzi dvoma prirodzenými jazykmi však pôjde o preklad z prirodzeného jazyka do programovacieho jazyka. Na porovnanie výsledkov s modelom, ktorý nepoužíva syntaktický rozklad textu budem okrem v celej práci spomínaného systému implementovať ešte jeden, ktorý je akýmsi prototypom výsledného systému. Jeho detailom sa nevenujem do hĺbky, keďže sa dá povedať, že ide o podmnožinu výsledného systému.

V kapitole 2 sa venujem vysvetleniu pojmov z oblasti spracovania prirodzeného jazyka, ako aj strojového učenia všeobecne. V kapitole 3 je krátko zhrnutý doterajší postup v oblasti generovania kódu, súčasné trendy, a dosiahnuté míľniky. V kapitole 4 opisujem dostupné datasety pre túto úlohu, a rozoberám vlastnosti jedného z nich vybratého pre túto prácu.

V kapitole 5 je návrh jednotlivých komponentov systému, a vzťahov medzi nimi. Kapitola 6 potom pojednáva o praktickejších aspektoch tohoto návrhu, venuje sa implementácii a konkrétnym použitým technikám. Posledná kapitola 7 sa venuje testovaniu výsledného systému, vyhodnoteniu výsledkov, a porovnaniu systému s jeho prototypom.

Kapitola 2

Koncepty v spracovaní prirodzeného jazyka a strojovom učení

2.1 Transformer

Transformer [23] je jedna z architektúr jazykových modelov. Základným konceptom tejto architektúry je mechanizmus pozornosti[1]. Tento model dosahuje v porovnaní s predchádzajúcimi jazykovými modelmi oveľa lepšie výsledky, a to najmä v oblasti strojového prekladu, ako aj v iných oblastiach spracovania prirodzeného jazyka.

Pozornosť

Pozornosť je mechanizmus využívaný v spracovaní prirodzeného jazyka. Jeho podstatou je venovanie pozornosti modelu na jednotlivé časti vstupu. V architektúre Transformer sa používa tzv. *self-attention*, to znamená, že model sa učí, aký vzťah majú jednotlivé slová medzi sebou. Pozornosť sa počíta podľa tohto vzorca:

$$Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V \tag{2.1}$$

kde Q, K, V sú matice *Query*, *Key*, *Value*. Výpočet pozornosti je ukázaný graficky na obrázku 2.1

Multi-head attention

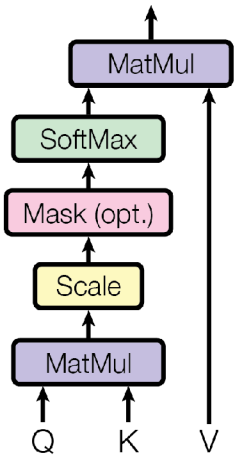
Architektúra Transformera využíva zložitejšiu verziu pozornosti, tzv. *Multi-head attention*. Podstata tejto techniky je rozdelenie vstupnej matice na h častí s rovnakým počtom prvkov, pričom pre každú z týchto častí sa vypočíta pozornosť osobitne. Z toho vyplýva, že musí platiť $\mod d_{model}, h = 0$. Výsledok sa potom konkatenuje do jednej matice. Princíp tejto techniky je opísaný vzorcom:

$$MultiHead(Q, K, V) = Concat(head_1, ..., head_h)W^O \tag{2.2}$$

kde

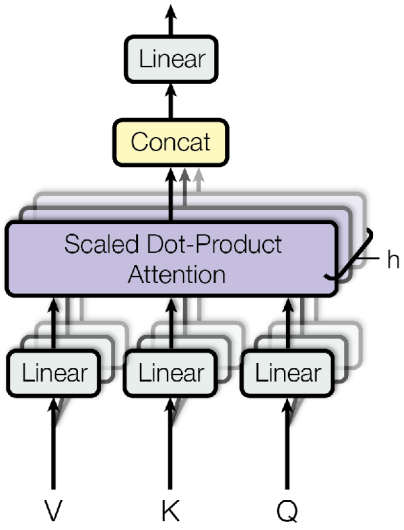
$$head_i = Attention(QW_i^Q, KW_i^K, VW_i^V) \tag{2.3}$$

[1]Pozornosť - ang. originál *Attention*



Obr. 2.1: Attention [23]

Matice W_i^Q, W_i^K, W_i^V z rovnice 2.3, nazývané aj váhové matice, patria medzi trénovateľné parametre modelu. Ich celkový počet je $3h$, kde h je vyššie uvedený počet častí MHA[2]. Počet týchto častí je nazývaný aj počet *attention heads*[3]. Architektúra MHA je znázornená na obrázku 2.2.



Obr. 2.2: Multi-head Attention [23]

[2]MHA - Multi-head Attention
[3]attention heads - hlavy

Enkóder

Enkóder je jedna z hlavných súčastí Transformera. Skladá sa z N identických vrstiev. V originálnej publikácii [23] bolo použitých $N = 6$ vrstiev, tento počet je však možné ľubovoľne meniť na základe implementačných potrieb.

Vrstva enkódera

Jedna vrstva enkódera sa skladá z 2 hlavných častí: *Multi-Head attention* a z doprednej neurónovej siete s 1 skrytou vrstvou. Každá z týchto častí pracuje nasledovne:

$$y = LayerNorm(x + Dropout(Function(x)) \tag{2.4}$$

kde x je vstup, y je výstup, $Function$ je samotná funkcia vykonávaná danou časťou enkódera, $Dropout$ je funkcia dropout[4], a $LayerNorm$ je normalizácia vrstvy[5].

Dopredná neurónová sieť je klasická plne prepojená sieť, ktorej skrytá vrstva má v originálnej práci [23] rozmer 4-krát väčší, ako jej vstup a výstup[6].

Dekóder

Dekóder je druhá hlavná časť Transformera. Podobne ako enkóder, skladá sa z rovnakého počtu indentických častí. Rovnako pre ňu platí aj rovnica 2.4. V čom je však odlišná, je zloženie jednej vrstvy. Namiesto 2 častí obsahuje 3 časti, konkrétne jednu MHA naviac.

Prvá z dvoch MHA využíva maskovanie z dôvodu, aby dekóder videl len na doposiaľ predpovedaný výstup. Druhá z nich je klasická MHA, avšak namiesto duplikovania vstupu na 3 časti je namiesto 2 z nich, konkrétne $Query$ a Key použitý výstup enkódera. Tretia hodnota $Value$ je výstup predchádzajúcej MHA.

Posledná časť vrstvy dekódera je taktiež dopredná neurónová sieť, rovnako ako pri enkóderi.

Embedding a positional encoding

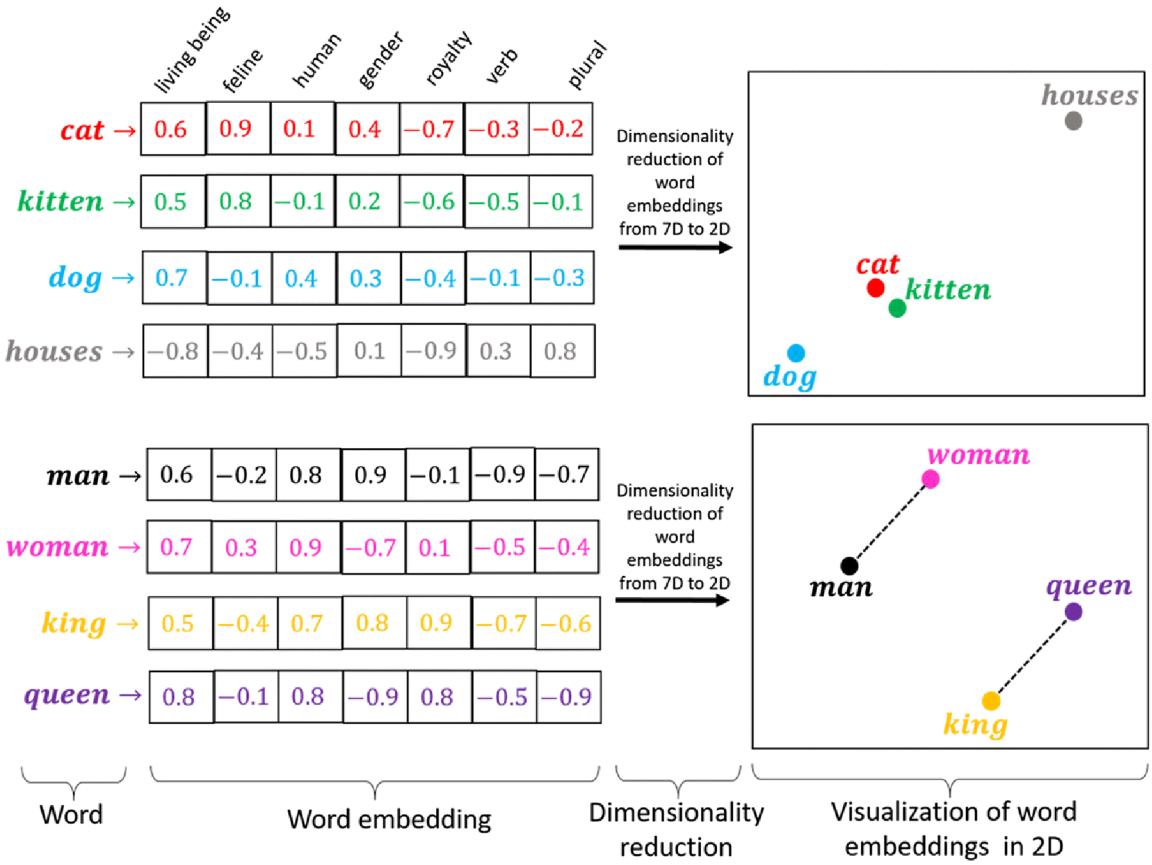
Pri vstupe do Transformera sú jednotlivé dáta sekvenciou slov, pričom každé slovo je reprezentované celočíselnou hodnotou v rozsahu rovnajúcemu sa definovanej veľkosti slovnej zásoby. Model však potrebuje slová reprezentovať iným spôsobom. Na to slúži *Embedding* [13], ktorý je vo svojej podstate jednoduchou vyhľadávacou tabuľkou, ktorá pre každé celočíselné označenie slova uchováva jeho vektorovú reprezentáciu. Tento vektor má rozmer rovnajúci sa d_{model}, pričom ide o vektor desatinných čísel. Podstatou takejto reprezentácie je uloženie slova do n-dimenzionálneho priestoru, pričom slová s podobným významom by mali byť blízko seba. Z globálneho hľadiska modelu je *Embedding* maticou o rozmere $d_{model} * vocab\_size$, ktorej hodnoty patria medzi trénovateľné parametre. Vizuálne znázornenie *Embeddingu* v menšej dimenzii je na obrázku 2.3.

Po transformácii vstupného slova na vektorovú reprezentáciu nasleduje positional encoding. Ide o maticu hodnôt, ktorá sa pripočíta ku sekvencii vektorových reprezentácií slov. Táto matica udáva slovám kontext na základe ich pozície vo vete. Počíta sa podľa vzorca:

$$\begin{aligned} PE_{(pos,2i)} &= \sin\left(pos/10000^{2i/d_{model}}\right) \\ PE_{(pos,2i+1)} &= \cos\left(pos/10000^{2i/d_{model}}\right) \end{aligned} \tag{2.5}$$

[4] dropout - náhodné zakrytie istej časti výstupu [20]
[5] normalizácia vrstvy - ang. originál Layer Normalization [2]
[6] V tomto prípade veľkosť vstupu aj výstupu - d_{model}

kde *pos* je pozícia slova vrámci vstupu, a i je index vo vektorovej reprezentácii slova.

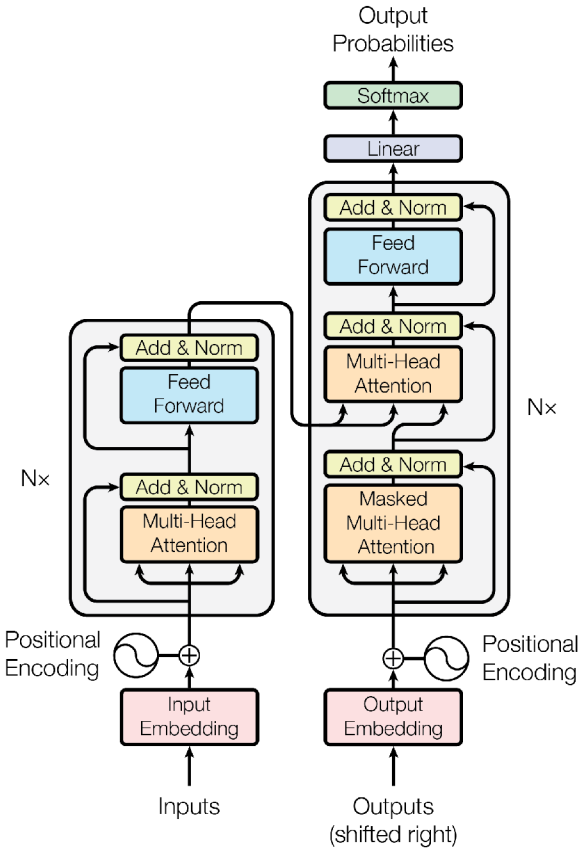


Obr. 2.3: Vizuálne znázornenie embeddingu [8]

Model

Celý model sa skladá z vyššie uvedených súčastí. Architektúra modelu je znázornená na obrázku 2.4. V trénovacej fáze je na vstup privedený zoznam slov (veta) ako zoznam celočíselných reprezentácii týchto slov vrámci preddefinovanej slovnej zásoby. Tento zoznam potom prejde *embeddingom*, ktorý ho transformuje na maticu o veľkosti $d_{model} * seq\_len$, kde seq_len je počet slov. K tejto matici sa potom pripočíta matica *positional encodingu*, ktorá má rovnaké rozmery. Táto matica potom vstúpi do enkódera.

V enkóderi táto matica prejde jeho N vrstvami, ktoré majú za úlohu zachytiť súvislosti slov medzi sebou na základe mechanizmu pozornosti. Výsledná matica je potom poslaná do dekódera. Dekóder najprv prevedie cieľový vstup na maticu vektorových reprezentácií rovnako ako enkóder. Táto matica rovnako prejde všetkými vrstvami dekódera s tým, že v každej vrstve je kombinovaná s výstupom enkódera, ako bolo opísané vyššie. Táto vlastnosť je veľmi dôležitá, keďže sa v nej model učí súvislosti medzi vstupom a cieľovým výstupom. Po výstupe z dekódera je finálna matica spracovaná plne prepojenou vrstvou, ktorej výstup má rozmer $seq\_len * vocab\_size$. Potom je na tento výstup aplikovaná funkcia *softmax* [9], ktorá transformuje logaritmické rozloženie na klasické rozloženie pravdepodobnosti. Toto rozloženie vyjadruje pravdepodobnosť dalšieho nasledujúceho slova vo výstupnej sekvencii.



Obr. 2.4: Architektúra Transformera [23]

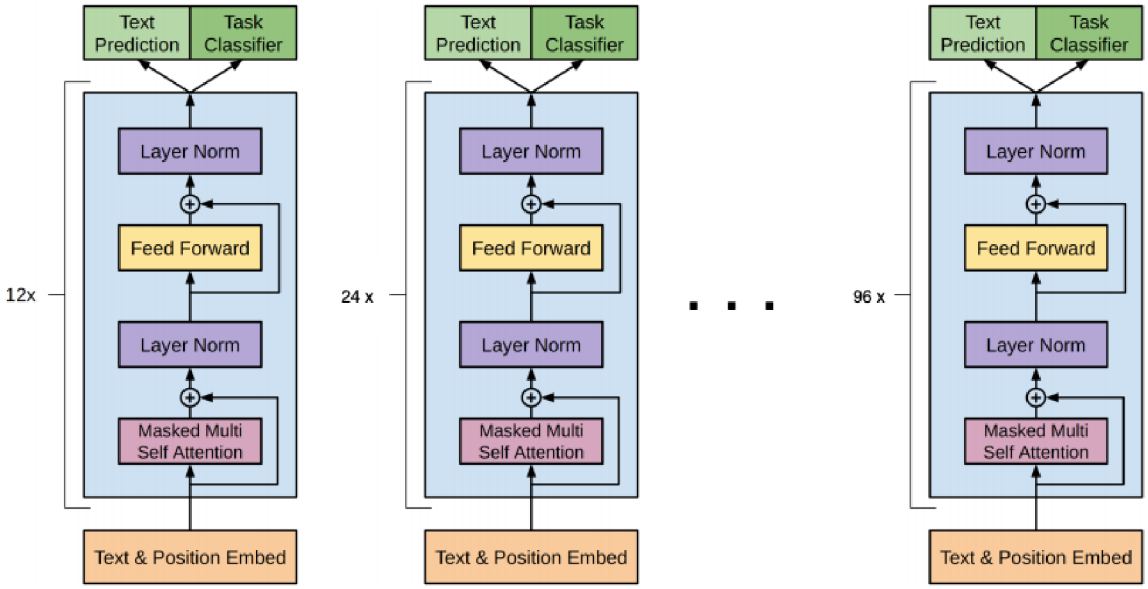
2.2 POS tagging

POS tagging, alebo Part-Of-Speech tagging je proces rozkladu vety na slovné druhy. Existujú mnohé databázy, takzvané stromové banky, ktoré pozostávajú z viet rozložených na slovné druhy. Jednou z nich je aj Penn Treebank, ktorá obsahuje okolo 3 miliónov viet rozložených na syntaktické stromy pomocou slovných druhov [22].

2.3 GPT

GPT, alebo Generative Pre-Trained Transformer je typ Transformera, ktorý je určený na generovanie textu. Od klasického modelu transformera sa odlišuje tým, že neobsahuje dekóder, len vrstvu enkóderov napojených za sebou. Jedným z najnovších a najväčších modelov v tejto kategórii je GPT-3 [4], ktorý dosiahol veľmi dobré výsledky vo viacerých úlohách, či už generovanie textu, alebo odpovedanie na otázky v prirodzenom jazyku. Na obrázku 2.5 je znázornená architektúra viacerých GPT modelov. Úplne napravo je model GPT-3, ktorý obsahuje až 96 vrstiev enkóderov.

[7]Zdroj: https://dzlab.github.io/assets/2020/07/20200725-gpt3-model-architecture.textpng



Obr. 2.5: Architektúra modelov GPT[7]

2.4 Metriky v strojovom učení

Stratová funkcia

Stratová funkcia[8] je funkcia, ktorá sa používa pri strojovom učení na výpočet chyby modelu. Cieľom trénovania modelu je potom minimalizácia tejto chyby. Spomedzi mnohých v tejto práci používam tzv. categorical cross-entropy loss, ktorej výpočet je uvedený vo vzorci[9] 2.6.

$$Loss = -\sum_{i=1}^{\text{output size}} y_i * log\hat{y}_i \tag{2.6}$$

V tomto vzorci je $\hat{y}_i$ i-ty člen vektora výstupu modelu, y_i je i-ty člen vektora skutočného výstupu, a *output size* je dĺžka vektora výstupu

Presnosť

Výpočet presnosti modelu je dôležitý pre informáciu o stave tréningu modelu. V tejto práci som použil jednoduchý výpočet presnosti, ktorý porovná predpovedané a skutočné tokeny. Na základe toho potom určí, aká časť z nich sa rovná, a to je výsledok výpočtu presnosti. Kód tejto funkcie je znázornený na obrázku 2.6.

2.5 Tokenizácia

Tokenizáciou sa rozumie rozdelenie textu na slová, tzv. tokeny. Táto metóda je nevyhnutná pre použitie sekvenčných jazykových modelov. Existuje viacero prístupov k tokenizácii.

[8]angl. originál loss function

[9]Zdroj: https://peltarion.com/knowledge-center/documentation/modeling-view/build-an-ai-model/loss-functions/categorical-crossentropy

```
249  def accuracy_function(real, pred):
250          accuracies = tf.equal(real, tf.argmax(pred, axis=2))
251
252          mask = tf.math.logical_not(tf.math.equal(real, 0))
253          accuracies = tf.math.logical_and(mask, accuracies)
254
255          accuracies = tf.cast(accuracies, dtype=tf.float32)
256          mask = tf.cast(mask, dtype=tf.float32)
257          return tf.reduce_sum(accuracies)/tf.reduce_sum(mask)
258
```

Obr. 2.6: Kód zodpovedný za výpočet presnosti. Porovnanie predpovedaných a skutočných tokenov sa deje na riadku 250. Potom sa aplikuje maska, ktorá vynechá z porovnania tzv. *padding* tokeny. Na riadku 257 sa potom vydelí suma správnych predpovedí s celkovou sumou.

Rozdelenie textu na slová

Najjednoduchšou formou tokenizácie je rozdelenie textu na slová podľa medzier, alebo regulárnych výrazov. Táto technika funguje dobre pre malý obsah dobre štruktúrovaného textu. Problém nastáva pri väčších obsahoch textu. Týmto spôsobom vznikne príliš veľa rozdielnych slov, čím sa zväčšuje veľkosť slovnej zásoby. To výrazne zvyšuje výpočetné národky pri trénovaní modelu.

Rozdelenie textu na znaky

Ďalšou z foriem tokenizácie je rozdelenie textu na jednotlivé znaky. Výhodou je malá veľkosť slovnej zásoby a jednoduchá implementácia. Hlavnou nevýhodou je príliš veľká dĺžka jednej vety, čo znova zvyšuje výpočetnú náročnosť jazykových modelov. Ďalším z mínusov je aj to, že model sa nenaučí súvislosti medzi jednotlivými slovami, keďže každé slovo je len jeden znak.

Rozdelenie textu na podslová

Lepšou formou tokenizácie je rozdelenie textu na podslová. V prirodzenom jazyku je bežné, že slová vznikajú spojením viacerých slov. Tento fakt využíva práve tátp metóda. Ušetrí sa ako veľkosť slovnej zásoby, tak aj dĺžka vety, a navyše sa model lepšie naučí súvislosti aj medzi jednotlivými časťami slov.

Byte-Pair Encoding

Existuje viacero prístupov k rozdeleniu textu na podslová. Jedným z nich je Byte-Pair Encoding [7], skrátene BPE. Spočíva v tom, že sa postupne vezmú najviac používané dvojice znakov, a sú nahradené novým znakom. Tento proces prebieha rekurzívne, až pokiaľ sa nedôjde k požadovanej veľkosti slovnej zásoby. Túto metódu používa aj tokenizér SentencePiece [16], ktorý je použitý v tejto práci. Ukážka pseudokódu prebratá z práce [7] je na obrázku 2.7.

Listing 1 Compression algorithm (pseudocode)

```
While not end of file
   Read next block of data into buffer and
      enter all pairs in hash table with counts of their occurrence
   While compression possible
      Find most frequent byte pair
      Replace pair with an unused byte
      If substitution deletes a pair from buffer,
         decrease its count in the hash table
      If substitution adds a new pair to the buffer,
         increase its count in the hash table
      Add pair to pair table
   End while
   Write pair table and packed data
End while
```

Obr. 2.7: Pseudokód tokenizačnej metódy BPE

2.6 Vzorkovacie heuristiky

Jazykové modely sa vo všeobecnosti snažia predpovedať ďalšie slovo v sekvencii. Deje sa to tak, že pre každú sekvenciu vrátia zoznam všetkých slov zo slovnej zásoby s príslušnými pravdepodobnosťami, že dané slovo je nasledujúce. Na spôsob samotného výberu poznáme viacero algoritmov, resp. heuristík.

Greedy search

Greedy search [3] je algoritmus, ktorý vždy vyberie najlepšie lokálne riešenie. Pri jazykových modeloch to znamená, že pri každej predpovedi ďalšieho slova sa vyberie to s najvyššou pravdepodobnosťou. Ide o najtriviálnejšiu vzorkovaciu heuristiku pri jazykových modeloch.

Beam search

Beam search [6] je algoritmus, ktorý prehľadáva stavový priestor ponechaním N najlepších možností, kde N nazývame šírkou lúča[10]. Skóre sa počíta pre každý lúč od začiatku do konca. Tento algoritmus som upravil na zmenšenie pamäťovej aj časovej náročnosti. V praxi to znamená, že v každej iterácii jazykového modelu sa vyberie N slov s najvyššími pravdepodobnosťami. V originálnom algoritme by sa vybrala celá slovná zásoba, to by však bolo príliš náročné. Tieto slová sú potom konkatenované ku súčasným N lúčom, čím dostaneme N^2 možností. Z týchto možností sa ponechá len N s najlepším skóre, a proces sa opakuje. Skóre lúča sa počíta ako súčin pravdepodobností každého slova v tomto lúči. Vyšší súčin znamená lepšie skóre.

[10] angl. beam width

Top-K

Top-K [10] je algoritmus, ktorý pri každwj iterácii jazykového modelu vyberie K slov s najvyšíími pravdepodobnosťami. Z týchto slov náhodným výberom vyberie jedno, pridá ho do sekvencie, a proces sa opakuje. Týmto spôsobom sa snaží predísť prediktívnemu a repetitívnemu chovaniu jazykového modelu.

2.7 Softmax

Softmax [9] je funkcia, ktorá premieňa vektor hodnôt v rozsahu celých reálnch čísel na vektor hodnôt v rozsahu od 0 po 1, pričom súčet čísel tohto vektora je 1. Táto funkcia sa využíva v strojovom učení, kde premieňa výsledné skóre modelu na rozloženie pravdepodobnosti. Počíta sa podľa vzorca [26]:

$$\sigma(\vec{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \tag{2.7}$$

kde $\vec{z}$ je vstupný vektor, K je dĺžka vstupného vektora, z_i a z_j sú i-ty a j-ty prvok vektora $\vec{z}$, a e je Eulerovo číslo.

Kapitola 3

Zhrnutie doterajšieho postupu

Generovanie kódu je pomerne nová disciplína v oblasti spracovania prirodzeného jazyka a strojového učenia. Napriek tomu už je to pár rokov, čo sa rôzni výskumníci zaoberajú touto témou a jej použitím v praxi. Dá sa povedať, že ide o akýsi logický postup z oblasti generovania sumarizácie a komentárov z poskytnutého kódu. Táto úloha vyzerá intuitívne jednoduchšia. Má však s generovaním kódu veľa spoločného.

V práci [11] sa venujú tejto tematike. Používajú rozklad kódu na AST, ktorý je potom serializovaný. Následne je pomocou poskytnutých anotácií natrénovaný model, ktorý sa formou strojového prekladu snaží nájsť súvislosť medzi kódom a komentárom (anotáciou).

Problematike generovania či doplňovania kódu sa poslednej dobe venuje veľa pozornosti, najmä v súvislosti s GPT-3. Viacero článkov, ako napríkad aj článok [5] získalo pozornosť širšieho publika tým, ako demonštrovali schopnosti modelu GPT-3 generovať jednoduchý kód podľa textového popisu. Toto spustilo záujem o použitie GPT modelov, ako aj Transformerov všeobecne, na generovanie či doplňovanie kódu.

Model typu GPT bol použitý napríklad v práci [21], kde bolo použitých 1,2 miliardy riadkov kódu v rôznych programovacích jazykoch na trénovanie modelu. Úlohou bolo vytvoriť systém na doplňovanie kódu. V tomto prípade nebol použitý rozklad na AST, model však dosiahol veľmi dobré výsledky. Pochopiteľne, jednalo sa o model s veľkým počtom parametrov, konkrétne 366 miliónov. Takéto modely sú extrémne výpočtovo náročné na tréning aj nasadenie.

Mnoho z pokusov o generovanie kódu má spoločnú jednu vec, a tou je rozloženie kódu na syntaktický strom (AST). Jedná sa napríklad o prácu [27], kde bol použitý model typu RNN [19] na predpovedanie štruktúry výsledného kódu.

Ďalšou z používaných metód syntaktického rozkladu kódu je rozklad AST na graf. Tento prístup bol použitý v práci [24], kde navrhli tzv. *AST Graph Attention Block*. Ide o modul, ktorý je podobný *Attention* bloku v Transformeri s tým rozdielom, že spracúva grafy.

V práci [17] bol zvolený trochu odlišný prístup ku doplňovaniu kódu. Rovnako ako u iných prác bol použitý rozklad kódu na AST. Rozdiel bol v tom, že v tejto práci boli použité 2 rozdielne neurónové siete. Princíp bol v tom, že na generovanie syntaktických tokenov bola použitá sieť typu *LSTM* [19], a na generovanie slovných tokenov bola použitá tzv. *Pointer Mixture Network*, ktorá bola definovaná v tejto práci.

Poslednou prácou spomenutou v tejto kapitole je práca [15], ktorá sa venuje návrhu a implementácii systému na doplňovanie kódu. V tejto práci bol taktiež použitý rozklad kódu na AST s tým, že sa experimentovalo s rôznymi metódami serializácie AST do sekvencie. Použitý dataset bol *py150* [18], ktorý obsahuje spracované AST pre každý zo 150 tisíc súborov v jazyku Python.

Kapitola 4

Dáta

Pre natrénovanie modelu bolo potrebné použiť kvalitný dataset. Úlohou výsledného modelu je predpovedať kód v jazyku Python na základe poskytnutého popisu, príp. hlavičiek funkcií. Existuje niekoľko potenciálnych zdrojov takýchto dát. Jedným z nich je kvalitne dokumentovaný kód, druhým sú napr. Jupyter notebooky.

4.1 Dostupné datasety

Vzhľadom na fakty uvedené vyššie prišlo do úvahy viacero datasetov. Jedným z nich bol dataset *CodeSearchNet* [12], ktorý obsahuje kolekciu implementácií funkcií a ich dokumentácie. Tento dataset bol vytvorený zbieraním verejne dostupných zdrojových kódov a obsahuje cez 2 milióny párov funkcií a dokumentačných reťazcov.

Ďalší z kandidátov na použiteľný dataset bol dataset JuICe [1], ktorý pozostáva z dát pozbieraných z cca. 673 tisíc verejne dostupných Jupyter notebookov. Výsledkom je cca. 1,5 milióna kusov kódu, pričom ku každému z nich je priradený jeho kontext. Každý kus kódu je obsah nejakej bunky v notebooku, a kontext je obsah viacerých buniek pred ním. Tieto bunky sú zmes kusov kódu a textových popisov. Práve z tohto dôvodu som sa rozhodol tento dataset nepoužiť, keďže na jeden cieľový kus kódu sa mapuje viacero popisov a iných kusov kódu, čo sťažuje praktické použitie takéhoto modelu.

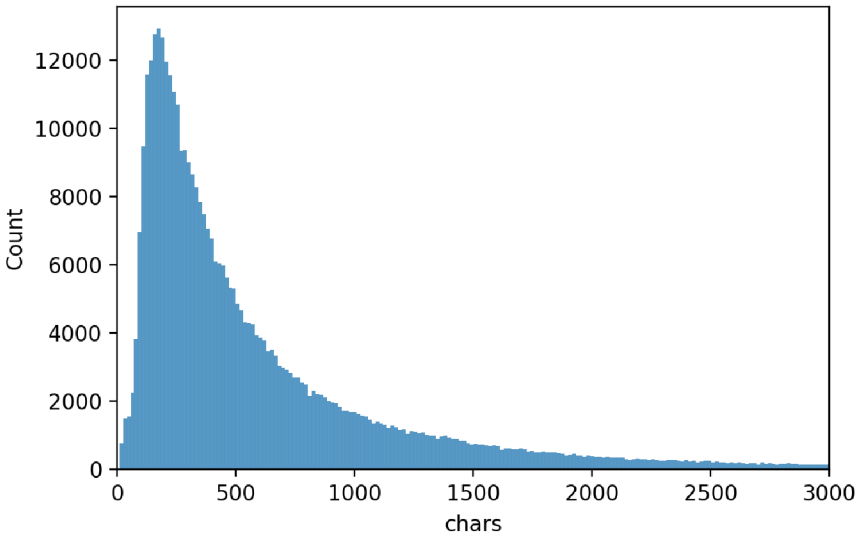
4.2 CodeSearchNet

CodeSearchNet je dataset obsahujúci kódy v 6 programovacích jazykoch.[1] V tejto práci sa zameriavam len na kód napísaný v programovacom jazyku Python. Hlavným dôvodom je, že Python je najviac "prirodzenýä najviac populárny programovací jazyk, čo znamená, že pri úlohách ako napr. automatické doplňovanie kódu dosiahol spomedzi iných jazykov najlepšie výsledky. [21] Súhrnné štatistiky tohoto datasetu sú v tabuľke 4.1. Histogramy dĺžky kódu a dokumentačných reťazcov v počte znakov sú na obrázkoch 4.1, resp. 4.2.

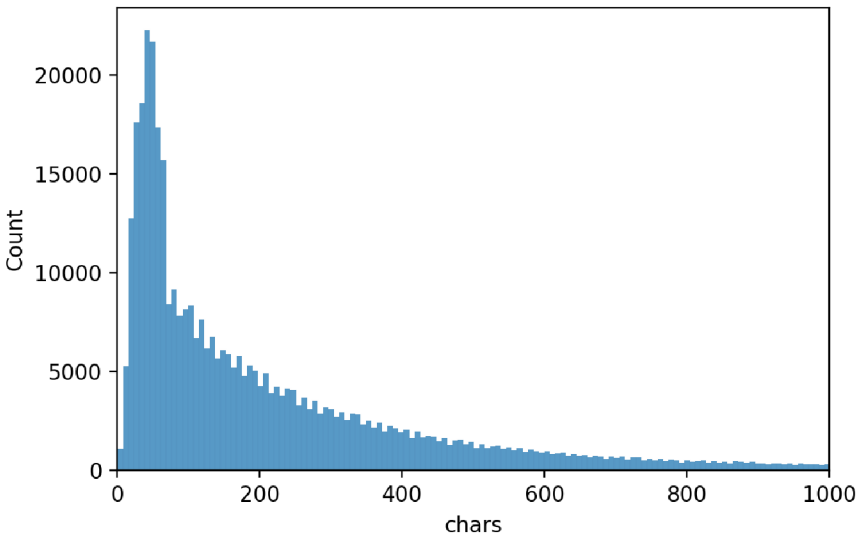
[1] Konkrétne Python, Java, PHP, Javascript, Go, Ruby

Tabuľka 4.1: Štatistiky datasetu CodeSearchNet pre jazyk Python. Počet slov je daný po rozdelení textu ľubovoľným počtom bielych znakov.

	Kód	Dokumentačné reťazce
Počet dátových vzoriek	412178	412178
Priemerný počet riadkov	17,06	6,19
Priemerný počet slov	59,15	36,71
Priemerný počet znakov	730	295,01



Obr. 4.1: Histogram dĺžky kódu v počte znakov



Obr. 4.2: Histogram dĺžky dokumentačných reťazcov v počte znakov

Kapitola 5

Návrh riešenia

V tejto práci som vypracoval konkrétne 2 modely, v návrhu a implementácii sa však venujem len tomu druhému. Prvý model bol takmer identický s druhým, s jediným rozdielom v jeho veľkosti a v tom, že bolo použité syntaktické spracovanie kódu. Pri prvom bol na tokenizáciu aj detokenizáciu použitý len BPE tokenizér, a dataset bol len prefiltrovaný od príliš dlhých vstupov, čím sa zachovalo viac ako 90% pôvodného datasetu.

5.1 Spracovanie dát

Dáta sú z pôvodného zdroja stiahnuté v ZIP formáte, je potrebné ich najprv extrahovať. Obsah tohto archívu pozostáva z viacerých JSONL[1] súborov. Každý z týchto súborov je potrebné načítať riadok po riadku a uložiť do poľa. Každý z týchto riadkov reprezentuje jeden dátový bod.

Každý dátový bod obsahuje niekoľko položiek. Pre túto prácu budú stačiť 2 z nich: *code* a *docstring*. Tieto reprezentujú originálny text obsahujúci celý kód (vrátane dokumentačného reťazca), resp. samotný dokumentačný reťazec. Je teda potrebné odstrániť z kódu samotný dokumentačný reťazec. Ďalej nasleduje odstránenie dátových bodov, pri ktorých sa dĺžka kódu nachádza mimo isté určené hodnoty. Podľa obrázku 4.1 vidíme, že väčšina kódu je kratšieho rozsahu, teda neprídeme o príliš veľkú časť datasetu. Tento krok je potrebný, aby sme zvýšili presnosť modelu, keďže je lepšie mať dátovú sadu v čo najviac uniformnom formáte. Navyše, pri veľmi krátkych alebo veľmi dlhých úsekoch kódu je väčšia šanca, že sa jedná o nekvalitné dáta, keďže pri takejto veľkosti datasetu nie je možné mať na 100% čisté dáta.

Cieľom filtrovania dokumentačných reťazcov je získanie krátkych, jednoduchých, výstižných, a ideálne jednovetových popisov kódu funkcií v anglickom jazyku. To samozrejme nie je úplne triviálna úloha, keďže tieto reťazce obsahujú často aj iný text, napr. príklady použitia, alebo popis parametrov. Po získaní nadhľadu nad formátom týchto reťazcov som usúdil, že bude najlepšie ponechať len prvý riadok u každého z nich, keďže ten často obsahoval práve žiadanú formu textového popisu. Ukážka formy niektorých dokumentačný reťazcov je na obrázku 5.1. Niekedy však tento popis chýbal, a často od prvého riadku začínal formálny popis parametrov. Väčšina z týchto popisov sa držala formátov reStructuredText[2] a Epytext[3], čo sa dá ľahko využiť. Preto som navrhol systém, ktorý pomocou

[1] https://jsonlines.org/
[2] https://www.python.org/dev/peps/pep-0287/
[3] http://epydoc.sourceforge.net/epytext.html

regulárneho výrazu určí, či sa prvý riadok podobá na jeden z týchto formátov. Ak áno, tak tento dokumentačný reťazec je vynechaný z datasetu. Celý tento proces je znázornený na obrázku 5.2. Druhá časť spracovania dát pred vstupom do samotného modelu pozostáva z prevodu textov na sekvencie tokenov. Tejto problematike sú venované nasledujúce kapitoly.

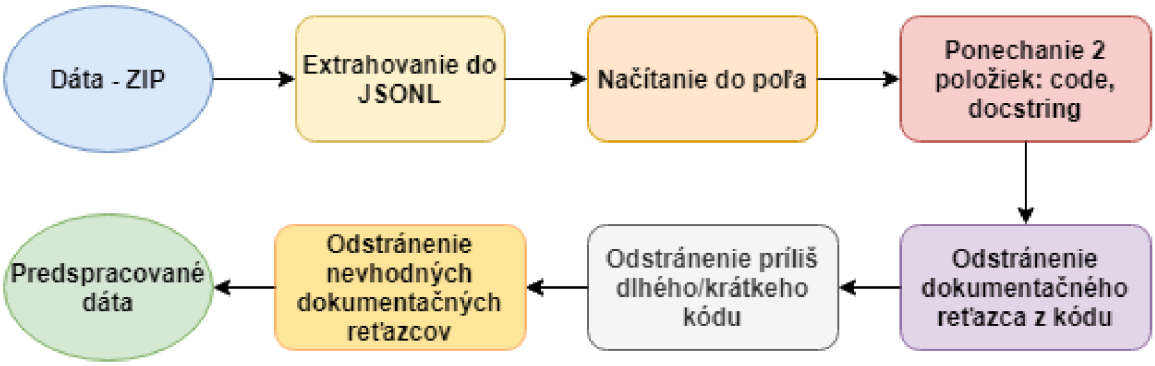
```
-----------------------------------------------
Shows the face recognition results visually.

    :param img_path: path to image to be recognized
    :param predictions: results of the predict function
    :return:
-----------------------------------------------
Convert a dlib 'rect' object to a plain tuple in (top, right, bottom, left) order

    :param rect: a dlib 'rect' object
    :return: a plain tuple representation of the rect in (top, right, bottom, left) order
-----------------------------------------------
Make sure a tuple in (top, right, bottom, left) order is within the bounds of the image.

    :param css:  plain tuple representation of the rect in (top, right, bottom, left) order
    :param image_shape: numpy shape of the image array
    :return: a trimmed plain tuple representation of the rect in (top, right, bottom, left) order
-----------------------------------------------
```

Obr. 5.1: Ukážka formy dokumentačných reťazcov



Obr. 5.2: Predspracovanie dát

5.2 Syntaktické spracovanie vstupu

Jazykové modely ako napr. Transformer sa využívajú na spracovanie sekvencií slov. Pri prirodzenom jazyku vykazujú vysokú presnosť. Čo sa týka spracovania kódu programovacích jazykov, viacero prác ukazuje, že model dosiahne lepšie výsledky, ak sa mu nejakým spôsobom dodajú informácie o syntaxi vstupného kódu. Prehľad týchto prác je v kapitole 3.

V tejto práci som sa zameral ako na syntaktické spracovanie kódu, tak aj dokumentačných reťazcov. Transformovať kód na syntaktický strom nie je zložité, existuje viacero knižníc, ktoré to dokážu. Rovnako existujú aj knižnice pre rozklad viet v anglickom jazyku na syntaktický strom. Problém je v tom, že Transformer je sekvenčný model, jeho vstupom aj výstupom je sekvencia slov, nie stromová štruktúra. Je teda potrebné tieto syntaktické stromy transformovať na sekvencie, ktoré model dokáže spracovať. Navyše, v prípade kódu musí tento proces fungovať aj opačne, keďže konečným výstupom modelu má byť čistý text.

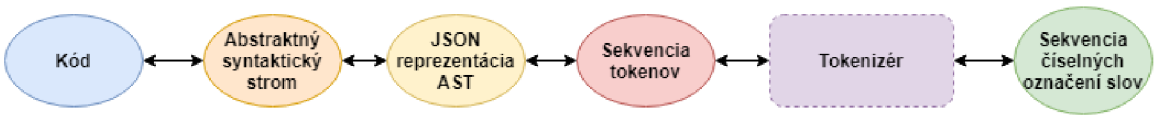
Problematike prevodu stromu na sekvenciu sa venovali aj v práci [27], kde použili metódu pre-order prechodu stromom. Rozhodol som sa túto metódu použiť aj v mojej práci. Narozdiel od ostatných prístupov som sa rozhodol použiť túto stratégiu aj pri prevode vstupného textu. Moja domnienka je, že model bude schopný nájsť súvislosť medzi syntaxou vety a kódu.

Spracovanie kódu

Kód v textovej podobe musí byť najprv prevedný na abstraktný syntaktický strom [14]. Následne tento strom musí byť prevedený do sekvenčnej formy. Keďže to nie je úplne triviálna úloha, bude existovať ešte jeden mezikrok, ktorý prevedie abstraktný syntaktický strom na JSON[4] reprezentáciu, z ktorej sa následne vygeneruje sekvencia tokenov. Detailnejšej implementácii sa budem venovať v kapitole 6. Následne je potrebné analogicky implementovať systém, ktorý zo sekvencie tokenov vyprodukuje najprv JSON reprezentáciu, následne syntaktický strom a potom samotný kód. Princíp tejto procedúry je znázornený na obrázku 5.3.

Spracovanie dokumentačných reťazcov

Dokumentačné reťazce musia byť najprv skonvertované na syntaktický strom, podobne ako aj kód. Prirodzený jazyk ale nemá tak zložitú štruktúru ako kód, a preto môžeme previesť jeho syntaktický strom priamo na sekvenciu tokenov. Opačný prevod potom už nie je potrebný. Princíp tejto procedúry je znázornený na obrázku 5.4.



Obr. 5.3: Prevod kódu na sekvenciu tokenov a naopak

Obr. 5.4: Prevod dokumentačného reťazca na sekvenciu tokenov

Prevod syntaktického stromu na sekvenciu

Syntaktický strom, podobne ako každá iná stromová štruktúra, obsahuje 2 typy uzlov: *vnútorné* a *listové*. Vnútorné uzly v prípade syntaktického stromu reprezentujú jednotlivé štruktúry, z ktorých pozostáva text, z ktorého bol tento strom vytvorený. Listové (koncové) uzly sú v tomto prípade atomické jednotky textu, ktoré z hľadiska syntaxe nie je možné ďalej deliť. Každý z vnútorných uzlov má určený typ, a každý z koncových uzlov reprezentuje literál (napr. slovo v prípade dokumentačných reťazcov, alebo názov premennej, číslo v prípade kódu).

[4] https://www.json.org/json-en.html

Čo sa týka vnútorných uzlov, všetky ich typy sú predom dané, takže je možné každému z nich priradiť jedno číslo. Sekvencia tokenov kódu však musí byť spätne preveditelná na strom, preto musíme štruktúru stromu istým spôsobom zakódovať do sekvencie. Keďže sa jedná o pre-order prechod, postačí na to jednoduchý algoritmus. Začína na koreňovom uzle stromu:[5]

1. Spracuj súčasný uzol
2. Ak má uzol nespracovaného potomka, zaznač ho ako spracovaného, prejdi na neho a choď na krok 1
3. Pridaj do sekvencie token *UP*
4. Ak má uzol predchodcu, prejdi na neho a choď na krok 2
5. Koniec

Token *UP* je špeciálny token, ktorým sa značí návrat v strome o úroveň vyššie. Princíp tohoto prevodu je ukázaný na obrázku 5.5. Prevod opačným smerom je potom už jednoduchý, zo sekvencie vygenerovanej týmto spôsobom je možné jednoznačne spätne vytvoriť strom.

Tokenizácia sekvencie

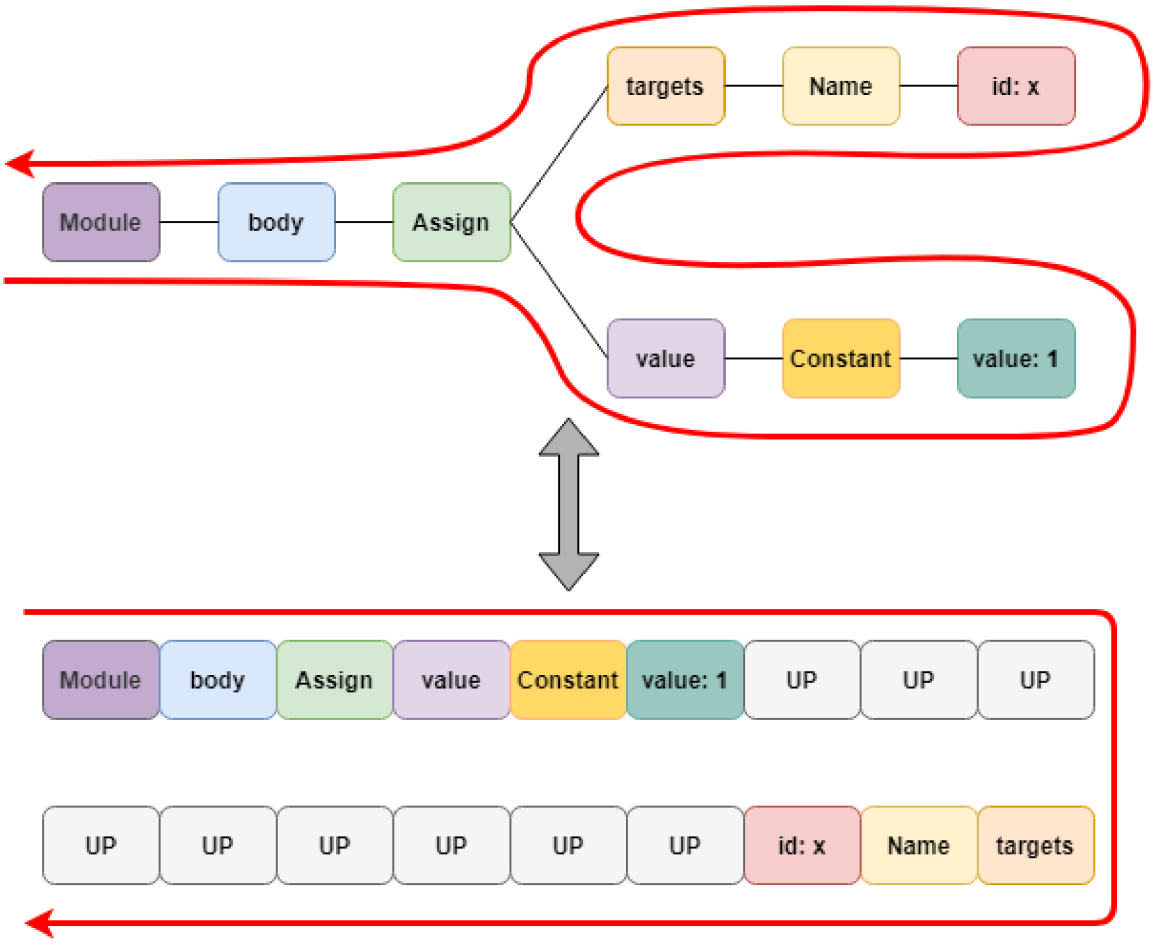
Výsledkom syntaktického spracovania budú sekvencie tokenov ako pre kód, tak pre dokumentačný reťazec. Model však potrebuje dostať sekvenciu celých čísel, ktoré reprezentujú jednotlivé slová. O mapovanie tokenov na čísla sa postará tokenizér, ktorého funkcia bude opísaná v podkapitolách 2.5 a 6.2. V tejto kapitole je zatiaľ vnímaný ako tzv. *black-box* (viď obrázky 5.3 a 5.4).

5.3 Tokenizér

Hlavnou úlohou tokenizéra v jazykových modeloch je prevod textu na sekvenciu tzv. tokenov. Každý z týchto tokenov je celé číslo v rozsahu veľkosti slovnej zásoby modelu. Toto rozloženie slovnej zásoby je lepšie znázornené na obrázku 6.15 v podkapitole 6.3. Pri tradičných jazykových modeloch sa nejakou z dostupných metód rozdelí text na jednotlivé slová (tokeny), pričom každému z nich je priradené jedno číslo.

V prípade tejto práce však rozlišujeme 2 typy tokenov. Jedným z nich je token reprezentujúci vnútorný uzol syntaktického stromu, ktorý má jednoznačne priradené svoje číselné označenie typu. Druhým z nich je token reprezentujúci koncový uzol stromu, ktorý už priradený typ nemá. Na základe týchto skutočností je potrebné navhrnúť štruktúru, ktorá reprezentuje jeden token. Pri vnútorných uzloch potrebujeme uložiť číslo jeho typu, pri koncových uzloch zasa jeho textový obsah. Číselné označenie koncového uzla (tokenu) môže byť špeciálna konštanta *WORD*, ktorá značí, že sa jedná o koncový uzol. Takýto token bude predaný ďalej tokenizéru, ktorý na jeho miesto vloží sekvenciu 1 až N tokenov. Ukážka rôznych tokenov je na obrázku 5.6.

[5] Spracovaním uzlu sa myslí jeho pridanie do výslednej sekvencie



Obr. 5.5: Prevod medzi syntaktickým stromom a sekvenciou tokenov

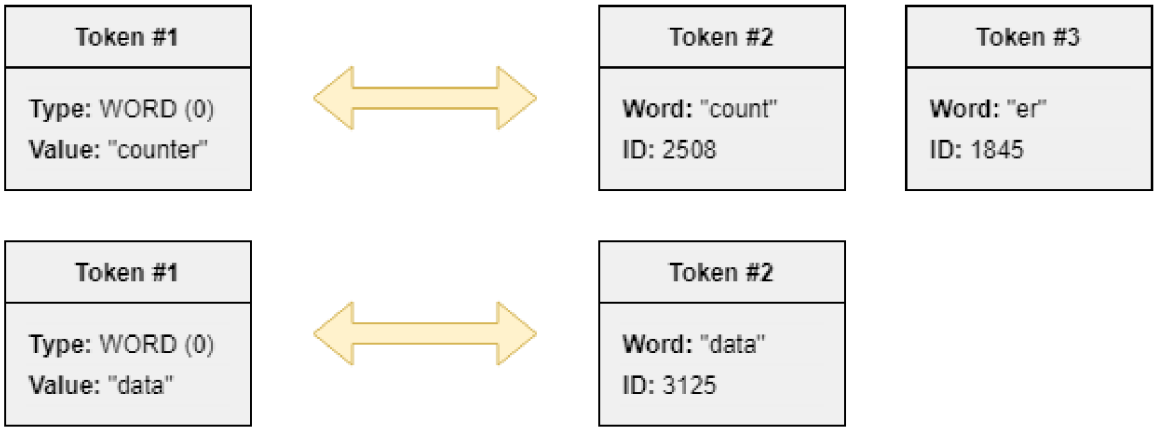
Rozdelenie slov na tokeny

Pri tokenoch typu *WORD* nevieme určiť ich typ. Na priradenie celočíselnej hodnoty teda musíme využiť iný postup. Tu prichádzajú na rad tradičné metódy tokenizácie, ktoré sú využívané v jazykových modeloch. Podstatou týchto úloh je rozčlenenie textu na slová, a očíslovanie každého slova. Poznáma viacero takýchto metód. Najprimitívnejšie z týchto metód rozdeľujú slová na základe bielych znakov. Trochu pokročilejšou metódou je BPE, ktorá sa vyznačuje tým, že umožňuje každé slovo rozdeliť na viacero podslov, čím sa dosiahne vyššie pokrytie slovnej zásoby, a zároveň to dokáže spraviť s menšou veľkosťou slovnej zásoby. Toto je lepšie opísané v podkapitole 2.5.

Token #1	Token #2	Token #3	Token #4
Type: Call (9) **Value:** None	**Type:** Return (15) **Value:** None	**Type:** WORD (0) **Value:** "counter"	**Type:** UP (1) **Value:** None

Obr. 5.6: Ukážka rôznych tokenov

Príklad použitia tejto metódy je na obrázku 5.7. Po spracovaní stromu na sekvenciu budú tokeny typu *WORD* nahradené 1 až *N* tokenmi. O toto rozdelenie sa postará tokenizér s využitím BPE metódy. Z toho vyplýva, že celá slovná zásoba pozostáva z viacerých častí. Prvou z nich sú špeciálne tokeny, ktoré potrebuje samotný tokenizér. Jedná sa o tokeny, ako napr. *PAD*, *UNK*, alebo špeciálne znaky ako nový riadok, tabulátor, atď. Druhou časťou je token *UP*. Treťou časťou sú tokeny, ktoré sa vzťahujú na typ uzla v syntaktickom strome. Štvrtou a poslednou časťou sú slovné tokeny, teda tie, ktoré tokenizér vyprodukuje pri spracovaní *WORD* tokenov. Rozdelenie slovnej zásoby je názorne ukázané na obrázku 5.8. *N* vyznačuje veľkosť slovnej zásoby.



Obr. 5.7: Príklad tokenizácie

Obr. 5.8: Rozdelenie slovnej zásoby

5.4 Model

Najdôležitejšou súčasťou celého systému je samotný jazykový model. Spomedzi rôznych jaykových modelov bude v tomto systéme použitý model typu Transformer, ktorý je podrobnejšie opísaný v podkapitole 2.1. Tento model pozostáva z 2 vstupov a jedného výstupu. Prvý zo vstupov (enkóder) slúži pre zdrojový jazyk, v našom prípade dokumentačné reťazce. Druhý zo vstupov slúži pre cieľový jazyk, v našom prípade kód.

Model bude použitý v 2 fázach: tréning a nasadenie. V tréningovej fáze prejdú obidva vstupy spracovaním na tokeny. Následne sa tieto tokeny predjú modelu, ktorý na základe toho poskytne výstup vo forme rozloženia pravdepodobnosti pre jednotlivé slová. To sa

následne použije na optimalizáciu parametrov modelu. Tento proces sa opakuje, a týmto sa model trénuje.

V druhej fáze je natrénovaný model použitý na predikciu písaného kódu. Na vstup enkódera sa rovnako umiestnia tokeny reprezentujúce dokumentačné reťazce. Na vstup dekódera sa umiestnia tokeny kódu, ktoré už boli vygenerované modelom. Na začiatku žiadny kód vygenerovaný nebol, preto sa na tento vstup umiestni špeciálny *START* token. Výstup potom prejde cez kontrolu syntaxe, ďalej sa vybranou metódou vyberie jedno z možných slov. Toto slovo je potom súčasťou výstupu, a zároveň sa pridá na vstup dekódera. Tento proces sa opakuje, kým sa buď nedosiahne maximálna dĺžka sekvencie, alebo sa neukončí syntaktický strom návratom ku koreňu. Ukážka integrácie modelu do celého systému je na obrázku 5.9.

5.5 Kontrola syntaxe

Model pri predikcii slov vyprodukuje rozloženie pravdepodobností pre každé slovo zo slovnej zásoby. Najjednoduchšou heuristikou je vybrať slovo s najvyššou pravdepodobnosťou. Pri klasickom jazykovom modele by to fungovalo. Problémom v prípade tejto práce je, že jednotlivé predpovedané slová zodpovedajú syntaktickým jednotkám textu, a musia byť spätne previeditelné na syntaktický strom. V praxi to znamená, že výber slov je obmedzený len na tie, ktoré dodržujú syntax.

Je potrebné navrhnúť systém, ktorý bude obmedzovať ďalšie možné slová (tokeny) na základe doterajších slov. Princíp je nasledovný:

1. Inicializácia zoznamu použiteľných slov
2. Model vygeneruje pravdepodobnosti pre všetky slová
3. Zo zoznamu použiteľných slov sa vyberie to s najvyššou pravdepodobnosťou
4. Vybrané slovo sa pošle systému, ktorý vygeneruje nový zoznam použiteľných slov
5. Ak je zoznam použiteľných slov prázdny, proces sa skončí
6. Vybrané slovo sa pošle modelu, a pokračuje sa krokom 2

Takýmto spôsobom bude prebiehať generovanie slova za slovom. Tieto slová potom prejdú procesom konverzie zo sekvencie tokenov na text, ktorý bol opísaný v predchádzajúcich sekciách.

Štruktúra systému

Hlavnou úlohou tohto systému bude pre danú sekvenciu doterajších slov vytvoriť zoznam slov, ktoré môžu nasledovať. Systém by mal fungovať interaktívne, tzn. po každom ďalšom vloženom slove vedieť poskytnúť zoznam povolených slov. Zároveň musí systém poznať gramatické pravidlá jazyka kódu.

Keďže systém si musí byť vedomý súčasnej pozície v syntaktickom strome, rozhodol som sa, že bude pracovať so zásobníkovou štruktúrou. Na tento zásobník by sa ukladali prichádzajúce slová. Ak by prichádzajúcim slovom bol token *UP*, tak by sa z vrcholu zásobníka odstránilo jedno slovo. Princíp činnosti tohoto systému je znázornený na obrázku 5.10.

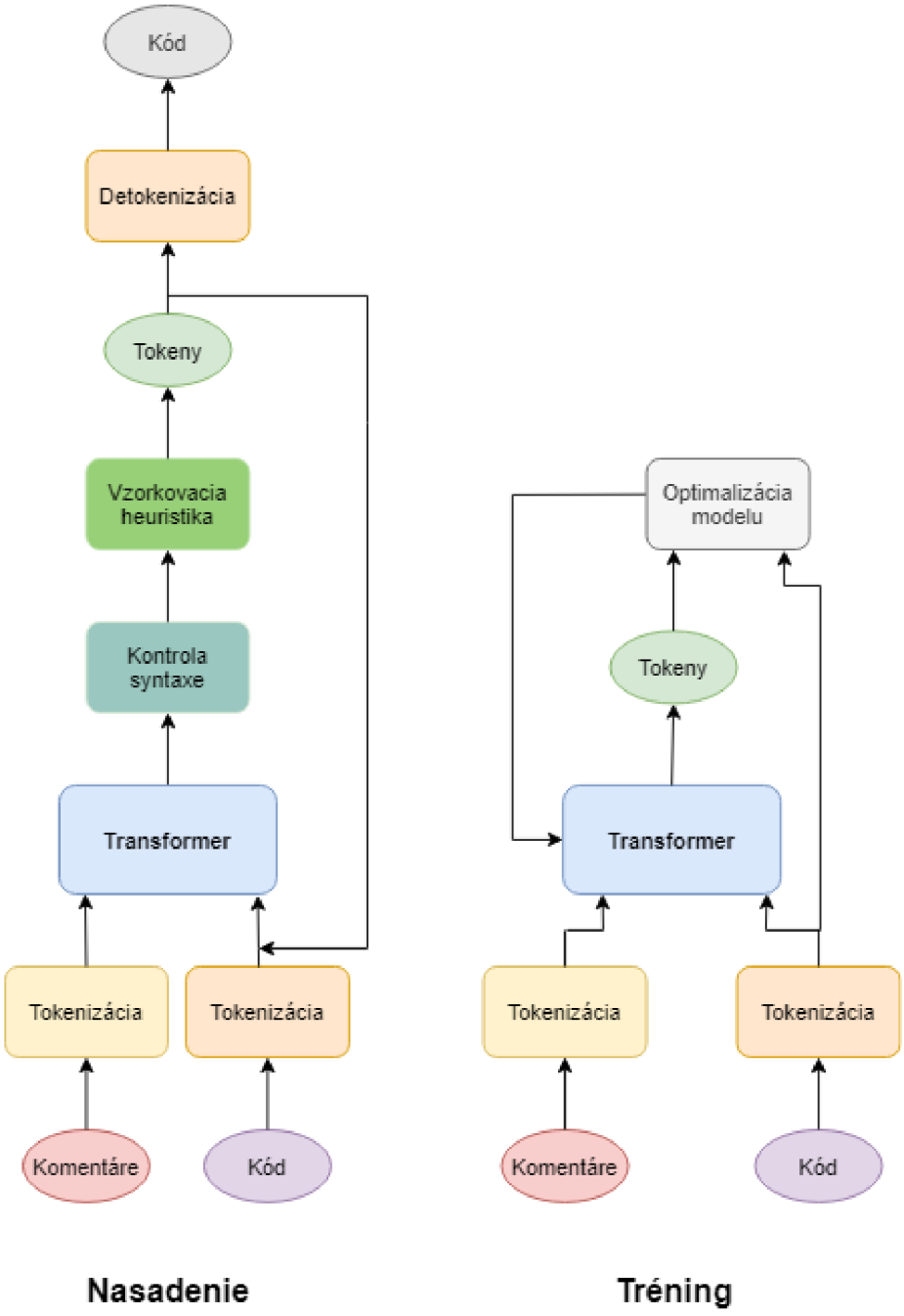
5.6 Vzorkovacie heuristiky

Pri jazykových modeloch získame pre danú sekvenciu slov číselné hodnoty, pričom každá z nich určuje pravdepodobnosť, že nasledujúce slovo bude to dané slovo. Najprimitívnejšia stratégia je jednoduchý výber slova s najvyššou pravdepodobnosťou. V tejto práci budem implementovať túto jednoduchú variantu, rovnako aj Top-K a Beam search, a budem porovnávať ich úspešnosť, ako aj časovú zložitosť. Heuristiku Beam Search som z hľadiska časovej náročnosti trochu upravil, opis toho algoritmu je v podkapitole 2.6, kde sa nachádza aj opis metódy Top-K. Ďalej budem túto upravenú verziu nazývať Beam Search.

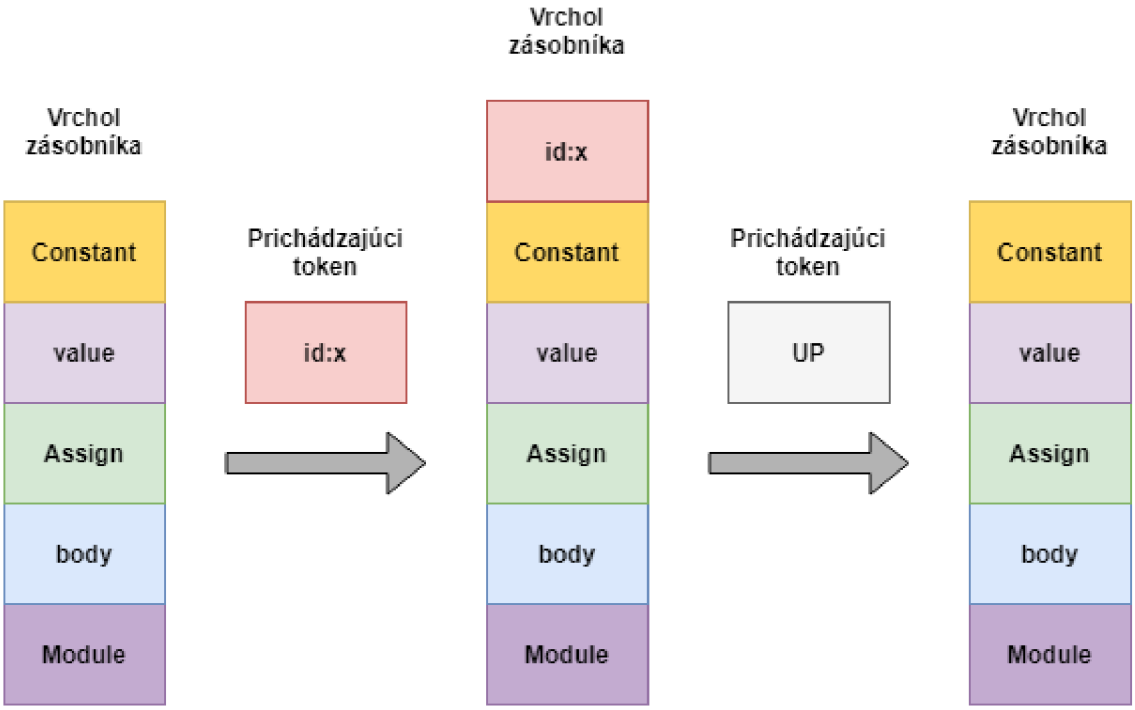
5.7 Užívateľské rozhranie

Ako rozhranie na demonštráciu funkcie modelu som sa rozhodol použiť editor Sublime Text[6]. Tento editor obsahuje jednoduché prostredie pre písanie pluginov, navyše v jazyku Python. Po stlačení istej klávesovej skratky sa spustí funkcia, ktorá analyzuje text v okolí kurzora, potom ho pošle cez modul requests na server, ktorý naspäť vráti kód na doplnenie. Tento server bude obsluhovať systém, ktorý je ukázaný na obrázku 5.9 v časti nasadenie. Ukážka použitia tohoto rozhrania je na obrázku 5.11.

[6] https://www.sublimetext.com/



Obr. 5.9: Integrácia modelu do systému



Obr. 5.10: Systém na kontrolu gramatiky

Obr. 5.11: Použitie užívateľského rozhrania. Modul "Systém"je znázornený na obrázku 5.9 v sekcii "Nasadenie".

Kapitola 6

Implementácia

Pre implementáciu tohoto systému som sa rozhodol použiť prostredie jazyka Python verzie 3.8. Python je univerzálne použiteľný jazyk, jeho prostredie obsahuje množstvo knižníc, ktoré uľahčujú prácu s dátami, strojovým učením, a podobne.

6.1 Spracovanie dát

Dáta z originálneho zdroja boli stiahnuté v ZIP formáte. Tento ZIP súbor obsahuje 14 súborov formátu GZIP. Na extrakciu týchto súborov bola použitá knižnica gzip[1]. Po extrakcii každého z týchto archívov dostaneme textový súbor, ktorý pozostáva z riadkov, pričom každý riadok je text vo formáte JSON[2]. Na načítanie tohoto textu bola použitá knižnica json[3]. Táto obsahuje funkciu *loads*, ktorá konvertuje text v JSON formáte na pythonovský slovník[4].

Po načítaní celého datasetu dostaneme zoznam slovníkov, pričom každý prvok tohoto zoznamu je zároveň jeden dátový bod. Každý z týchto slovníkov obsahuje 12 položiek, najčastejšie ide o rôzne metadáta[5]. Pre účely tohoto modelu bude potrebné použiť len 2 z týchto položiek, a to *code* a *docstring*. Tieto položky obsahujú čistý nespracovaný textový formát kódu a dokumentačného reťazca, ktorý prináleží tomuto kódu.

Pre každý z párov kód, dokumentačný reťazec je potom zavolaná funkcia *remove_docstring*. Táto funkcia má za úlohu odstrániť dokumentačný reťazec z kódu, keďže položka *code* obsahuje kód aj s dokumentačným reťazcom. Na to je použitá knižnica ast[6], konkrétne jej podčasť *NodeTransformer*. Kód je pomocou funkcie *ast.parse* najprv rozložený na abstraktný syntaktický strom. Potom je na tento strom aplikovaná funkcia, ktorá vezme každý element typu *Expr*, tzn. výraz, ktorý obsahuje konštantu typu *str*, tzn, reťazec. Toto sa deje z dôvodu, že v takejto formu vystupuje dokumentačný reťazec v kóde. Ak platí podmienka, že tento výraz obsahuje dokumentačný reťazec, alebo naopak, je z kódu odstránený. Ak sa nenájde ani jeden takýto reťazec, tak je daná dvojica kódu a dokumentačného reťazca z datasetu odstránená. Ukážka tohoto algoritmu je na obrázku 6.1.

Ďalšou súčasťou je filtrovanie kódu na základe jeho dĺžky. Aby bol konkrétny príklad kódu zahrnutý do datasetu, musí spĺňať 2 podmienky. Prvou je maximálny počet riadkov,

[1] https://docs.python.org/3/library/gzip.html
[2] https://www.json.org/json-en.html
[3] https://docs.python.org/3/library/json.html
[4] https://docs.python.org/3/tutorial/datastructures.html#dictionaries
[5] https://github.com/github/CodeSearchNet
[6] https://docs.python.org/3/library/ast.html

ktorý bol nastavený na 6. Druhou je maximálny počet tokenov, ktorý bol nastavený na 256 z dôvodu dimenzie použitého jazykového modelu. Výsledná rozhodujúca hranica je napokon znížená na 254, keďže musíme započítať aj *START* a *END* tokeny.

Po filtrovaní na základe dĺžky kódu nasleduje filtrovanie dokumentačných reťazcov. Pri každom z nich je ponechaný len prvý riadok. Ak je tento riadok prázdny, alebo splňuje formát regulárneho výrazu, ktorý zahŕňa najpoužívanejšie formáty dokumentačnch reťazcov, je vynechaný. Viac o týchto formátoch je napísané v podkapitole 5.1. Ukážka kódu je na obrázku 6.2.

Druhá časť spracovania dát potom pozostáva z tokenizácie výsledného textu. Obidva výstupy sú priebežne ukladané do formátu pickle[7] a následne skomprimované do formátu GZIP.

```
class DocstringRemover(ast.NodeTransformer):
    def __init__(self, docstring):
        ast.NodeTransformer.__init__(self)
        self.docstring = docstring
        self.removed = False

    def visit_Expr(self, node):
        const = node.value
        if isinstance(const, ast.Constant):
            val = const.value
            if isinstance(val, str):
                if self.docstring in val or val in self.docstring:
                    self.removed = True
                    return None
```

Obr. 6.1: Ukážka časti kódu na odstránenie dokumentačného reťazca

```
def remove_tags(doc):
    lines = doc.split('\n')
    first_line = lines[0].strip('\t ')
    if first_line == '' or re.search(':\w+(\s+\w+)?:\s+\w+', first_line):
        return None
    return re.sub('\s+', ' ', first_line)
```

Obr. 6.2: Ukážka časti kódu na filtrovanie dokumentačných reťazcov

[7] https://docs.python.org/3/library/pickle.html

6.2 Tokenizácia

Ako bolo uvedené v podkapitole 5.3, tokenizácia pozostáva z dvoch krokov. V prvom je text prevedený na syntaktický strom a následne na sekvenciu, v druhom sú slovné tokeny, tzn. tokeny, ktoré reprezentujú listové uzly stromu, spracované pomocou klasického tokenizéra.

Rozklad kódu na syntaktický strom

V prvej fáze prevodu kódu na syntakitcký strom využijeme modul ast[8], ktorý vyprodukuje objekt typu *ast.Module*. Tento objekt je koreňovým elementom, a obsahuje celý syntaktický strom. Ukážka tohoto stromu je na obrázku 6.3.

```
Module(
    body=[
        FunctionDef(
            name='helloWorld',
            args=arguments(
                posonlyargs=[],
                args=[],
                vararg=None,
                kwonlyargs=[],
                kw_defaults=[],
                kwarg=None,
                defaults=[]),
            body=[
                Expr(
                    value=Call(
                        func=Name(
                            id='print',
                            ctx=Load()),
                        args=[
                            Constant(
                                value='Hello world!',
                                kind=None)],
                        keywords=[]))],
            decorator_list=[],
            returns=None,
            type_comment=None)],
    type_ignores=[])
```

Obr. 6.3: Ukážka syntaktického stromu

Tento strom je následne prevedený do JSON reprezentácie. Jedná sa o akýsi medzikrok medzi AST objektom a sekvenciou tokenov. AST objekt obsahuje parametre, ktoré sú len

[8] https://docs.python.org/3/library/ast.html

informačného charakteru, a sú prebytočné. Je to napríklad parameter *ctx*, ktorý určuje kontext, v ktorom sa využíva nejaká premenná. Môže nadobúdať hodnoty *Load()*, *Store()* a *Del()*. Tento parameter sa už neukladá do JSON reprezentácie, je kompletne vynechaný.

Ďalšou z potrebných úprav pôvodného AST je pridanie vlastnej dvojice tokenov *string* a *other*. V pôvodnom AST vzniká problém v listových uzloch, kde nie je explicitne určený typ konštanty. Príkladom takejto nekonzistencie je napr literál *123*, ktorý môže vyjadrovať ako číslo, tak aj reťazec. Ďalej to môže byť literál *None*, ktorý môže vyjadrovať špeciálnu konštantu *None*, a znova aj reťazec. Ak chceme hodnoty týchto uzlov poslať tokenizéru na rozdelenie na slová, je potrebné nejakým spôsobom uchovať informáciu o type tejto konštanty. Preto som zaviedol tieto tokeny. Jeden z týchto tokenov je vždy vložený pred každú konštantu, čím sa určí, či sa jedná o reťazec, alebo inú hodnotu. Jediný rozdiel je, že pri vytváraní JSON reprezentácie je na tokeny typu *other* zavolaná funkcia *str()*[9], a pri spätnej rekonštrukcii z JSON na AST je na tokeny typu *other()* zavolaná funkcia *eval()*[10]. Ukážka kódu pri vytváraní JSON stromu z AST je na obrázku 6.4, opačný prípad je na obrázku 6.5.

```
if not isinstance(tree, ast.AST):
    if isinstance(tree, str):
        return 'string', tree
    else:
        return 'other', str(tree)
```

Obr. 6.4: Zavedenie špeciálnych tokenov pri prevode z AST na JSON strom

```
if not isinstance(tree, dict):
    if tree[0] == 'string':
        return tree[1]
    else:
        return eval(tree[1])
```

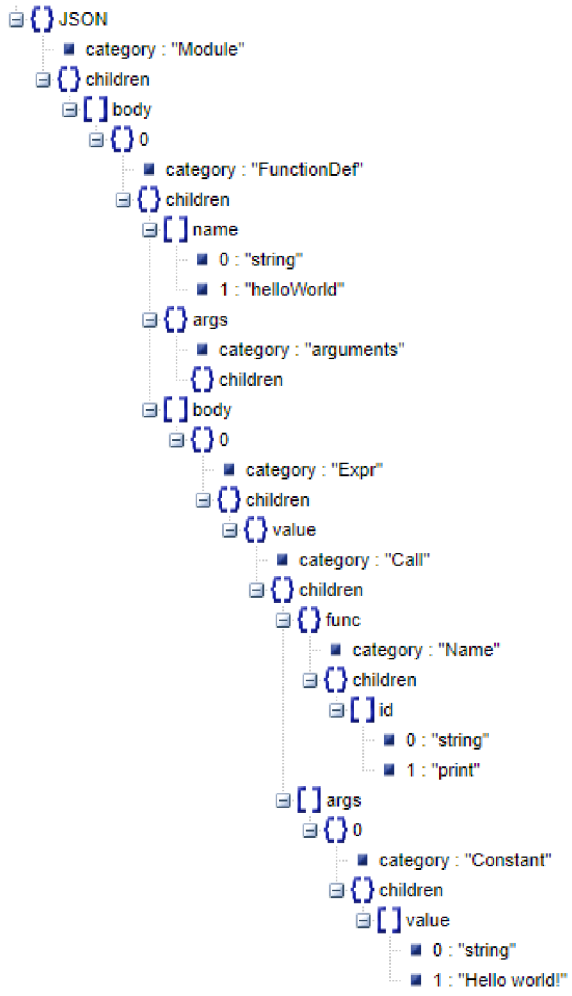
Obr. 6.5: Použitie špeciálnych tokenov pri prevode z JSON na AST strom

O prevod z AST na JSON strom sa stará funkcia *parse_tree()*. Táto funkcia pracuje rekurzívne pre celý strom. Začína od koreňového elementu. Pri každom elemente najprv zistí, či nie je koncový. Ak áno, vráti sa dvojica príslušného typu koncovej konštanty, buď *string* alebo *other*, a hodnoty tejto konštanty. Táto dvojica sa potom uloží do výsledného

[9] https://docs.python.org/3/library/functions.html
[10] https://docs.python.org/3/library/functions.html

JSON stromu. V opačnom prípade sa iteruje cez všetky podelementy daného elementu. Pri každej iterácii sa kontroluje, či daný element nemá byť vynechaný, ako napr. elementy typu *ctx*. Potom v prípade, že daný element je prázdny (jeho hodnota je *None*), a zároveň je z hľadiska abstraktnej gramatiky nepovinným atribútom, je takisto vynechaný. Ak element vynechaný nebol, je spracovaný ďalším rekurzívnym volaním tejto funkcie, a výsledok tohoto volania je priradený do JSON stromu.

Štruktúra JSON stromu je podobná AST stromu. Jedná sa o pythonovský slovník. Každý element obsahuje 2 položky: *category* a *children*. Položka *category* obsahuje názov kategórie tohoto elementu. Položka *children* obsahuje zoznam podelementov daného elementu. Tento zoznam ma taktiež podobu slovníka, kde kľúč je názov daného atribútu, a hodnota je samotný element. V prípade, že ide o koncový element, jeho hodnota je dvojica typu a hodnoty. Na obrázku 6.6 je ukážka štruktúry JSON stromu.



Obr. 6.6: Ukážka JSON reprezentácie abstraktného syntaktického stromu

Enumerácia tokenov kódu

Aby bolo možné rozložiť text kódu na tokeny, je nutné ich istým spôsobom očíslovať. Preto musí byť k dispozícii nejaký konečný zoznam všetkých tokenov. Pre Python verzie 3.8

existuje definícia abstraktnej gramatiky[11], ktorá je určená pre parser jazyka Python. Na základe tejto gramatiky potom vieme získať všetky tokeny. Ukážka časti tejto gramatiky je na obrázku 6.7.

```
slice = Slice(expr? lower, expr? upper, expr? step)
      | ExtSlice(slice* dims)
      | Index(expr value)

boolop = And | Or

operator = Add | Sub | Mult | MatMult | Div | Mod | Pow | LShift
             | RShift | BitOr | BitXor | BitAnd | FloorDiv

unaryop = Invert | Not | UAdd | USub

cmpop = Eq | NotEq | Lt | LtE | Gt | GtE | Is | IsNot | In | NotIn

comprehension = (expr target, expr iter, expr* ifs, int is_async)

excepthandler = ExceptHandler(expr? type, identifier? name, stmt* body)
                attributes (int lineno, int col_offset, int? end_lineno, int? end_col_offset)

arguments = (arg* posonlyargs, arg* args, arg? vararg, arg* kwonlyargs,
             expr* kw_defaults, arg? kwarg, expr* defaults)

arg = (identifier arg, expr? annotation, string? type_comment)
       attributes (int lineno, int col_offset, int? end_lineno, int? end_col_offset)

-- keyword arguments supplied to call (NULL identifier for **kwargs)
keyword = (identifier? arg, expr value)
```

Obr. 6.7: Časť abstraktej gramatiky jazyka Python verzie 3.8

Z prvého pohľadu na túto gramatiku je možné rozdeliť tokeny na 2 typy:

1. Uzly
2. Parametre

Parametre sú tokeny, ktoré sú súčasťou nejakého uzla. Na obrázku 6.7 sa jedná napr. o parametre na prvom riadku, konkrétne *lower*, *upper* a *step*. Tieto parametre majú svoj typ, ktorý však nie je podstatný dôležité je ich zaradenie do nejakého uzla. Ďalšou dôležitou vlastnosťou týchto parametrov je ich kardinalita. Podľa tejto abstraktnej gramatiky rozlišujeme 3 typy parametrov:

1. Jednopočetný
2. Viacpočetný
3. Nepovinný

Jednopočetný parameter musí byť prítomný vždy a práve raz. Vyznačuje sa tým, že pred názvom atribútu nie je žiadny zo symbolov * alebo ?. Na obrázku 6.7 ide napr. o parameter *value* na treťom riadku. V praxi to znamená, že uzol *Index* musí obsahovať práve jeden parameter *value*, a tento musí byť typu *expr*. Typ *expr* na tomto obrázku nie je, môže však

[11] https://raw.githubusercontent.com/python/cpython/3.8/Parser/Python.asdl

reprezentovať jeden z mnohých uzlov. Viacpočetný parameter musí byť prítomný aspoň raz, ale môže byť prítomný aj viackrát. Vyznačuje sa symbolom * pred názvom parametra. Príkladom je parameter *dims* na obrázku 6.7 v druhom riadku. Nepovinný parameter sa môže najviac jedenkrát vyskytnúť v uzle, nemusí sa však vyskytnú ani raz. Vyznačuje sa symbolom ? pred názvom parametra. Príkladom takého parametra je parameter *lower* v prvom riadku.

Uzly sú také tokeny, ktoré nie sú parametre. V praxi to znamená, že nie sú súčasťou žiadneho iného tokenu. Môžu obsahovať parametre, ale nemusia. Na obrázku 6.7 sa jedná napr. o token *Add*, ktorý neobsahuje žiadne parametre, alebo aj o token *Slice*, ktorý obsahuje 3 nepovinné parametre.

Na základe týchto znalostí je potom možné extrahovať informácie o tokenoch a uložiť ich do nejakej štruktúry. Rozhodol som sa pre tieto účely využiť slovníkovú štruktúru, najmä preto, že potrebujeme mapovať typy tokenov na čísla, ktoré potom budú slúžiť na tokenizáciu textu. Na získanie rôznych informácií o tokenoch, vrátane mapovania na čisla slúži funkcia *create_code_dicts()*. Funkcia očakáva jeden argument, celé číslo, ktoré určuje, od akého čísla sa budú tokeny číslovať. Je to potrebné z dôvodu, že samotý BPE tokenizér má svoje špeciálne tokeny, ktoré sa číslujú od 0.

Táto funkcia najprv stiahne abstraktnú gramatiku pomocou modulu requests[12], z ktorej extrahuje všetky potrebné informácie. Jej výstupom je slovník, ktorý obsahuje 4 slovníky:

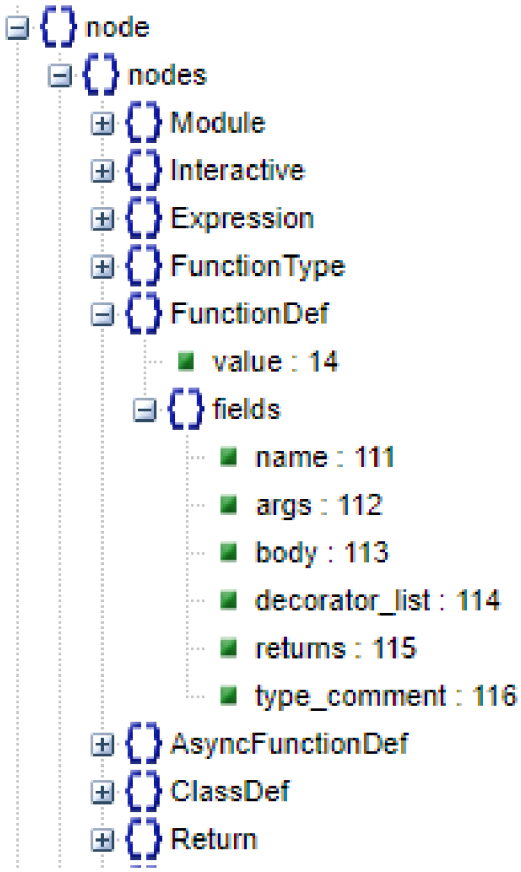
1. node_dict
2. attr_dict
3. elem_dict
4. item_dict

Každý z týchto slovníkov uchováva rôzne informácie o tokenoch. Slovníky *elem_dict* a *item_dict* nie sú potrebné pre mapovanie tokenov na čísla, využívajú sa pri kontrole gramatiky, ktorej sa venujem v podkapitole 6.4. Slovník *node_dict* slúži práve na toto mapovanie. Obsahuje položky *nodes* a *special*. Položka *nodes* obsahuje mapovanie tokenov na čísla na základe abstraktnej gramatiky. Položka *special* obsahuje mapovanie špeciálnych tokenov *string* a *other*.

V položke *nodes* sú ako kľúče zahrnuté všetky uzly. Hodnota pri každom z týchto kľúčov je slovník, ktorý obsahuje kľúče *value* a *fields*. Kľúč *value* má ako hodnotu priradené číslo toho konkretného uzla. Kľúč *fields* obsahuje slovník, ktorý priradzuje čísla každému z parametrov tohto uzla. Tento slovník môže byť aj prázdny za predpokladu, že tento uzol neobsahuje žiadne parametre. Kľúče v tomto slovníku sú názvy parametrov, a hodnoty sú ich čísla. Ukážka časti tohoto slovníka je na obrázku 6.8.

Ďalším z využívaných slovníkov pri vytváraní JSON stromu je *attr_dict*. Tento obsahuje informácie o type tokenov, tj. či sú mnohopočetné, alebo nepovinné. Obsahuje 2 položky: *list* a *optional*. Každá z týchto položiek je zoznamom čísel, ktoré kódujú tokeny (tieto čísla sú určené slovníkom *node_dict*). Ak sa číslo tokenu nachádza v položke list, tak ide o mnohopočetný token, podobne ak sa nachádza v položke optional, tak ide o nepovinný token. Jednoppočetné tokeny nie je potrebné ukladať, keďže sa jedná o zvyšné parametre, ktoré nie sú ani v jednom zozname.

[12]https://docs.python-requests.org/en/master/



Obr. 6.8: Časť slovníka *node_dict*. Expandovaná je iba časť uzlov.

Funkcia *create_code_dicts* používa najmä regulárne výrazy na rozdelenie taxtu na podčasti. Z týchto častí sa potom získavajú informácie. Keďže táto abstraktná gramatika je štruktúrovaná, získanie informácii o tokenoch nie je zložité. Na obrázku 6.9 je ukážka začiatku tejto funkcie.

```
def create_code_dicts(token_offset):
    url = 'https://raw.githubusercontent.com/python/cpython/3.8/Parser/Python.asdl'
    text = requests.get(url).text
    default_types = text.split('\n')[1].split('-- ')[1].split(', ')
    default_types.remove('object')
    text = re.sub('\s*--[^\\n]+\\n', '\n', text)
    matches = re.finditer('(\w+) = ((((([\\n\s]+\| )?\w+(\([\w\\n, *?]+\))?)+)|(\([\w\\n, *?]+\)))', text)
```

Obr. 6.9: Časť kódu funkcie na extrakciu informácií o tokenoch z abstraktnej gramatiky

Rozloženie syntaktického stromu kódu na tokeny

Ako už bolo spomenuté v kapitolách venujúcich sa návrhu, model, ktorý budeme používať, očakáva na vstupe sekvenciu slov (tokenov). My máme k dispozícii strom v JSON formáte a slovníky na mapovanie tokenov na čísla. Z tohoto je potrebné vytvoriť sekvenciu očíslovaných tokenov.

O tento prevod sa stará funkcia *flatten_tree()*. Na vstup dostane JSON strom a *node_dict*, jej výstupom je sekvencia dvojíc. Táto dvojica pozostáva z textovej formy tokenu a príslušného číselného kódu. Okrem tokenov, ktoré dostali priradené číslo podľa slovníka totiž existujú ešte 2 ďalšie špeciálne tokeny. Jeden z nich, token *UP* má číslo o 1 menšie, ako prvý token v slovníku. Ako bolo spomenuté v podkapitole 5.2, tento token je nevyhnutný pre následnú rekonštrukciu stromu zo sekvencie.

Ďalším z týchto tokenov je token *WORD*, ktorého číslo je analogicky o 2 menšie, ako číslo prvého tokenu v slovníku. Tento token slúži na označenie listových uzlov, tzn. konštánt, ktoré sa z gramatického hľadiska nedajú ďalej deliť. Ide napr. o názvy premenných, reťazcové literály, číselné literály, konštantu *None*, a podobne. V ďalšej fáze tokenizácie sa tento token nahradí sekvenciou 1 až N tokenov, o čo sa postará BPE tokenizér. Funkcia *flatten_tree()* pracuje rekurzívne, a to taým spôsobom, že sa výsledok volania funkcie vždy pridá na koniec výsledného zoznamu tokenov. Základom funkcie je implementácia pre-order prechodu cez JSON strom. Pri každom návrate o úroveň vyššie je teda pridaný do sekvencie token *UP*, kvoôli spätnej kompatibilite. Táto funkcia začína od koreňového elementu stromu, ktorý je spracovaný. Potom sú rekurzívne spracovaní všetci potomkovia tohoto elementu. Typ každého elementu sa vyčíta z položky *category* v JSON strome, zoznam potomkov potom z položky *children*. Pri spracovávaní každého elementu sa zo slovníka *node_dict* zistí číslo príslušného elementu (tokenu). Spôsob, akým sa získava číslo tokenov je ukázaný na časti kódu tejto funkcie na obrázku 6.10. Ukážku výstupu tejto funkcie je možné vidieť na obrázku 6.11.

```
166  def flatten_tree(tree, node_dict, token_offset, inference=False):  # pre-order
167      TOKEN_UP = token_offset + 1
168      cat = tree['category']
169      tokens = [(cat, node_dict['nodes'][cat]['value'])]
170
171      children = list(tree['children'])
172      if inference and cat == 'FunctionDef':
173          children = modify_tree(children)
174
175      for child in children:
176          tokens.append((child, node_dict['nodes'][cat]['fields'][child]))
177          child_item = tree['children'][child]
```

Obr. 6.10: Časť kódu funkcie na prevod JSON stromu na sekvenciu tokenov. Na riadku 169 sa ukladá súčasne spracovávaný element do dvojice, ktorá reprezentuje token. Jeho hodnota je získaná zo slovníka *node_dict*. Na riadku 175 sa potom iteruje cez všetkých potomkov tohoto elementu. Na riadku 176 sa podobne ukladá hodnota súčasného potomka do výslednej sekvencie, znova sa pristupuje k hodnote v slovníku. Na riadku 177 sa potom získa samotná hodnota príslušného potomka, a funkcia ju spracováva ďalej.

```
[('Module', 10), ('body', 105), ('FunctionDef', 14), ('name', 111),
('string', 256), ('helloWorld', 8), ('up', 9), ('args', 112),
('arguments', 99), ('up', 9), ('up', 9), ('body', 113), ('Expr', 35),
('value', 176), ('Call', 55), ('func', 209), ('Name', 62), ('id', 223),
('string', 256), ('print', 8), ('up', 9), ('up', 9), ('up', 9),
('args', 210), ('Constant', 58), ('value', 216), ('string', 256),
('Hello world!', 8), ('up', 9), ('up', 9), ('up', 9), ('up', 9),
('up', 9), ('up', 9), ('up', 9), ('up', 9), ('up', 9), ('up', 9)]
```

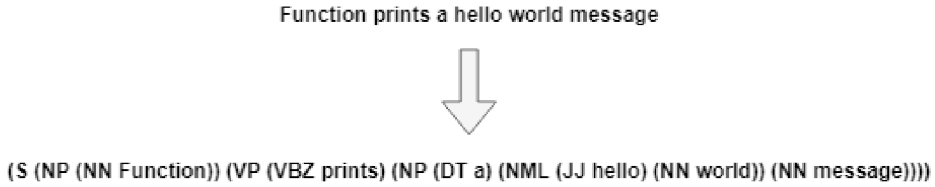
Obr. 6.11: Ukážka výstupu funkcie *flatten_tree()*. Token *Module* je prvý v poradí, má číslo 10. Token *UP* má číslo 9. Všetky slovné tokeny dostali priradené číslo 8.

Tokenizácia dokumentačných reťazcov

Keďže v tejto práci používam syntaktickú analýzu kódu, je potrebné ju použiť aj pri dokumentačných reťazcoch, resp. komentároch[13]. Podobne ako pri kóde, je nutné najprv rozložiť tieto komentáre na syntaktický strom.

Rozloženie prirodzeného jazyka na takýto strom nie je triviálna úloha. Preto som sa rozhodol pre tieto účely využiť modul benepar[14], ktorý to dokáže. Ide o modul postavený na knižnici spaCy[15]. Tento modul rozdelí vetu na slová, a následne na syntaktický strom. Jednotlivé syntaktické elementy sú používané na základe projektu Penn Treebank [25].

Formát syntaktického stromu, ktorý produkuje modul benepar, je textový. Ukážka vstupnej vety a výstupného syntaktického stromu je na obrázku 6.12.



Obr. 6.12: Prevod textu na syntaktický strom za použitia modulu benepar

Z obrázku 6.12 vidíme, že tento syntaktický strom ma formu textu. To nám značne uľahčuje prevod na sekvenciu tokenov, keďže z textu je vidno, že ide v podstate o serializáciu stromu pre-order prechodom. Takýto istý prechod je využívaný aj pri spracovaní kódu.

Podobne ako pri kóde, je potrebné aj v tomto prípade získať zoznam tokenov a očíslovať ho. Na získanie všetkých tokenov slúži funkcia *get_all_tokens()*. Táto funkcia najprv pomocou modulu requests[16] stiahne obsah stránky[17], kde sa nachádza takmer kompletný zoznam vetných členov podľa projektu Penn Treebank. Potom sú z tejto stránky pomocou modulu bs4[18] extrahované všetky potrebné vetné členy. Jedná sa zväčša o 1 až 3 písmenové kódy, ako je vidno na obrázku 6.12. Tento zoznam je doplnený o ďalšie 4 tokeny, ktoré boli buď uvedené v publikácii [25], alebo boli nájdené na základe prechádzania datasetu komentárov, a neboli obsiahnuté na danej stránke. Kompletný zoznam týchto vetných členov

[13] V priebehu práce používam termín komentár a dokumentačný reťazec, tieto termíny sú z hľadiska práce identické.

[14] https://pypi.org/project/benepar/

[15] https://spacy.io/

[16] https://docs.python-requests.org/en/master/

[17] https://www.surdeanu.info/mihai/teaching/ista555-fall13/readings/PennTreebankConstituents.html

[18] https://pypi.org/project/beautifulsoup4/

používaných modulom benepar sa mi totiž nepodarilo nájsť. Nie je to však taký problém, zo stotisícov komentárov som zaznamel problém chýbajúceho vetného člena v slovníku len párkrát, takéto dátové vzorky môžu byť vzhľadom na veľkosť datasetu vynechané.

Potom, čo funkcia *get_all_tokens()* získa tento zoznam, zavolá sa funkcia *create_token_dict()*, ktorá tento zoznam premení na očíslovaný slovník. Kľúčami v tomto slovníku sú názvy tokenov, hodnotami sú príslušné čísla.

Po získaní slovníku na mapovanie tokenov na čísla môžeme previesť syntaktický strom na sekvenciu. Funkcia *flatten_tree_doc()* dostane na vstup tento text a slovník s tokenmi, a následne ho premení na sekvenciu tokenov, ktorá má rovnaký formát, ako pri kóde. Funkcia najprv vyčistí regulárnym výrazom tento text, aby sa dal jednoducho rozdeliť na časti podľa medzier. Po rozdelení dostávame zoznam vetných členov a samotných slov. Funkcia iteruje týmto zoznamom, a pri každom prvku zisťuje, či sa začína otvorenou zátvorkou. Ak áno, nájde sa v slovníku tokenov a pridá sa do sekvencie. Ak nie, znamená to, že sa pridá do sekvencie token *UP*. Ukážka výstupu tejto funkcie je na obrázku 6.13. Ako si je možné všimnúť, štruktúra je zhodná so sekvenciou tokenov pri kóde, na obrázku 6.11.

```
[('S', 47), ('NP', 32), ('NN', 28), ('Function', 8), ('up', 9),
('up', 9), ('VP', 62), ('VBZ', 61), ('prints', 8), ('up', 9),
('NP', 32), ('DT', 15), ('a', 8), ('up', 9), ('NML', 72), ('JJ', 21),
('hello', 8), ('up', 9), ('NN', 28), ('world', 8), ('up', 9),
('up', 9), ('NN', 28), ('message', 8), ('up', 9), ('up', 9),
('up', 9), ('up', 9)]
```

Obr. 6.13: Ukážka výstupu funkcie *flatten_tree_doc()*. Tento výstup je výsledkom vstupu z obrázku 6.12.

BPE tokenizácia

Výstupom metód uvedených v predchádzajúcich kapitolách je zoznam dvojíc hodnôt, ako na obrázkoch 6.11 a 6.13, pričom každá z týchto dvojíc je jeden token. Ako je vidno, tieto zoznamy obsahujú aj tokeny typu *WORD* (na obrázkoch označené číslom 8), ktoré musia byť ďalej spracované. Tu prichádza na rad BPE tokenizér, ktorý každé z týchto slov nahradí sekvenciou podslov, a každému priradí číselné označenie.

BPE tokenizér používaný v tejto práci sa nazýva SentencePiece [16]. Obsahuje jednoduché API pre Python, čo z neho robí vhodnú voľbu pre tento projekt. Rovnako ho použili v práci [21], čím som sa inšpiroval.

Tento tokenizér, podobne ako iné jazykové modely, vyžaduje tréning. Tréning prebieha bez učiteľa, čo v praxi znamená, že mu postačuje veľký obsah textu, tzv. korpus. Keďže kód a komentáre majú odlišnú štruktúru, bolo potrebné natrénovať 2 tokenizéry, jeden pre kód a druhý pre komentáre. Pre každý z nich bol vytvorený osobitný korpus. Tie boli vytvorené tak, že sa načítal dataset po prvej fáze spracovania, t.j. pred tokenizáciou, ale prefiltrovaný od nepotrebných dát. Obsahy týchto datasetov sa potom jednoducho konkatenovali za sebou do 2 veľkých súborov. Obidva tokenizéry boli natrénované na týchto korpusoch, a potom boli pripravené na použitie.

Vo fáze použitia tokenizéra je mu dodaná sekvencia tokenov, ako napr. na obrázku 6.11. Potom prebieha iterácia cez všetky tokeny. Ak je token typu *WORD*, je jeho slovná hodnota predaná tokenizéru, konkrétne cez metódu *encode_as_ids()*. Táto vráti zoznam čísel, ktoré kódujú toto slovo. Tento zoznam je potom vložený namiesto *WORD* tokenu. V tejto fáze sa

token zmení z dvojice už iba na jedno číslo, keďže už nie je potrebné uchovávať informáciu o texte *WORD* tokenu. Ukážka výslednej sekvencie je na obrázku 6.14.

```
10, 105, 14, 111, 256, 345, 3984, 15987, 6333, 9, 112, 99, 9, 9, 113,
35, 176, 55, 209, 62, 223, 256, 2196, 9, 9, 9, 210, 58, 216, 256,
752, 3984, 8443, 1, 9, 9, 9, 9, 9, 9, 9, 9, 9
```

Obr. 6.14: Príklad finálnej formy tokenizácie. Tento príklad vznikol spracovaním sekvencie z obrázka 6.11

6.3 Detokenizácia kódu

Keďže úlohou modelu je predpovedať, resp. generovať napísaný kód, je potrebné vedieť previesť tokeny späť na text. Podobne ako pri tokenizácii, BPE tokenizér dokáže previesť sekvenciu čísel späť na slovo. BPE tokenizér však spracuje len časti, ktoré zodpovedajú slovným tokenom. To, ktoré čísla zodpovedajú slovu a ktoré nie, sa dá jednoducho zistiť podľa rozsahu tokenov. Tento rozsah je znázornený na obrázku 6.15. Ako je možné vidieť na obrázku, ak tokenizér vidí číslo s hodnotou v rozsahu od 259 do 15999, vie, že sa jedná o slovo. Do tohoto slova konkatenuje ďalšie čísla, až pokiaľ nenarazí na číslo mimo toho rozsahu. Túto sekvenciu čísel premení na slovo a nastaví mu ako token konštantu *WORD*, zvyšok sa ponechá na spracovanie ďalej.

Obr. 6.15: Rozloženie slovnej zásoby modelu. Veľkosť slovnej zásoby je dokopy 16000 slov. Čísla na rozhraní 2 sekcií označujú číslo prvého tokenu v sekcii napravo. Token *WORD* má číslo o 1 menšie, ako *UP*, teda 9.

Výsledkom tejto operácie je zoznam dvojíc hodnôt, podobne ako na obrázku 6.11. Tie sa ďalej pošlú funkcii *construct_tree()*, ktorá funguje opačným spôsobom, ako *flatten_tree()*. Tá zo sekvencie tokenov vytvorí JSON strom, jeho štruktúru je schopná zrekonštruovať vďaka *UP* tokenom. Tento výsledný JSON strom sa potom premení na AST strom pomocou funkcie *unparse_tree()*. Táto musí robiť opačnú operáciu, ako funkcia *parse_tree()*. Jedným z úskalí tohoto prístupu je, že funkcia *parse_tree()* odstránila zo stromu nepotrebné parametre, čo môže spôsobovať problémy pri spätnej konštrukcii kódu. Modul ast sa totiž bez niektorých parametrov nezaobíde. Toto je vo funkcii *unparse_tree()* vyriešené spôsobom, ktorý je ukázaný na obrázku 6.16. Tento kód na riadku 150 prejde všetkými položkami, ktoré gramatika obsahuje. Ak nejaká z týchto položiek ešte nebola pridaná do stromu, tak potom v závislosti od toho, či sa jedná o mnohopočetný alebo jednopočetný token, sa pridá do stromu buď to prázdny zoznam, alebo konštanta *None*.

```
150         for item, code in node_dict['nodes'][tree['category']]['fields'].items():
151             if item not in tree['children']:
152                 if code in attr_dict['list']:
153                     exec(f'new_tree.{item} = []')
154                 else:
155                     exec(f'new_tree.{item} = None')
```

Obr. 6.16: Ukážka kódu funkcie *unparse_tree()*

Po vytvorení AST stromu sa už len zavolá funkcia *unparse()* z modulu astunparse[19], ktorá premení AST na text reprezentujúci výsledný kód.

6.4 Kontrola gramatiky

Ako už bolo spomenuté v návrhu, model za každým vyprodukuje zoznam tokenov, zoradený podľa pravdepodobnosti, že daný token je ďalší v sekvencii. Keďže systém musí tieto tokeny premeniť na text na základe syntaktických pravidiel, s vysokou pravdepodobnosťou môže nastať, že danú sekvenciu tokenov nebude možné premeniť na text. Preto je zavedený systém na kontrolu gramatiky. Funguje tak, že za každým novým tokenom sa aktualizuje jeho stav, a potom vráti zoznam tokenov, ktoré môžu nasledovať. Tento zoznam sa porovná so zoznamom pravdepodobností z modelu. Napokon sa vyberie ako ďalší ten token, ktorý má spomedzi zoznamu gramaticky povolených tokenov najvyššiu pravdepodobnosť. Tento sa znova pošle systému na kontrolu gramatiky, a proces sa opakuje.

Najpodstatnejšou triedou tohoto systému je trieda *GrammarChecker*. Okrem iného obsahuje táto trieda 2 najpoužívanejšie metódy: *next_token()* a *get_next_tokens()*. Metóda *next_token()* príjma ako argument číslo tokenu, ktorý prichádza ako ďalší. Táto funkcia aktualizuje vnútorný stav systému, čím sa zmení zoznam povolených nasledujúcich tokenov. Funkcia *get_next_tokens()* vráti tento zoznam. Aby sa do istej miery predišlo zacykleniu modelu, obsahuje táto trieda metódu *set_max_words()*. Touto metódou sa môže nastaviť limit na dĺžku jedného slova v počte tokenov. Ak je napr. maximálna dĺžka nastavená na 3 a systém už obdržal 3 po sebe idúce slovné tokeny, tak ďalší povolený token bude už len token *UP*, ktorý značí koniec slova.

Tento systém ďalej obsahuje 3 triedy, z ktorých každá reprezentuje jeden typ tokenu:

1. Node
2. Attribute
3. Special

Prvá z týchto tried reprezentuje uzlové tokeny, druhá parametrové, a tretia reprezentuje dvojicu špeciálnych tokenov *string* a *other*. Ďalej tento systém obsahuje triedu *Stack*, ktorá je jednoduchou implementáciou zásobníka. Tento zásobník používa trieda *GrammarChecker*, a ukladá na neho prichádzajúce tokeny. Podľa typu prichádzajúceho tokenu vždy vytvorí novú inštanciu triedy, ktorá zodpovedá jeho typu tzn. buď *Node*, *Attribute* alebo

[19] https://pypi.org/project/astunparse/

Special, a uloží ju na vrchol zásobníka. Podľa dvojice slovníkov *elem_dict* a *attr_dict* potom zistí, aké ďalšie tokeny môžu nasledovať. Podľa tejto informácie potom aktualizuje svoj vnútorný stav.

Dôvod, prečo je pre každý typ tokenu vytvorená osobitná trieda je taký, že každá z týchto troch tried obsahuje metódu *get_next_tokens()*. Po vytvorení inštancie danej triedy a jej umiestnení na vrchol zásobníka sa teda len jednoducho zavolá metóda priamo v tejto inštancii, a táto vráti povolené tokeny, ktoré po nej môžu nasledovať. Týmto sa výrazne zjednodušuje celý systém, keďže predáva úlohy jeho subkomponentom, a potom ich len na základe istých kritérií upraví.

6.5 Model a tréning

Na samotnú predikciu tokenov bol použitý klasický Transformer [23]. Tento model dosiahol veľmi dobré výsledky v modelovaní sekvencií slov.

V tejto práci som použil implementáciu Transformera zo stránky frameworku Tensorflow[20], ktorú som trochu upravil, aby sa dala použiť v tomto systéme.

Okrem samotného modelu som vytvoril aj funkciu pre tréning. Jedným z vylepšení je aj systém pre zálohu natrénovaného modelu v istých intervaloch. Vzhľadom na veľkosť modelu a počet parametrov je potrebné ponechať len max. 3 posledné uložené stavy. Okrem natrénovaných parametrov modelu sa ukladá aj stav, ako napr. presnosť, poradie vzorky v epoche atď. Keďže som pred každou epochou zamiešal poradie vzoriek, vždy sa uloží len pole ich poradí. To sa potom pri prípadnom prerušení a znovunačítaní použije tak, aby bolo zachované poradie vzoriek vrámci jednej epochy. V opačnom prípade by sa totiž mohli vzorky opakovať, či niektoré by ani neprešli tréningom.

Tréning obidvoch modelov prebiehal na službe Google Colab[21]. Použitá grafická karta bola NVIDIA Tesla T4 16GB[22].

[20] https://www.tensorflow.org/text/tutorials/transformer
[21] https://colab.research.google.com/
[22] https://www.nvidia.com/en-us/data-center/tesla-t4/

Kapitola 7

Testovanie

7.1 Prvá fáza testovania

Ako prvú z metód vyhodnocovania modelov som použil funkciu presnosti opísanú v podkapitole 2.4. Táto funkcia sa zameriava len na výstup tokenov, nie samotného detokenizovaného textu. Druhý model bol trénovaný na 10 epoch, avšak uložil som si postup po každej epoche. Najlepšiu presnosť dosiahol po epoche číslo 10. Preto údaje o jeho presnosti budú pochádzať z tejto epochy. Porovnanie parametrov a výsledkov je v tabuľke 7.1.

Tabuľka 7.1: Porovnanie parametrov a výsledkov modelov. Prvý model je kontrolný, bez rozkladu na syntax, druhý je opisovaný v celej práci, obsahuje syntaktický rozklad.

	Prvý model	Druhý model
Veľkosť datasetu	377649	118021
Počet vrstiev Transformera	6	8
Počet *attention heads*	8	12
Počet epoch pri vyhodnocovaní	8	10
Batch size	16	32
Veľkosť slovnej zásoby	16000	16000
Rozmer vektoru jedného slova	512	768
Maximálna dĺžka sekvencie	512	256
Rozmer *feed forward* siete	2048	3072
Dropout rate	0,05	0,1
Počet parametrov modelu	69M	169M
Počet vzorkov pri vyhodnocovaní	1000	1000
Približná doba tréningu	72 hodín	30 hodín
Syntaktické spracovanie textu	**Nie**	**Áno**
Strata	**2,1728**	**0,8331**
Presnosť	**60,03%**	**85,41%**

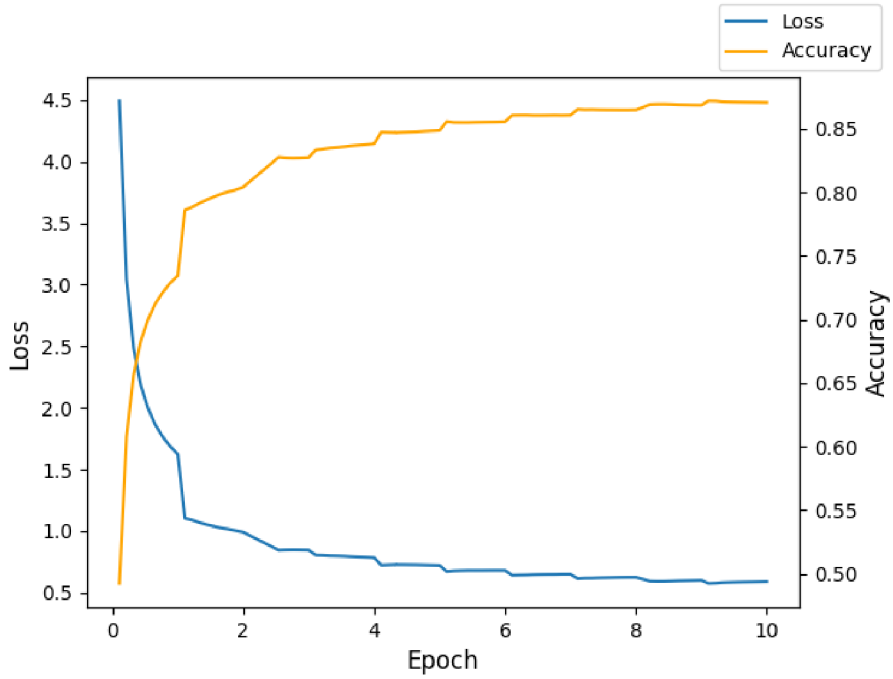
Ako je vidno z tabuľky 7.1, druhý model dosiahol v porovnaní s prvým lepšie výsledky. Napriek tomu, že mal väčšie rozmery a kratšiu dĺžku sekvencie, mal 3,2-krát menší dataset, a potreboval na tréning o 60% menej času, ako prvý model. Tento rozdiel sa dá pripísať použitiu syntaktického systému.

Ako bolo spomenuté, model bol vyhodnocovaný po 10. epoche, keďže mal z hľadiska presnosti a straty najlepšie výsledky. Tieto parametre sa však od 5. epochy už výrazne

nemenili, čo značí, že model mal určite dostatok epoch na tréning. Progres v jednotlivých epochách je zhrnutý v tabuľke 7.2. Podobne, hodnoty straty a presnosti počas tréningu na validačných dátach sú znázornené na obrázku 7.1.

Tabuľka 7.2: Prehľad postupu tréningu modelu po jednotlivých epochách. Epocha č. 1 zahrnutá nie je, pretože model mal po tejto epoche výrazne horšie výsledky.

Epocha	Strata	Presnosť
2	0,9067	83,06%
3	0,8771	83,98%
4	0,8356	84,16%
5	0,8455	84,65%
6	0,8475	84,41%
7	0,8551	84,82%
8	0,8332	85,12%
9	0,8528	84,81%
10	0,8331	85,41%



Obr. 7.1: Priebeh straty a presnosti na validačných dátach počas tréningu. Skokové prerušenia grafu sa vyskytujú na rozhraniach epoch z dôvodu, že strata a presnosť boli počítané ako priemer za celú epochu.

7.2 Druhá fáza testovania

Metriky ako presnosť či strata samozrejme nemusia znamenať, že model bude použiteľný v praxi. Preto som v druhej fáze testovania skúmal, ako sa systém dá použiť spoločne s editorom Sublime Text. V tejto fáze som experimentoval aj s rôznymi heuristikami, ako Top-K

či Beam search. Systém som v tomto prípade testoval po 5. epoche, keďže s pribúdajúcimi epochami sa systém pravdepodobne trochu pretrénoval, a vracal repetitívny kód.

Jedným z problémov tohoto systému je čas poskytnutia odpovede. Systém som testoval na svojom PC, ktorý obsahuje grafickú kartu NVIDIA GeForce GTX 1050Ti 4GB[1]. Porovnával som časy rôznych typov požiadavku, s rôznymi parametrami. Všetky časy sa však pohybovali v rozmedzí niekoľkých minút, čo je v praxi slabo použiteľné. Prehľad dĺžky času odpovede systému v závislosti na vstupe a parametroch je v tabuľke 7.3.

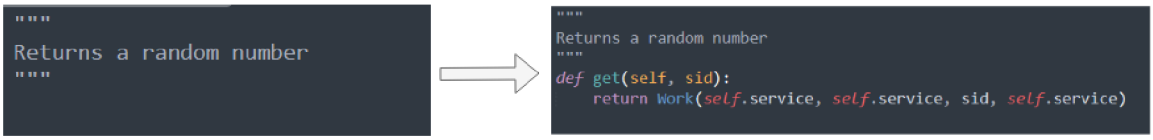
Tabuľka 7.3: Prehľad dĺžky časovej odozvy systému v rôznych situáciách. Každý vstup s algoritmom beam search mal šírku lúča nastavenú na 2.

Typ vstupu	**Čas**
Dokumentačný reťazec	5m 6s
Dokumentačný reťazec - beam search	9m 39s
Dokumentačný reťazec + hlavička	6m 46s
Dokumentačný reťazec + hlavička - beam search	17m 42s
Dokumentačný reťazec + hlavička + kód	4m 19s
Dokumentačný reťazec + hlavička + kód - beam search	8m 18s
Hlavička	4m 6s
Hlavička - beam search	10m 44s
Hlavička + kód	4m 13s
Hlavička + kód - beam search	8m 11s

Ako vidno z tabuľky 7.3, tieto časy odozvy sú príliš dlhé na reálne použitie. V tejto tabuľke nie je zahrnutá heuristika Top-K. Je to z dôvodu, že na viacerých príkladoch nedokázala vyprodukovať žiadny kód. To nastalo buď preto, že bola dosiahnutá maximálna dĺžka sekvencie skôr, ako bol dokončený AST, alebo heuristika zvolila token *other* namiesto tokenu *string* vtedy, keď to nebolo vhodné. Ako je uvedené v podkapitole 6.2, na tokeny typu *other* sa zavolá funkcia *eval()*, čo má v prípade nejakého reťazca za následok chybu v programe, napr. ak premenná s názvom tohto reťazca neexistuje.

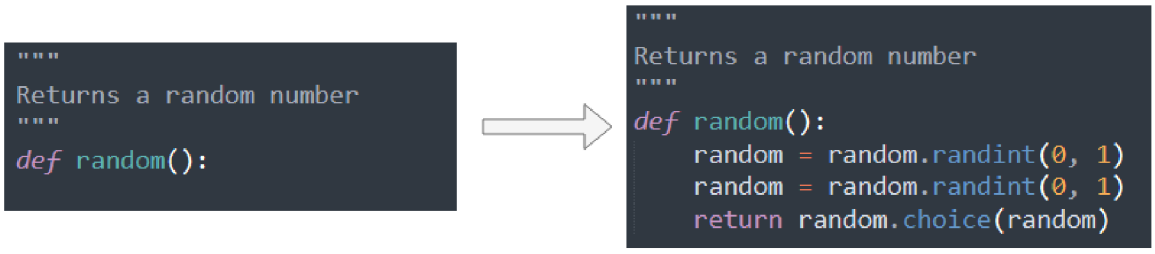
Čo sa týka samotného produkovaného kódu, jeho ukážky sú na obrázkoch 7.2 až 7.11. Každý z týchto obrázkov v tomto poradí zodpovedá jednému riadku v tabuľke 7.3.

Obr. 7.2: Ukážka vstupu a výstupu systému.

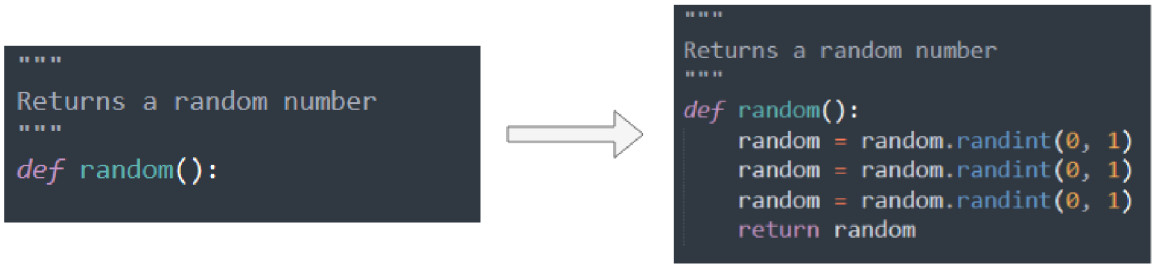


Obr. 7.3: Ukážka vstupu a výstupu systému. Použitá heuristika bola beam search so šírkou lúča 2.

[1]https://www.nvidia.com/en-in/geforce/products/10series/geforce-gtx-1050/



Obr. 7.4: Ukážka vstupu a výstupu systému.



Obr. 7.5: Ukážka vstupu a výstupu systému. Použitá heuristika bola beam search so šírkou lúča 2.

Ako je z obrázkov vidno, model nie je v tejto forme použiteľný v praxi. Podľa obrázkov 7.6 a 7.10, resp. aj 7.7 a 7.11 je zrejmé, že model neberie ohľad na obsah dokumentačného reťazca. Túto skutočnosť som otestoval na viacerých príkladoch, ktoré vzhľadom na veľkosť obrázkov nebudem všetky uvádzať.

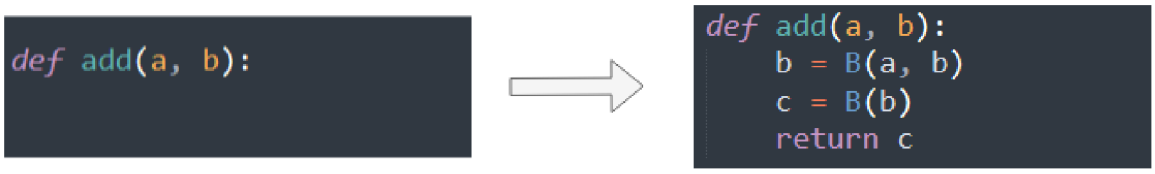
Ďalším z postrehov je, že metóda beam search produkuje väčšinou dlhší a komplexnejší kód, ako klasický greedy search. Príkladom toho sú páry obrázkov 7.8 a 7.9, alebo 7.4 a 7.5.

Model dokáže vyprodukovať aj sémanticky a logicky správny kód. Príkladom toho je napr. obrázok 7.6, ked model správne uhádol, že má vrátiť výsledok *c*. Tento kód je správny, obsahuje však prebytočné riadky. V tomto prípade metóda beam search dokázala nájsť optimálnejšie riešenie, ako je vidno na obrázku 7.7. Táto metóda oproti greedy search v tomto prípade ušetrila prebytočné 2 riadky kódu.

Z týchto poznatkov o testovaní som usúdil, že model dokáže pochopiť syntaktickú štruktúru kódu. Dokáže vyprodukovať syntakticky správny, dokonca aj spustiteľný či logicky správny kód. Problémom je, že nenašiel súvislosť medzi štruktúrou dokumentačných reťazcov a kódu. To mohlo byť zapríčinené tým, že dĺžka dokumentačných reťazcov je v priemere cca. 2,5-krát menšia, ako dĺžka kódu, viď. tabuľka 4.1 a obrázky 4.1 a 4.2. Ďalšou z vecí, ktorú som si všimol je, že model sa pravdepodobne až príliš zameriava na syntax, a neberie až tak do úvahy samotný obsah slov. Na druhej strane, ak je mu dodaná hlavička funkcie či

Obr. 7.6: Ukážka vstupu a výstupu systému.

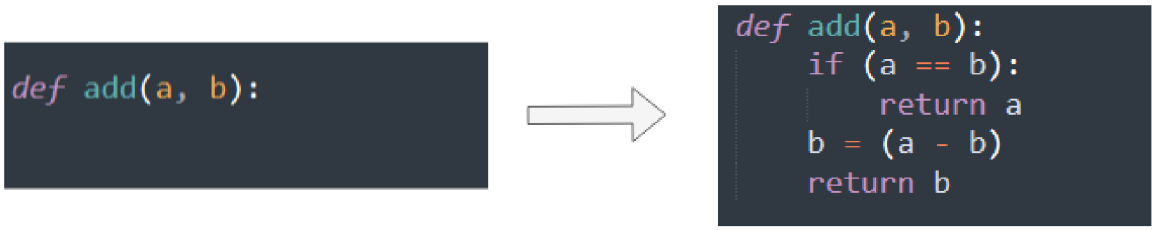
Obr. 7.7: Ukážka vstupu a výstupu systému. Použitá heuristika bola beam search so šírkou lúča 2.



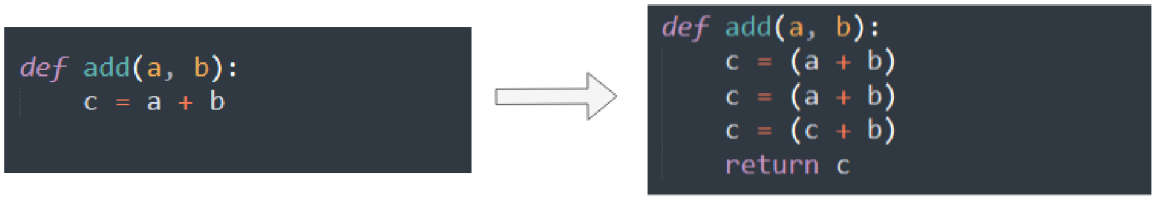
Obr. 7.8: Ukážka vstupu a výstupu systému.

prvý riadok, vyzerá to tak, že na základe jeho kontextu dokáže generovať kód, ktorý aspoň zdanlivo súvisí so vstupom.

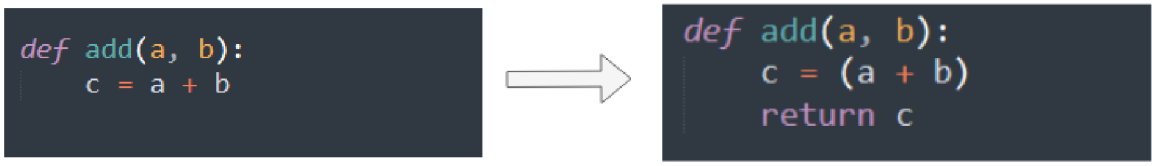
Čo sa týka časovej odozvy, táto je ne reálne použitie príliš vysoká. Jeden z dôvodov je použitá grafická karta pri vyhodnocovaní, ktorá je na úrovni priemerného PC. Ak by tento model bol spustený na nejakom serveri vo výpočtovom centre, určite by čas odozvy bol rádovo nižší. Navyše, model bol trénovaný od nuly, bez použitia predtrénovaných parametrov. Nedá sa rozmermi porovnávať s modelmi, ako napr. GPT-3. Myslím, že tento model slúži ako dôkaz, že úloha generovania či doplňovania kódu je aspoň teoreticky realizovateľná aj s menšími modelmi.



Obr. 7.9: Ukážka vstupu a výstupu systému. Použitá heuristika bola beam search so šírkou lúča 2.



Obr. 7.10: Ukážka vstupu a výstupu systému.



Obr. 7.11: Ukážka vstupu a výstupu systému. Použitá heuristika bola beam search so šírkou lúča 2.

Kapitola 8

Záver

Cieľom tejto práce bolo navrhnúť a implementovať systém na generovanie kódu z textového popisu funkcionality. Napriek tomu, že model nedosiahol výsledky porovnateľné s najlepšími modelmi v tejto oblasti, či výsledky hodné komerčného použitia, sa dá povedať, že cieľ bol splnený. Bol navhrnutý a implementovaný systém, ktorý dokázal na základe dostupných dát generovať kód.

V prvej časti práci som sa venoval základným konceptom v oblasti spracovania jazyka a strojového učenia. Zoznámil som sa s modelmi typu Transformer, či GPT. Naučil som sa, ako sa dajú použiť a spracovať dáta získané z veľkých repozitárov. Zistil som, aký je momentálny postup v oblasti generovania kódu, aké techniky a modely sa používajú, a aké sú trendy v tejto oblasti. Rozhodol som sa navhrnúť systém, ktorý by dokázal generovať kód aj s menšími rozmermi modelu, či menej dátami. Inšpiroval som sa pri tom viacerými prácami, ktoré sa snažili extrahovať syntaktické informácie z kódu, na základe ktorých potom trénovali modely. Rozhodol som sa túto myšlienku prevziať a rozšíriť ju aj na extrakciu syntaktických informácií z textových popisov kódu. Výsledky som nakoniec porovnal s jednoduchým prototypom systému, vyhodnotil som ich ako číselne, tak aj subjektívne z pohľadu programátora.

Pri vyhodnotení som najprv porovnával systém s jeho prototypom na základe presnosti uhádnutých slov. Prototyp v tejto metrike dosiahol presnosť 60%, zatiaľ čo systém, ktorému sa venujem v celej práci dosiahol presnosť až 85%. Je pravda, že model použitý v tomto systéme obsahoval oproti jeho prototypu takmer 2,5-krát viac trénovateľných parametrov, napriek tomu jeho tréning trval o 60% kratšie, a veľkosť jeho datasetu bola 3,2-krát menšia. Toto pripisujem z veľkej časti použitiu syntaktickej analýzy v systéme, ktoré prototyp neobsahoval. Výsledný kód produkovaný systémom bol syntakticky správny, v niektorých prípadoch dokonca aj logicky. Problémom je, že čas vyhodnotenia jednej požiadavky bol príliš dlhý na použitie systému v praxi. Ďalším zo získaných poznatkov je, že model nenašiel príliš veľkú súvislosť medzi syntaktickou štruktúrou textových popisov a kódu. Napriek tomu si myslím, že použitie tohoto prístupu na trochu väčších modeloch by mohlo priniesť dobré výsledky.

Táto práca ma naučila veľa vecí o jazykových modeloch, spracovaní dát, a použití týchto modelov na veľkých objemoch dát. V práci by som mohol pokračovať tak, že by som sa viac sústredil na filtrovanie dát, konkrétne na to, aby kód a textové popisy mali podobnú dĺžku. Ďalšou z možností by bolo využitie iného typu modelu, ktorý by na tejto úlohe dosiahol lepšie výsledky.

Literatúra

[1] Agashe, R., Iyer, S. a Zettlemoyer, L. *JuICe: A Large Scale Distantly Supervised Dataset for Open Domain Context-based Code Generation.* 2019.

[2] Ba, J. L., Kiros, J. R. a Hinton, G. E. *Layer Normalization.* 2016.

[3] Black, P. E. greedy algorithm. *Dictionary of Algorithms and Data Structures.* Február 2005. Dostupné z: https://www.nist.gov/dads/HTML/greedyalgo.html.

[4] Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. et al. *Language Models are Few-Shot Learners.* 2020.

[5] Bussler, F. Will The Latest AI Kill Coding? *Towards Data Science.* Júl 2020. Dostupné z: https://towardsdatascience.com/will-gpt-3-kill-coding-630e4518c04d.

[6] Freitag, M. a Al Onaizan, Y. Beam Search Strategies for Neural Machine Translation. *Proceedings of the First Workshop on Neural Machine Translation.* Association for Computational Linguistics. 2017. DOI: 10.18653/v1/w17-3207. Dostupné z: http://dx.doi.org/10.18653/v1/W17-3207.

[7] Gage, P. A New Algorithm for Data Compression. *C Users J.* USA: R & D Publications, Inc. február 1994, zv. 12, č. 2, s. 23–38. ISSN 0898-9788.

[8] Gautam, H. Word Embedding: Basics. *Medium.* Marec 2020. Dostupné z: https://medium.com/@hari4om/word-embedding-d816f643140.

[9] Goodfellow, I., Bengio, Y. a Courville, A. *Deep Learning.* MIT Press, 2016. Adaptive Computation and Machine Learning series. ISBN 9780262337373. Dostupné z: https://books.google.sk/books?id=omivDQAAQBAJ.

[10] Holtzman, A., Buys, J., Du, L., Forbes, M. a Choi, Y. *The Curious Case of Neural Text Degeneration.* 2020.

[11] Hu, X., Li, G., Xia, X., Lo, D. a Jin, Z. Deep code comment generation. In:. Máj 2018, s. 200–210. DOI: 10.1145/3196321.3196334. ISBN 978-1-4503-5714-2.

[12] Husain, H., Wu, H.-H., Gazit, T., Allamanis, M. a Brockschmidt, M. *CodeSearchNet Challenge: Evaluating the State of Semantic Code Search.* 2020.

[13] Jurafsky, D. a Martin, J. H. *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition.* 1st. USA: Prentice Hall PTR, 2000. ISBN 0130950696.

[14] Kim, J. a Lee, Y. A Study on Abstract Syntax Tree for Development of a JavaScript Compiler. *International Journal of Grid and Distributed Computing*. Jún 2018, zv. 11, s. 37–48. DOI: 10.14257/ijgdc.2018.11.6.04.

[15] Kim, S., Zhao, J., Tian, Y. a Chandra, S. *Code Prediction by Feeding Trees to Transformers*. 2021.

[16] Kudo, T. a Richardson, J. *SentencePiece: A simple and language independent subword tokenizer and detokenizer for Neural Text Processing*. 2018.

[17] Li, J., Wang, Y., Lyu, M. R. a King, I. Code Completion with Neural Attention and Pointer Networks. *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence*. International Joint Conferences on Artificial Intelligence Organization. Jul 2018. DOI: 10.24963/ijcai.2018/578. Dostupné z: http://dx.doi.org/10.24963/ijcai.2018/578.

[18] Raychev, V., Vechev, M. a Yahav, E. Code Completion with Statistical Language Models. *SIGPLAN Not.* New York, NY, USA: Association for Computing Machinery. jún 2014, zv. 49, č. 6, s. 419–428. DOI: 10.1145/2666356.2594321. ISSN 0362-1340. Dostupné z: https://doi.org/10.1145/2666356.2594321.

[19] Sherstinsky, A. Fundamentals of Recurrent Neural Network (RNN) and Long Short-Term Memory (LSTM) network. *Physica D: Nonlinear Phenomena*. Elsevier BV. Mar 2020, zv. 404, s. 132306. DOI: 10.1016/j.physd.2019.132306. ISSN 0167-2789. Dostupné z: http://dx.doi.org/10.1016/j.physd.2019.132306.

[20] Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I. a Salakhutdinov, R. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *Journal of Machine Learning Research*. 2014, zv. 15, č. 56, s. 1929–1958. Dostupné z: http://jmlr.org/papers/v15/srivastava14a.html.

[21] Svyatkovskiy, A., Deng, S. K., Fu, S. a Sundaresan, N. *IntelliCode Compose: Code Generation Using Transformer*. 2020.

[22] Taylor, A., Marcus, M. a Santorini, B. The Penn Treebank: An overview. Január 2003. DOI: 10.1007/978-94-010-0201-1_1.

[23] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L. et al. *Attention Is All You Need*. 2017.

[24] Wang, Y. a Li, H. *Code Completion by Modeling Flattened Abstract Syntax Trees as Graphs*. 2021.

[25] Warner, C., Bies, A., Brisson, C. a Mott, J. Addendum to the Penn Treebank II style bracketing guidelines: BioMedical Treebank annotation. December 2004.

[26] Wood, T. What is the Softmax Function? *DeepAI*. Máj 2019. Dostupné z: https://deepai.org/machine-learning-glossary-and-terms/softmax-layer.

[27] Yin, P. a Neubig, G. *A Syntactic Neural Model for General-Purpose Code Generation*. 2017.

Príloha A

Obsah pamäťového média

BP.zip
- **model1/** - prvý model - obsah uvedený na nasledujúcej strane
- **model2/** - druhý model
  - **checkpoints/** - natrénované parametre modelu
  - **main.py** - hlavná časť programu
  - **download.py** - stiahnutie a extrakcia dát
  - **preprocess_data.py** - spracovanie dát
  - **tokenizer.py** - tokenizácia dát
  - **create_corpus.py** - vytvorenie korpusu pre tokenizér
  - **train.py** - tréning modelu
  - **transformer.py** - implementácia Transformera
  - **evaluate.py** - testovanie modelu na dátach
  - **grammar_check.py** - kontrola gramatiky
  - **predict.py** - generovanie kódu
  - **server.py** - server na posielanie žiadostí na generovanie kódu
  - **setup.py** - stiahnutie potrebných dát pre parser dokumentačných reťazcov
  - **code.model** - natrénované parametre tokenizéra pre kód
  - **code.vocab** - natrénované parametre tokenizéra pre kód
  - **doc.model** - natrénované parametre tokenizéra pre dokumentačné reťazce
  - **doc.vocab** - natrénované parametre tokenizéra pre dokumentačné reťazce
  - **Makefile** - makefile pre inštaláciu potrebných knižníc
  - **requirements.txt** - zoznam požadovaných knižníc pre Python
- **sublime-plugin/** - plugin pre generovanie kódu v editore Sublime Text
- **README.md** - návod na spustenie systému
- **BP-source/** - zdrojové súbory na kompiláciu bakalárskej práce
- **BP.pdf** - bakalárska práca vo formáte PDF
- **plagat.png** - stručný plagát k práci

model1/ - prvý model

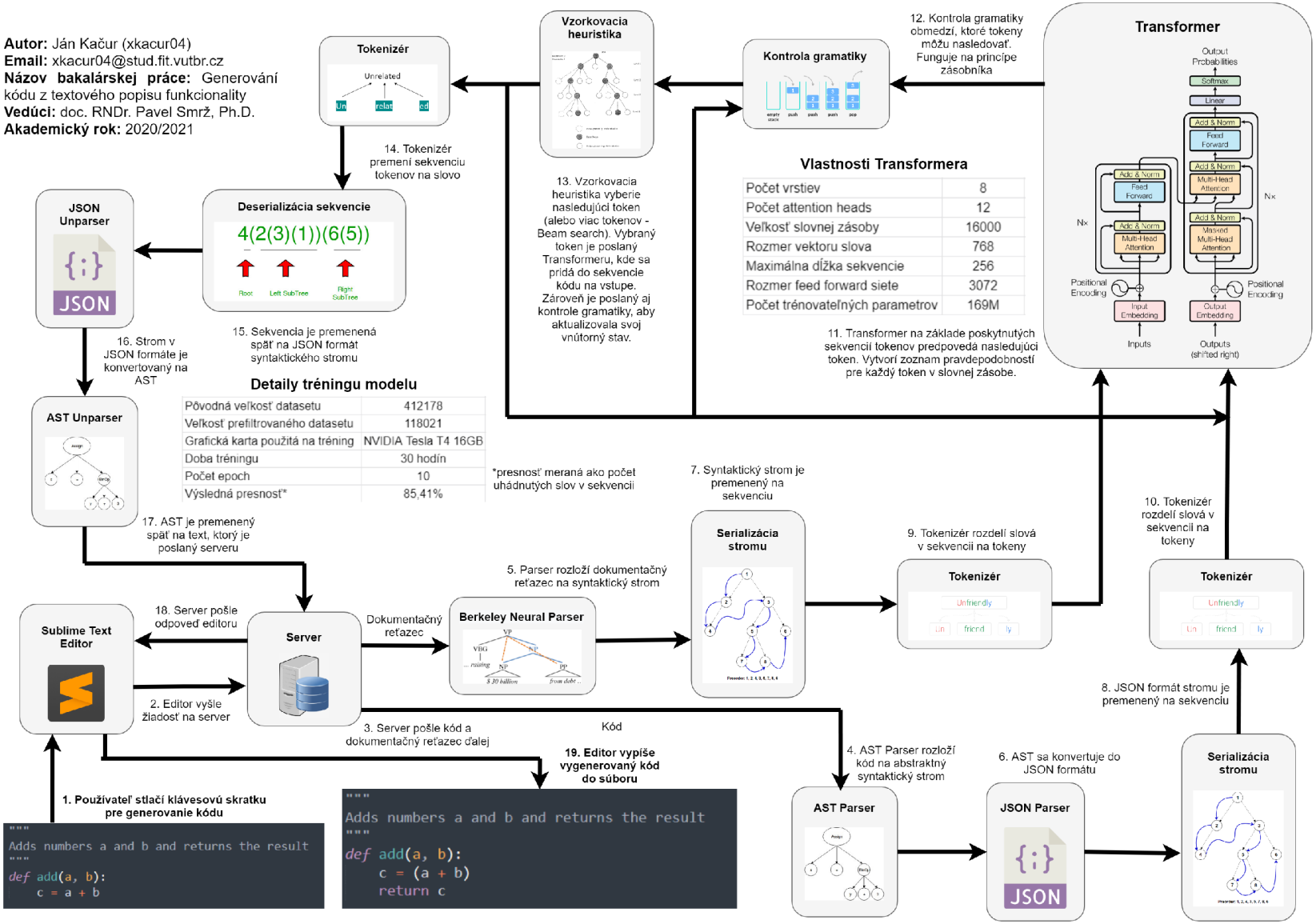
- **checkpoints/** - natrénované parametre modelu
- **transformer.py** - implementácia Transformera
- **train.py** - tréning modelu
- **server.py** - server na posielanie žiadostí na generovanie kódu
- **predict.py** - generovanie kódu
- **evaluate.py** - testovanie modelu na dátach
- **index.html** - užívateľské rozhranie pre generovanie kódu
- **encode_tokens.py** - tokenizácia
- **train_tokenizer.py** - tréning tokenizéra
- **evaluate.py** - testovanie modelu na dátach
- **extract_data.py** - spracovanie dát
- **code.model** - natrénované parametre tokenizéra pre kód
- **code.vocab** - natrénované parametre tokenizéra pre kód
- **doc.model** - natrénované parametre tokenizéra pre dokumentačné reťazce
- **doc.vocab** - natrénované parametre tokenizéra pre dokumentačné reťazce

Príloha B

Plagát

Na nasledujúcej strane je priložená zmenšená verzia plagátu k tejto práci (obrázok B.1). Originálna verzia vo vysokom rozlíšení sa nachádza na SD karte. Tu uvádzam zdroje použitých obrázkov v plagáte:

- **Sublime Text logo** - https://www.sublimehq.com/images/sublime_text.png
- **Server logo** - https://www.nicepng.com/png/detail/207-2073247_download-database-server-icon.png
- **Berkeley Neural Parser logo** - https://www.researchgate.net/profile/Mohit-Bansal-10/publication/220873576/figure/fig1/AS:669396477419527@1536607965393/A-PP-attachment-error-in-the-parse-output-of-the-Berkeley-parser-on-Penn-Treebank.png
- **AST Parser logo** - https://miro.medium.com/max/261/0*ykaApIklGcJ7Qzhw
- **JSON Parser logo** - https://miro.medium.com/max/512/1*4e5PgSBgxgFolqvob9x_Wg.png
- **Serializácia logo** - https://www.techiedelight.com/wp-content/uploads/Preorder-Traversal.png
- **Tokenizér logo** - https://www.thoughtvector.io/blog/subword-tokenization/Subword%20Units.svg
- **Tokenizér logo 2** - https://miro.medium.com/max/726/1*ALFCq4JyyqGJzeH7JjOKww.png
- **Deserializácia logo** - https://media.geeksforgeeks.org/wp-content/uploads/BtreeRepresent.jpg
- **Kontrola gramatiky logo** - https://cdn.programiz.com/sites/tutorial2program/files/stack.png
- **Vzorkovacia heuristika logo** - https://d3i71xaburhd42.cloudfront.net/6901af3198b79e4054f842c9ff28ccb98ead8a59/4-Figure1-1.png
- **Transformer** - [23]



Obr. B.1: Zmenšená verzia plagátu k práci