Politechnika Warszawska

WYDZIAŁ ELEKTRONIKI
I TECHNIK INFORMACYJNYCH

Instytut Informatyki

Praca dyplomowa inżynierska

na kierunku Informatyka
w specjalności Inżynieria Systemów Informacyjnych

Generowanie automatycznych podpowiedzi w środowiskach do programowania

Rafał Lewanczyk
Numer albumu 293140

promotor
dr inż. Paweł Zawistowski

WARSZAWA 2021

Generowanie automatycznych podpowiedzi w środowiskach do programowania

Streszczenie.
Uzupełnianie kodu jest funkcją odpowiedzialną za przewidywanie tego co chce, lub może napisać programista. Poprawnie działająca może w znaczący sposób wspomóc oraz uefektywnić jego pracę. Istnieje wiele podejść do tego problemu np. przy pomocy metod słownikowych lub parsowania na bieżąco pisanego kodu.

W tej pracy omawiam podejście polegające na zastosowaniu metod uczenia maszynowego. Generuje ono posortowaną według prawdopodobieństwa wystąpienia listę sugestii, które mogą zostać wykorzystane przez programistę w trakcie pracy nad programem.

Opisuję badane przeze mnie architektury, oraz porównuję je pod względem skuteczności. Wymieniam również napotkane wyzwania.

Końcowy model został wytrenowany na 9000 plikach w języku Python, pochodzących z publicznie dostępnych repozytoriów na serwisie GitHub, oraz oceniony na 4500 plikach. Jego skuteczność wyniosła 68% dla 5 najlepszych predykcji, a średni czas odpowiedzi 18 ms.

Model ten został zaimplementowany jako wtyczka do środowiska SublimeText 3.

Słowa kluczowe: Uczenie maszynowe, autouzupełnianie, kod źródłowy

Generating code autocompletions for Integrated Development Environments

Abstract. Code autocompletion is a feature that offers suggestions for what a software developer may want to write. Correct suggestions improve efficiency of programmers job. There are a lot of forms of code autocompletion, for example using dictionary methods or parsing code online.

In this paper, I propose approach for code autocompletion using machine learning. It generates ranked by its probabilities of occurrences list of suggestions, which can be used by software developer at edit time.

I describe explored architectures and compare them to each other in order to select the best one. I also discuss challenges regarding this approach.

The final model was trained using 9000 Python source files from publicly available repositories from service GitHub and evaluated on 4500 files. The evaluation results in 68% accuracy for top 5 suggestions, and mean time of response resulted in 18 ms.

The system is available as a plugin for integrated development environment SublimeText 3.

Keywords: Machine Learning, Autocomplete, Source Code

załącznik nr 3 do zarządzenia
nr 28 /2016 Rektora PW

Politechnika Warszawska

……..
miejscowość i data

……..
imię i nazwisko studenta
……..
numer albumu
…….…….
kierunek studiów

OŚWIADCZENIE

Świadomy/-a odpowiedzialności karnej za składanie fałszywych zeznań oświadczam, że niniejsza praca dyplomowa została napisana przeze mnie samodzielnie, pod opieką kierującego pracą dyplomową.

Jednocześnie oświadczam, że:

- niniejsza praca dyplomowa nie narusza praw autorskich w rozumieniu ustawy z dnia 4 lutego 1994 roku o prawie autorskim i prawach pokrewnych (Dz.U. z 2006 r. Nr 90, poz. 631 z późn. zm.) oraz dóbr osobistych chronionych prawem cywilnym,
- niniejsza praca dyplomowa nie zawiera danych i informacji, które uzyskałem/-am w sposób niedozwolony,
- niniejsza praca dyplomowa nie była wcześniej podstawą żadnej innej urzędowej procedury związanej z nadawaniem dyplomów lub tytułów zawodowych,
- wszystkie informacje umieszczone w niniejszej pracy, uzyskane ze źródeł pisanych i elektronicznych, zostały udokumentowane w wykazie literatury odpowiednimi odnośnikami,
- znam regulacje prawne Politechniki Warszawskiej w sprawie zarządzania prawami autorskimi i prawami pokrewnymi, prawami własności przemysłowej oraz zasadami komercjalizacji.

Oświadczam, że treść pracy dyplomowej w wersji drukowanej, treść pracy dyplomowej zawartej na nośniku elektronicznym (płycie kompaktowej) oraz treść pracy dyplomowej w module APD systemu USOS są identyczne.

.....
czytelny podpis studenta

Spis treści

1. Wstęp

Środowiska programistyczne ułatwiają i przyśpieszają pisanie kodu m.in. poprzez proponowanie słów kluczowych oraz nazw zdefiniowanych w programie. Dzięki tej funkcji programista nie musi pisać ręcznie w całości długiej nazwy zmiennej lub metody, a w niektórych przypadkach nie musi pisać jej wcale. Podejście to charakteryzuję się parsowaniem na bieżąco kodu programu, oraz na podstawie reguł rządzących danym językiem programowania, proponowania nazw znajdujących sie w jego drzewie rozkładu, lub zaproponowaniu któregoś ze słów kluczowych na podstawie wcześniejszych ich wystąpień. Przykładem takiego zachowania może być zaproponowanie bloku $else$ po bloku if. Jednak takie podejście wiążę się z wieloma wadami.

- Stworzenie takiego systemu uzupełniającego wiąże się ze zdefiniowaniem wielu skomplikowanych reguł, które różnią się dla każdego języka programowania.
- System generujący propozycje nie uwzględnia kontekstu pisanego kodu. Na przykład w kodzie aplikacji internetowej można zaobserwować wiele powtarzających się szablonów, które będą się różnić od szablonów występujących w kodzie jądra systemu operacyjnego.
- Taki system ma problem z ocenieniem, która podpowiedź ma największe prawdopodobieństwo pojawienia się, przez co często podaje je, w z góry założonej kolejności np. posortowane leksykograficznie.

1.1. Problem

Problem

Istnieje wiele rodzajów uzupełniania kodu:

- przewidzenie kolejnego słowa (tokenu),
- przewidzenie dłuższej sekwencji słów (na przykład dokończenie linijki),
- generowanie funkcji, na podstawie jej opisu w komentarzu,
- uzupełnianie brakujących linii lub tokenów, w zaznaczonych miejscach w kodzie.

W tej pracy skupiam się na pierwszym z rodzajów tego problemu. Moim celem jest stworzenie systemu, który rozwiąże wcześniej wymienione problemy, oraz zaimplementowanie go jako wtyczka do środowiska SublimeText.

Kod a język naturalny

Metody statystyczne oraz rekurencyjne sieci neuronowe mają swoje zastosowanie w bardzo dużej ilości dziedzin, m.in. predykcji, klasyfikacji lub filtracji sygnałów. Rozwiązywany przeze mnie problem jest specjalnym przypadkiem problemu klasyfikacji, który opiszę w dalszej części pracy. Z tego względu zdecydowałem się właśnie na zastosowanie ich w moim rozwiązaniu.

Języki programowania dzielą wiele cech wspólnych z językiem naturalnym. Jednym z zastosowań języka naturalnego jest opisywanie algorytmów w skończonej liczbie

kroków, co jest również jedynym zastosowaniem języków programowania. Logika stojąca za wyrażaniem kolejnych kroków jest taka sama. Oba typy języków używane są do komunikacji, naturalny używany pomiędzy ludźmi, natomiast programowania między człowiekiem a komputerem. Jednak najważniejszą łączącą je cechą jest ich powtarzalność. W obu z dużym prawdopodobieństwem po jednym słowie może wystąpić tyko względnie niewielki zbiór innych słów.
Istotną różnicą dzielącą te rodzaje języków jest możliwość nadawania dowolnych nazw obiektom oraz metodom w językach programowania. Powoduje to, że nie można objąć wszystkich słów w słowniku danych treningowych. Słowa tego typu nazywane są słowami poza słownikiem. Różnica ta jest na tyle znacząca, że powoduje konieczność wprowadzenia zmian adaptacyjnych w metodach dotyczących modelowania języka naturalnego.

Zastosowania

Główny zastosowaniem tworzonego systemu jest usprawnienie pracy programisty. Jednak przy założeniu, że model działa dobrze istnieje więcej przypadków użycia:

- Tworzenie kodu na urządzeniu mobilnym. W dzisiejszych czasach urządzenia mobilne mają ogromne możliwości. Jedyną rzeczą, która je powstrzymuje przed użyciem ich w celu rozwoju oprogramowania, jest mała klawiatura dotykowa nie udostępniająca szybkiego dostępu do znaków specjalnych. Wtyczka mogłaby znacznie usprawnić pisanie poprzez przewidywanie znaków specjalnych (z czym jak pokażę później radzi sobie bardzo dobrze), jak i długich, niewygodnych do napisania nazw występujących w kodzie.
- Szukanie błędów w kodzie. Model może obliczyć prawdopodobieństwo wystąpienie następnego tokenu po czym sprawdzić czy pokrywa się on z faktycznie występującym tokenem. W ten sposób możemy określić miejsce w kodzie w którym należy spodziewać się, że został popełniony błąd.
- Kompresja kodu. Modele Sequence2Sequence sprawdzają się w zadaniu kompresji. Model mógłby nauczyć się wygenerować resztę programu na podstawie kilku pierwszych tokenów. W ten sposób, zamiast zapisywać cały kod źródłowy moglibyśmy zapamiętywać jedynie kilka krótkich sekwencji.

1.2. Cel

Celem tej pracy jest stworzenie modelu uczenia maszynowego przewidującego kolejny token podczas pisania kodu programu w języku python, oraz implementacja go jako wtyczki do zintegrowanego środowiska programistycznego SublimeText3.

2. Przegląd literatury

W ciągu ostatnich kilku lat przetwarzanie języka naturalnego bardzo się rozwinęło. Wraz z kolejnymi badaniami udało się uzyskać coraz lepsze rezultaty. Jednak dział badający zachowanie tych modeli na językach programowania jest nowy, co możemy zaobserwować po analizie zbioru prac [1] z nim związanych. Jest w nim dużo miejsca na nowe podejścia oraz badania. W tym rozdziale omówię prace istotne lub podobne do mojej.

Zadanie modelowania sekwencji

Problem przewidywania kodu źródłowego jest przypadkiem zadania modelowania sekwencji. Polega ono na określeniu prawdopodobieństwa zdania w języku. Poza przypisaniem prawdopodobieństwa każdej z sekwencji słów model językowy również określa, które ze słów występujących w słowniku ma największe szanse na pojawienie się jako następny element danej sekwencji.
Istnieje wiele metryk służących do oceniania działania tego typu modeli np. perpleksja. Jednak jako, że modele w tej pracy tworzone są z myślą o wykonywaniu określonego zadania, będą one oceniane oraz porównywane pod kątem radzenia sobie w danym zadaniu. Kryteria te są dokładnie opisane w rozdziale 3.4.

2.1. Metody uczenia maszynowego w przewidywaniu języków programowania

Subhasis Das, Chinmayee Shah [2] porównują ze sobą modele

- model z wagami o stałej długości okna (fixed window weight model),
- model macierzy wektorów (matrix vector model),
- sieć jednokierunkowa (feed-forward neural network),
- sieć jednokierunkowa ze skupieniem uwagi (feed-forward model with soft attention),
- model rekurencyjny z warstwą GRU.

Kod wejściowy jest poddany tokenizacji przy pomocy wyrażeń regularnych. Oceniane odbywa się poprzez dokładne dopasowanie pierwszego przewidzianego tokenu oraz na podstawie 3 najlepszych sugestii. Do treningu oraz testów używane są kody bibliotek Django, Twisted oraz jądra systemu Linux. Połowa plików źródłowych jednego z projektów używana jest jako zbiór treningowy natomiast druga połowa jako zbiór walidacyjny. Wszystkie modele osiągają dokładność przewidzeń równą około 60% dla 3 najlepszych sugestii, najlepiej radzi sobie model miękka uwagą z dokładnością 66.5%. Problem słów poza słownikiem rozwiązany jest przy pomocy słownika przypisującego token o nieznanej wartości do słowa które wpisał użytkownik.

Hellendoorn i Devanbu przeprowadzili eksperyment polegający na wykonaniu 15000 predykcji dla środowiska Visual Studio. Jako zbioru danych używają 14000 plików źródłowych w języku Java. W swojej pracy porównują skuteczność modelu n-gram z modelami rekurencyjnymi. Prezentują również dynamicznie aktualizowane modele n-gram działające w zagnieżdżonym zasięgu, rozwiązując w ten sposób problem skończonego słownika

oraz znacznie usprawniając sugestie. Rezultaty tej pracy pokazują, że pomimo znacznie lepszych wyników sieci rekurencyjnych w zadaniu modelowania języka naturalnego, modele n-gram w niektórych przypadkach radzą sobie lepiej z przewidywaniem kodu. Jednym z przytoczonych przykładów są wbudowane funkcje języka np. $len()$ w języku python, dla których głębokie sieci działają lepiej, jednak przegrywają przy często występujących, zróżnicowanych, mało popularnych bibliotekach zewnętrznych, w których model n-gram naturalnie radzi sobie lepiej, jednak przegrywa pod innymi względami. Swoje eksperymenty przeprowadzają dla stałych wartości hiperparametrów.

Pythia [3] stworzona przez zespół programistów Microsoftu, w którego skład wchodzą Alexey Svyatkovskiy, Shengyu Fu, Ying Zhao, Neel Sundaresan jest rozszerzeniem do wtyczki IntelliSense w środowisku Visual Studio Code. W swojej pracy używają wariacji modeli sieci rekurencyjnych. Jako zbiór treningowy użyte jest 2700 projektów z serwisu github [4] które wcześniej zostały poddane rozbiorowi na drzewa składniowe. Model osiąga skuteczność wynoszącą 92% dla najlepszych 5 sugestii oraz czas odpowiedzi w okolicach 100 ms. Eksperymentu tego nie będę w stanie dokładnie odtworzyć ze względu na inne przygotowanie danych wejściowych oraz przez inny zbiór danych (zbiór wykorzystany przez Pythie nie został udostępniony).

2.2. Rozwiązania komercyjne

W dużej mierze do powstania tej pracy przyczyniły się istniejące już rozwiązania komercyjne. Niestety ze względów licencyjnych nie są ujawnione dokładnie mechanizmy stojące za ich działaniem, metody użyte do treningu oraz dokładna skuteczność, przez co niemożliwe jest porównanie uzyskanych przeze mnie wyników z tymi podejściami.

Tabnine [5] jest wtyczką do najpopularniejszych środowisk programistycznych realizującą predykcję kolejnego tokenu w większości stosowanych języków programowania. Do jej treningu zostało wykorzystane 2 miliony projektów ze strony github [4]. Wtyczka opiera się na GPT-2, które używa architektury transformerów. W jej skład wchodzą również zaimplementowane przez twórców sztywne reguły dotyczące języka. Podejście to jest bardzo nowatorskie przez wykorzystanie jeszcze nie zbadanych dokładnie modeli oraz różni się od przedstawionego w tej pracy.

Open AI [6] realizuje generowanie kodu na podstawie opisu jego działania w komentarzu. Jest to połączenie zadania zrozumienia języka naturalnego przez maszynę, z zadaniem klasyfikacji. Słownik zamiast składać sie z pojedyńczych tokenów składa sie z całych funkcji, a na wejściu modelu otrzymujemy sekwencje słów zamiast poprzedzający kod.

2.3. Różnice w rozwiązaniach

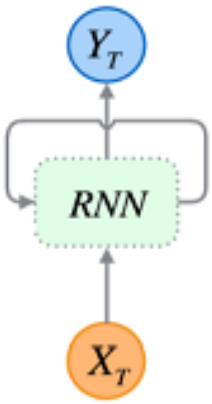
Moja praca różni się od powyższych tym, że skupiam się na dokładniejszym zbadaniu sieci rekurencyjnych, jako że w powyższych pracach nie zostały uwzględnione różne wariacje ich układu. Próbuję również uogólnić przewidywany kod poprzez trening na znacznie bardziej zróżnicowanych bibliotekach, oddających częstotliwość ich używania, aby sprawić by wtyczka była użyteczna w bardziej ogólnych zastosowaniach.

2.4. Sieci neuronowe

Sieć neuronowa to model matematyczny, realizujący obliczenia poprzez rzędy elementów przetwarzających zwanych warstwami. W skład każdej warstwy wchodzi określona liczba neuronów. Odpowiedzialnością neuronu jest wykonanie pewnej podstawowej operacji na swoim wejściu oraz przekazanie dalej wyniku.

W swojej pracy skupiam się na rekurencyjnych sieciach neuronowych (RNN). Sieci te charakteryzują się możliwością zapamiętywania stanu, co jest przydatne w przetwarzaniu sekwencji, w odróżnieniu do tradycyjnej sieci, w której zakładamy, że kolejne przetwarzane przez model elementy są niepowiązane. Wykorzystuję dwa rodzaje warstw sieci w różnych konfiguracjach. Zamysł stojący za sieciami rekurencyjnymi został przedstawiony na rysunku 2.1

Rysunek 2.1. Działanie rekurencyjnej sieci neuronowej



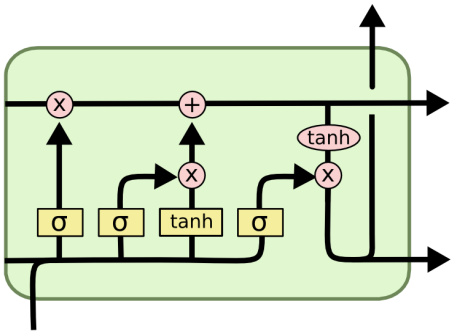
LSTM (Long Short-Term Memory) [7]

Architektura LSTM jest w tym momencie najpopularniejszą w zadaniach generowania sekwencji. Komórka LSTM wyposażona jest w trzy bramki: 'wejścia', 'zapomnienia', 'wyjścia'. Bramki te kontrolują które informacje zostaną zapamiętane, zapomniane oraz przewidziane na podstawie wytrenowanych parametrów. Komórka LSTM została przedstawiona na rysunku 2.2.

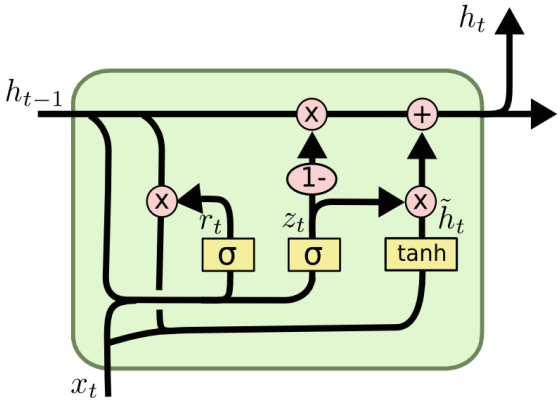
GRU (Gated Recurrent Unit) [8]

Komórka GRU posiada dwie bramki: 'wejścia' oraz 'zapomnij'. Podobnie do sieci LSTM

Rysunek 2.2. Komórka warstwy LSTM
źródło: https://morioh.com/p/34ad430cb59b



Rysunek 2.3. Komórka warstwy GRU
źródło: https://morioh.com/p/34ad430cb59b



decydują one o tym które informacje mają zostać zachowane, a które zapomniane. Komórka GRU została przedstawiona na rysunku 2.3.

2.5. Warstwa CNN

W części badanych przeze mnie modeli używam również warstwy konwolucyjnej CNN (Convolutional neural network). Składa się ona z szeregu warstw splotowych, które splatają się z mnożeniem lub iloczynem skalarnym. Jednym z zastosowań sieci CNN jest odnajdywanie wzorców, przez co sprawdza się ona w zadaniu przetwarzania języka naturalnego.

2.6. Modele statystyczne N-gram

Model N-gram jest modelem językowym mającym bardzo szerokie zastosowanie we wszystkich dziedzinach związanych z mową oraz pismem. N-gramy opierają się na statystykach i służą do przewidywania kolejnego elementu sekwencji.

Modele N-gram szacują prawdopodobieństwo kolejnego słowa na podstawie wcześniej występujących słów za pomocą prawdopodobieństwa warunkowego. Zatem model przewiduje x_i na podstawie danego ciągu $x_{i-(n-1)}, ..., x_{i-1}$. Zapisując wyrażenie przy pomocy prawdopodobieństwa otrzymujemy

$$P(x_i | x_{i-(n-1)}, ..., x_{i-1}),$$

co z definicji prawdopodobieństwa warunkowego możemy przekształcić do wyrażenia

$$\frac{P(x_i, x_{i-(n-1)}, ..., x_{i-1})}{P(x_{i-(n-1)}, ..., x_{i-1})}.$$

Poszczególne prawdopodobieństwa znajdujemy poprzez zbieranie statystyk ich wystąpienia w dużym korpusie tekstu. Zatem powyższy wzór możemy przybliżyć w następujący sposób

$$\frac{liczba(x_i, x_{i-(n-1)}, ..., x_{i-1})}{liczba(x_{i-(n-1)}, ..., x_{i-1})}.$$

2.7. Wykorzystany zbiór danych

W celach treningowych wykorzystałem zbiór danych zebranych przez grupę SRILAB [9], w celu stworzenia narzędzia DeepSyn. Zbiór składa się ze 150 tysięcy plików źródłowych w języku Python, pobranych z serwisu GitHub [4], po usunięciu powtarzających się plików, usunięciu kopii już istniejących repozytoriów oraz zachowaniu tylko programów, z których można wygenerować poprawne drzewo rozkładu mające co najmniej 30 tysięcy węzłów. Wszystkie projekty wchodzące w skład zbioru są wydane na licencjach pozwalających na kopiowanie oraz rozpowszechnianie takich jak MIT, BSD lub Apache. Zbiór jest podzielony na 100 tysięcy plików treningowych oraz 50 tysięcy plików walidacyjnych. W swoich eksperymentach użyłem podzbioru dostarczonych danych wynoszącego około 1% oryginalnego zbioru. Decyzja ta wynika z dwóch powodów. Jak wspomniał w swojej pracy Hellendoorn i Devanbu[10] modele uczenia głębokiego nie skalują dobrze przy dużych danych. Trening na pełnym zbiorze byłby niemożliwy ze względu na ograniczenia czasowe. Dla przykładu trening małego modelu LSTM o 512 komórkach oraz warstwą zanurzenia (Embedding) na 10 tysiącach plików zabiera około 1.5 godziny dla jednej epoki, na karcie graficznej GeForce RTX 2060. Przy założeniu, że czas ten będzie rosnąć proporcjonalnie do rozmiaru danych treningowych, jedna epoka zajmie około 15 godzin, a całkowity trening, za który przyjąłem 25 epok, około dwóch tygodni. Drugim jest to, że pracę naukowe, z którymi chcę porównać swoje wyniki również używają zbiorów o podobnych rozmiarach.

Do realizacji projektu zostały wykorzystane następujące narzędzia:

- Python 3.7.0,
- biblioteka TensorFlow 2.4.0,
- edytor SublimeText 3

3. Użyte metody

W tym rozdziale opiszę użyte przez siebie metody prowadzące, od zbioru kodów źródłowych programów, do działającego modelu przewidującej kolejny token w programie.

3.1. Badane modele

Rekurencyjne sieci neuronowe należą do rodziny sieci służących do przetwarzania sekwencji danych o określonej długości. Jak pokazali w swojej publikacji autorzy [11], sieć LSTM osiąga znacznie lepsze wyniki od sieci rekurencyjnych Hopfielda [12] oraz minimalnie lepsze wyniki od sieci GRU, kosztem dłuższego czasu treningu. Z tego względu w swoich badaniach skupiam się głównie na warstwie LSTM oraz w mniejszej mierze na warstwie GRU, która również wyprzedza sieć Hopfielda pod względem skuteczności.

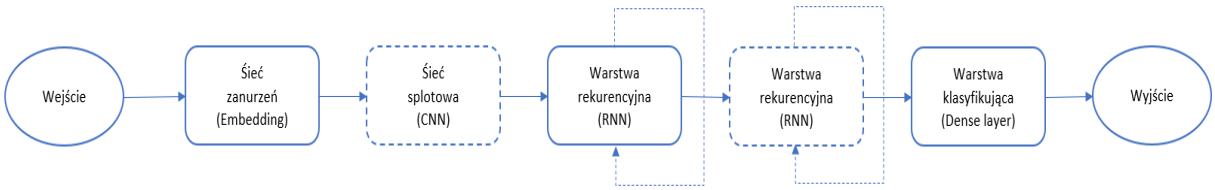
Wszystkie eksperymenty badające wpływ hiperparametrów przeprowadzę na sieci LSTM, oraz najlepszą ich kombinację zbadam przy wykorzystaniu warstwy GRU w celu porównania ich skuteczności w zadaniu przewidywaniu kodu. Na koniec zbadam również zachowanie wyznaczonego modelu dla różnych rozmiarów słownika.

Jak pokazał w swojej pracy Yoon Kim [13] połączenie warstwy rekurencyjnej z warstwą zanurzeń poprzez warstwę splotową może mieć pozytywny wpływ na skuteczność modelu. Jednak badanie to wykonywał jedynie w zadaniu modelowania języka naturalnego oraz dla modeli opartych na pojedyńczych znakach. Jednym z wykonanych przeze mnie eksperymentów będzie zastosowanie tej techniki w moim zadaniu, oraz oceny skuteczności tego podejścia dla modeli opartych na tokenach.

Ogólny badany przeze mnie model uczenia głębokiego został przedstawiony na rysunku 3.1. Składa się on z sieci zanurzeń, opcjonalnej warstwy CNN, 1 lub 2 warstw rekurencyjnych LSTM, oraz warstwy klasyfikującej.

Na architekturę całej wtyczki składa się wybrany przeze mnie model uczenia głębokiego połączony z 2 modelami typu n-gram. Zasada działania tej kombinacji została opisana w rozdziale 3.5.

Rysunek 3.1. Ogólny badany model. Opcjonalne warstwy zaznaczone linią przerywaną



Zdecydowałem, że nie ma potrzeby testowania wszystkich możliwych kombinacji parametrów w tabeli, ponieważ zajęło by to bardzo dużo czasu, oraz już na podstawie

Długość sekwencji	Warstwa CNN	Liczba warstw LSTM	Liczba neuronów w warstwie	Liczba wszystkich wag
1	nie	1	512	12,016,705
5	nie	1	128	3,302,593
			256	6,076,225
			512	12,016,705
		2	128	3,434,177
	tak	1	128	3,303,649
10	nie	1	128	3,302,593
			512	12,016,705
		2	128	3,434,177
		2	512	14,115,905
	tak	1	128	3,303,649
15	nie	1	128	3,302,593
			512	12,016,705

Tabela 3.1. Zestawienie wykonywanych eksperymentów

prac [3], [10] możemy łatwo stwierdzić, że część kombinacji nie osiągnie konkurencyjnych wyników przy pozostałych, na przykład pojedyńcza warstwa LSTM, o 128 komórkach i długości okna równej 1 jest zdecydowanie za prostym modelem aby przechwycić wszystkie zależności między danymi. Przeprowadzane przeze mnie eksperymenty przedstawione są w tabeli 3.1.

3.2. Hiperparametry

Badając różne w możliwości układów warstw należy rozważyć kilka szczególnych kwestii dotyczących hiperparametrów testowanych przeze mnie modeli. Poszczególnie zbadam

- dla jakiej długości sekwencji wejściowej model osiągnie najlepsze wyniki,
- jak rozmiar słownika wpływa na skuteczność modelu,
- jaka liczba warstw rekurencyjnych osiągnie najlepsze wyniki,
- dla jakiej liczby neuronów w warstwie model osiągnie najlepsze wyniki,
- jaki wpływ ma warstwa CNN,
- który rodzaj warstw rekurencyjnych osiągnie najlepsze wyniki.

3.3. Projektowanie wtyczki

Wstępne przetwarzanie danych

Przed dostarczeniem danych do modelu zostają one odpowiednio przetworzone. Operacja ta polega na usunięciu wszystkich wykomentowanych linii kodu w celu zapobiegnięcia uczenia modelu na słowach nie będących kodem. Następnie usuwane są wszystkie puste linie gdyż nie niosą one żadnej informacji o kolejnych tokenach. Ostatnim krokiem jest usunięcie wcięć w kodzie, gdyż wynikają one ze struktury programów w języku python, przez co próba przewidywania ich przy pomocy uczenia maszynowego nie ma sensu.

Modelowanie tokenów

Jak pokazują w swojej publikacji autorzy [14] mimo tego, że modele budujące kolejne słowa poprzez przewidywanie pojedyńczego znaku (character-level model) radzą sobie dobrze z modelowaniem języka naturalnego, oraz rozwiązują problem rozmiaru słownika, działają znacznie gorzej z językami programowania. Wniosek ten również potwierdza w swojej pracy autor [15] porównując model przewidujący znaki z modelem przewidującym tokeny. Z tego powodu realizuję tylko modele oparte na tokenach (token-level model).

Pierwszym krokiem jest zbudowanie słownika tokenów, które mogą pojawić się w kodzie. Buduję go poprzez przetworzenie wszystkich kodów źródłowych obu zbiorów treningowego oraz walidacyjnego modułem tokenize [16] wbudowanym w język Python. Moduł ten przyjmuje na wejściu kod źródłowy programu następnie zwraca listę kolejnych tokenów (nazw zmiennych, znaków specjalnych, słów kluczowych). Upewnia się on również czy kod jest poprawnie napisany, na przykład czy wszystkie nawiasy lub apostrofy zostały zamknięte. Pliki zawierające błędy w kodzie zostają pominięte. Znaki nowej lini również traktuję jako token, jednak nie uwzględniam wcięć w kodzie ze względu na to, że większość środowisk programistycznych stawia je automatycznie. Na przykład z kodu źródłowego:

Listing 1. Przykładowy program Python

```
1    for x in range(2, 10):
2        print("hello␣world")
```

otrzymamy listę **[for, x, in, range, (, 2, 10,), :, \n, print, (, "hello world",)]**. Następnie sortuje wszystkie wygenerowane tokeny według częstości występowania oraz wybieram top-n tokenów jako słownik i każdemu z nich przypisuję unikalną liczbę naturalną. Utworzony w ten sposób słownik nie jest kompletny ponieważ nie obejmuje on wszystkich możliwych nazw występujących w kodzie. Takiego rodzaju tokeny zostają zastąpione sztucznym tokenem '<UNKNOWN>'. W głównej mierze są to unikalne nazwy zmiennych oraz ciągi znaków. Zastępowanie tokenu '<UNKNOWN>' prawdziwym tokenem omawiam w rozdziale 3.5

Wybór podzbioru danych

Jak już wspomniałem w sekcji 2.7 dotyczącej zbioru danych, trening odbywa sie na podzbiorze wszystkich zgromadzonych danych. W tym celu wybrałem 9 najpopularniejszych, zewnętrznych bibliotek języka Python, stosowanych w zróżnicowanych dziedzinach rozwoju oprogramowania. Poniżej zamieszczam listę wybranych bibliotek, wraz z krótkim opisem:

- Django - rozwój aplikacji sieciowych,
- Numpy - wykonywanie obliczeń matematycznych wysokiego poziomu,
- Requests - wysyłanie zapytań http,

- Flask - rozwój aplikacji sieciowych,
- TensorFlow - Głębokie uczenie maszynowe,
- Keras - Wysokopoziomowe uczenie maszynowe,
- PyTorch - Głębokie uczenie maszynowe,
- Pandas - Zarządzanie dużymi zbiorami danych,
- PyQt - Tworzenie interfejsów użytkownika.

Aby upewnić się, że wybrany przeze mnie zbiór danych poprawnie oddaje rzeczywistość porównałem liczbę plików źródłowych zwierających konkretną bibliotekę ze zbioru, z odpowiadającą liczbą wszystkich plików źródłowych na platformie GitHub [4]. Stosunek tych dwóch liczb został przedstawiony w kolumnie 'Stosunek'. Zestawienie to znajduje się w tabeli 3.2

Tabela 3.2. Zestawienie zbioru danych z platformą GitHub

Biblioteka	Liczba plików GitHub	Liczba plików w zbiorze danych	Stosunek
Django	187000000	26732	0.014%
Numpy	55000000	9058	0.016%
Requests	42000000	6339	0.015%
Pandas	19000000	1328	0.007%
Flask	17000000	3230	0.019%
TensorFlow	10000000	96	0.001%
Keras	3000000	72	0.002%
PyQt	1000000	132	0.013%
PyTorch	1000000	0	0%

Jak możemy zaobserwować stosunek liczby plików jest dosyć zbliżony dla większości wybranych bibliotek wynosi około 0.015%. Wyjątkami są biblioteki uczenia maszynowego. Może to wynikać z tego, że dane pochodzą z 2018 roku. Jest to czas, w którym istniały początkowe wersje tych bibliotek oraz dopiero zaczynały zyskiwać na popularności. Od tego czasu również biblioteka Keras została scalona z biblioteką TensorFlow co znacznie wpłynęło na jej popularność w dzisiejszych czasach.
Uznaję, że dane w wystarczająco dobrym stopniu oddają częstotliwość zastosowania bibliotek. Końcowy podzbiór stworzyłem poprzez wybranie 1% plików źródłowych z każdej biblioteki. Końcowe zestawienie znajduje się w tabeli 3.3

Tabela 3.3. Zestawienie zbioru i podzbioru danych

	Cały zbiór danych	Podzbiór treningowy	Podzbiór walidacyjny
Pliki	150000	9103	4504
Tokeny	114641650	9118453	4482600

Rozmiar wykorzystanego zbioru danych różni się do zbioru użytego w publikacji [10] w którym użyto 16 milionów tokenów do treningu, oraz 5 milionów tokenów w celach walidacji. Na różnicę tą zdecydowałem się ze względu na ograniczenia sprzętowe

różniące oba projekty.

Trening

Zadanie polega na przewidzeniu kolejnego tokenu na podstawie zadanej sekwencji tokenów. Długość sekwencji jest stała oraz wyrażona poprzez wielkość okna będącą jednym z badanych hiperparametrów. Dla każdego z tokenów model wylicza jego wektor zanurzenia, wykonuje jeden krok w sieci rekurencyjnej po czym stosuje warstwę klasyfikującą (Dense layer) w celu wygenerowania logitów wyrażających logistyczne-prawdopodobieństwo kolejnego tokenu. Zatem dla zadanego okna tokenów długości W: $[t_1, t_2, ...t_W]$ obliczam wynik dla każdego możliwego wyjścia j, s_j jako funkcję z wektorów tokenów v_{t_i} z tokenów z okna.

$$g = [g_1, g_2, ..., g_W]$$
$$[g_1, g_2, ..., g_W] = RNN([v_{t_1}, v_{t_2}, ..., v_{t_W}])$$
$$s_j = p_j^T [g]$$

Minimalizowana funkcja strat jest entropią krzyżową pomiędzy prawdopodobieństwami $softmax$ dla każdego możliwego wyjścia a wykonaną predykcją.Funkcja wyrażoną wzorem:

$$L = -log(\frac{exp(s_{t_o})}{\sum_j exp(s_{t_j})})$$

gdzie t_o jest zaobserwowanym tokenem wyjściowym a g_i wyjściem $i-tej$ komórki którejś z badanych sieci rekurencyjnych.

Wagi sieci aktualizowane są po przetworzeniu porcji danych (mini-batch), której rozmiar jest stały. Przy treningach sieci rekurencyjnych rozmiar ten jest jedną z wartości kluczowych dla dobrej wydajności sieci. W moich eksperymentach wynosi on 128. Jest to kompromis pomiędzy rozsądnym czasem treningu oraz jakością wyjściowych sugestii. Jest to również najczęściej wybierana wartość w przytoczonych przeze mnie publikacjach.

Każda testowana architektura trenowana jest przez 25 epok. Wartość tą wybrałem na podstawie własnych eksperymentów wstępnych, z których wynika, że powyżej tej liczby model nie osiągał już lepszych rezultatów. Zbyt długi trening może również doprowadzić to przetrenowania modelu, czego należy unikać.

3.4. Ewaluacja

Jako, że badane przeze mnie modele tworzone są z myślą użycia ich w postaci wtyczki do środowiska programistycznego nie ma sensu ocenianie ich na podstawie pierwszej, najlepszej predykcji. Zamiast tego użyję dwóch następujących metryk:

Ewaluacja bez uwzględnienia kolejności

Jeśli poszukiwane słowo znajduje się w pierwszych n najlepszych predykcjach, bez

znaczenia na którym miejscu uważam sugestię za poprawną. Metrykę tą zastosowano przy ocenianiu systemów Pythia [3] dla której $n = 5$ oraz w pracy [2] gdzie $n = 3$. Zastosowanie jej pozwoli na porównanie wyników z tymi publikacjami. Miara ta jest dalej nazywana "Top n"

Ewaluacja z uwzględnieniem kolejności

Jednym z przytoczonym problemów we wstępie 1 jest to, że wiele wtyczek nie uwzględnia kolejności sugestii oraz proponuje je na przykład posortowane leksykograficznie. Proponowane przeze mnie rozwiązanie sortuje predykcje na podstawie prawdopodobieństwa ich wystąpienia. Należy uwzględnić tą kolejność w metryce. Ocena obliczana jest poprzez podzielenie prawdopodobieństwa poprawnej predykcji przez jego indeks w zbiorze zebranych predykcji. Dla przykładu powiedzmy, że model proponujący 10 sugestii zostaje użyty do wykonania 4 predykcji. Pierwsza wystąpi na 1. miejscu, druga na 3. miejscu, trzecia nie zmieści w w 10 najlepszych, a czwarta na 8. miejscu. W takim przypadku ocena modelu będzie wynosić $(\frac{1}{1} + \frac{1}{3} + 0 + \frac{1}{8})/4 = 0.36$. W ten sposób wyższe predykcje oceniane są zdecydowanie lepiej. Przy ocenie użyję 10 najlepszych sugestii. Metryki tej używa Erik van Scharrenburg [15] w swojej pracy. Zastosowanie jej pozwoli na porównanie wyników. Miara ta jest dalej nazywana "Top n z kolejnością"

3.5. Szczegóły implementacji

Problem słów poza słownikiem

Największym wyzwaniem przy modelowaniu języka programowanie jest rozwiązanie problemu słów poza słownikiem opisanego w podrozdziale 1.1. Jednym z podejść zastosowanym przez autorów [2] jest zastępowanie tokenów nie występujących w słowniku specjalnymi tokenami pozycyjnymi. Tego typu tokenowi, który powtarza się więcej niż raz w sekwencji, zostaje przypisany indeks jego wystąpienia odpowiadający jego pierwszemu wystąpieniu. W przypadku gdy przewidziany token nie mieści się w słowniku ale pojawił się wcześniej w sekwencji zostaje zastąpiony wcześniej podaną nazwą. Rozwiązanie to sprawdza się bardzo dobrze dla zmiennych o tej samej nazwie, znajdujących sie blisko siebie, na przykład w pętli for języka $C++$:

Listing 2. Przeparsowana pętla for

```
1    for (int POS_TOKEN_01=0; POS_TOKEN_01<10; POS_TOKEN_01++)
```

Metoda ta jednak ogranicza się do długości badanej sekwencji, która jest bardzo krótka względem przeciętnej długości kodów programów.

Inną metodą jest zastosowanie warstwy splotowej zamiast warstwy zanurzeń zaproponowaną przez Yoon Kim [13]. Metoda ta osiąga skuteczność na poziomie najlepszych dotychczas znanych modeli, jednak zastosowana jest dla modeli języka naturalnego

opartego na znakach, przez co prawdopodobnie nie sprawdziłaby się przy wybranych przeze mnie założeniach.

Hellendoorn wraz z Devanbu [10] proponują połączenie sieci rekurencyjnych z modelami statystycznymi $N-gram$. Metoda ta pozwala na bardziej ogólne predykcje. Pokazują również, że modele $N-gram$ radzą sobie lepie z przewidywaniem rzadko występujących unikalnych tokenów od sieci neuronowych, oraz kombinacja tych modeli osiąga mniejszą entropię niż każdy z tych modeli osobno.

Swoje eksperymenty przeprowadzam właśnie z zastosowaniem kombinacji sieci rekurencyjnej z modelami unigram oraz bigram, preferując podpowiedzi wykonane przez bigram. Zastosowanie większych modeli $N-gram$ nie ma sensu, ponieważ nałożyłoby to dodatkowe koszty obliczeniowe spowalniając wykonywanie predykcji, a generowany przez nie zbiór byłby w większości przypadków pusty. Stosuje tę metodę poprzez zastąpienie słów nie występujące w słowniku tokenem $<UNKNOWN>$ oraz uczę model uczenia głębokiego przewidywać go w odpowiednich miejscach. Następnie jeśli pośród zebranych predykcji znajdzie sie token $<UNKNOWN>$ zastępuje go zbiorem będącym sumą zbiorów predykcji unigramu oraz bigramu. Oba modele typu n-gram uczone są na bieżąco, na pisanym przez programistę tekście, zatem znają wszystkie użyte przez niego nazwy.

Rozmiar Słownika

W eksperymentach z tabeli 3.1 stosuję stałą wartość rozmiaru słownika wynoszącą 20000. Odpowiada ona usunięciu słów które nie pojawiają sie w zbiorze danych więcej niż 18 razy. Odcięte zostają w głównej mierze nazwy zdefiniowanych funkcji, zmiennych oraz ciągi znaków. Wybór ten spowodowany jest tym, że wartość ta ma ogromny wpływ na czas treningu oraz rozmiar modelu zapisanego na dysku. Wydłużony czas treningu spowodowany jest koniecznością obliczenia wartości funkcji $softmax$ dla każdego ze słów, natomiast dużo większy rozmiar wynika z konieczności przeskalowania warstwy wejściowej oraz wyjściowej. Rozmiar ten różni się od rozmiarów wybranych w przytoczonych przeze mnie pracach które wynoszą odpowiednio 74064 [10] oraz 2000 [2]. W celu odpowiedzi na pytanie czy rozmiar słownika ma znaczenie na badanie modelu przeprowadzę jeden eksperyment polegający na treningu modelu o najlepszej skuteczności na rozmiarze słownika 74064 zaproponowanego przez Hellendoorn'a [10]. Przykłady odciętych tokenów w słowniku rozmiaru 20000:
$accept\_inplace, IVGMM, book\_names, allvars, parent\_cards, 'references'$.

Wektory zanurzenia

Wektory zanurzeń odpowiedzialne są za przypisanie każdemu ze słów wektora, którym możemy wyrazić prawdopodobieństwo tokenu jako rezultat wielowarstwowej sieci neuronowej na wektorach o ograniczonej liczbie tokenów sąsiednich. W mojej pracy realizowane są one poprzez warstwę $Embedding$ należącą do biblioteki TensorFlow,

uczącą się równolegle z całym modelem. Wymiary wektorów zanurzeń są stałe dla każdego z wykonywanych eksperymentów oraz wynoszą one 32. Zdecydowałem się na mniejszy wymiar wektorów niż przedstawiony w pracy [10], która wynosiła 128 z powodu, że używam mniejszego słownika przez co znajduję się w nim dużo mniej zależności, które można wyrazić mniejszym wymiarem.

Optymalizator

W treningu używam optymalizatora $Adam$ o domyślnych parametrach dla każdego z modeli. Wybór ten wynikał z tego, że celem eksperymentów było badanie różnic wynikających z użycia odmiennych architektur. Odpowiednie strojenie optymalizatora typu SGD było by bardzo czasochłonne oraz komplikowałoby porównywanie ze sobą testowanych modeli. Jest to również optymalizator używany w pracach z którymi porównam uzyskane przeze mnie wyniki.

4. Analiza przeprowadzonych eksperymentów

4.1. Architektury

Zbadane architektury

Długość sekwencji	Warstwa CNN	Liczba warstw LSTM	Liczba neuronów w warstwie	Top 1	Top 5	Top 10 z kolejnością
1	nie	1	512	0.27	0.51	0.41
5	nie	1	128	0.23	0.59	0.37
			256	0.16	0.42	0.27
			512	0.30	0.63	0.43
		2	128	0.27	0.62	0.41
	tak	1	128	0.24	0.53	0.36
10	nie	1	128	0.06	0.25	0.13
			512	0.13	0.45	0.26
		2	128	0.34	0.66	0.47
		2	512	0.12	0.44	0.24
	tak	1	128	0.17	0.56	0.32
15	nie	1	128	0.03	0.10	0.06
			512	0.10	0.38	0.21

Tabela 4.1. Uzyskane skuteczności w eksperymentach

Wyróżniona architektura

Ze wstępnych badań najlepsze wyniki uzyskał model o parametrach przedstawionych w tabeli 4.2.

parametr	wartość
liczba warstw rnn	2
rodzaj rnn	lstm
liczba neuronów w warstwie	128
długość sekwencji wejściowej	10
liczba trenowanych parametrów	3,434,177
rozmiar słownika	20000
rozmiar wektora zanurzeń	32
rozmiar porcji danych	128
liczba epok	25
optymalizator	adam
funkcja celu	entropia krzyżowa

Tabela 4.2. Parametry najlepszego modelu z badań wstępnych

Uzyskując skuteczności przedstawione w tabeli 4.3.

Miara	Wynik
Top 1	0.34
Top 3	0.56
Top 5	0.66
Top 5 z kolejnością	0.46
Top 10 z kolejnością	0.47
najmniejsza wartość funkcji celu	1.67

Tabela 4.3. Wyniki uzyskane przez najlepszy model z badań wstępnych

Model ten osiągnął znacznie lepsze wyniki niż pozostałe badane modele. Na tym modelu zostaną wykonane dalsze badania dotyczące większego rozmiaru słownika oraz zastosowania warstwy gru zamiast lstm.

4.2. Różne rozmiary słownika

Kolejnym etapem badań jest sprawdzenie jak zachowa się model przy większych rozmiarach słownika. Przy badańach wstępnych wynosiła ona 20000. W tym eksperymencie użyje rozmiaru słownika takiego samego jak użyli w swojej pracy autorzy [10] wynoszącego 74000. Dla takiego rozmiaru pomijane są słowa występujące rzadziej niż 5 razy. wszystkie pozostałe parametry pozostają takie same.

Miara	Wynik
Top 1	0.30
Top 3	0.52
Top 5	0.61
Top 5 z kolejnością	0.42
Top 10 z kolejnością	0.43
najmniejsza wartość funkcji celu	1.75

Tabela 4.4. Wyniki uzyskane przez model o dużym słowniku

4.3. Zastosowanie warstwy gru

Ostatnim wykonanym przeze mnie eksperymentem jest porównanie skuteczności warstwy gru z warstwą lstm. Badanie to przeprowadzam dla parametrów wyróżnionego modelu 4.1 podmieniając rodzaj warstwy rnn. Uzyskane wyniki zostały przedstawione w tabeli 4.5.

Miara	Wynik
Top 1	0.32
Top 3	0.59
Top 5	0.68
Top 5 z kolejnością	0.46
Top 10 z kolejnością	0.47
najmniejsza wartość funkcji celu	1.57

Tabela 4.5. Wyniki uzyskane przez model o warstwie GRU

4.4. przegląd hiperparametrów

Wpływ długości sekwencji wejściowe

Model został wytrenowany na czterech długościach sekwencji wejściowej: $1, 5, 10, 15$. Można zaobserwować wzrost skuteczności modelu wraz ze wzrostem sekwencji, do 10 słów. dalsze próby powiększania sekwencji wpływają negatywnie na jego działanie. spowodowane jest to tym, że dla krótkich sekwencji takich jak $1, 5$ model posiada za mało kontekstu aby wykonać predykcję, natomiast sekwencje długości 15 są już zbyt specyficznymi częściami konkretnego programu aby móc je uogólnić dla przewidywanego kodu.

Wpływ długości rozmiaru słownika

Wielkość słownika ma duży wpływ na działanie modelu. Przy małych wielkościach można spodziewać się niepoprawnego działania ze względu na to, że sieć nie będzie znała słów, które ma przewidywać. jak pokazują wyniki eksperymentu w tabeli 4.4, słownik o bardzo dużej liczbie słów również pogarsza działanie algorytmu. Różnica ta wynosi około 5 punktów procentowych we wszystkich kategoriach. Duże rozmiary słownika również znaczenie spowalniają proces treningu.

Liczba warstw sieci rekurencyjnej

Eksperymenty zostały przeprowadzone dla jednej oraz dwóch warstw sieci rekurencyjnej. We wszystkich przeprowadzonych eksperymentach, modele o dwóch warstwach osiągnęły podobne lub lepsze wyniki od modeli o jednej warstwie.

Wpływ liczby neruronów w warstwie rekurencyjnej

Eksperymenty zostały przeprowadzone dla warstw składających cie z warstw sieci rekurencyjnej. Można zaobserwować, że duże liczby neuronów w warstwie nie poprawiają działania modelu. Najlepszy rezultat oraz inne rezultaty zbliżone do niego uzyskują warstwy składające się ze 128 neuronów.

Wpływ warstwy CNN

Przeprowadziłem dwa eksperymenty z zastosowaniem warstwy CNN, oba dla jednej

warstwy LSTM o 128 neuronach różniących się długością sekwencji wejściowe. Po porównaniu tych modeli z modelami o tych samych parametrach bez warstwy CNN, zauważalna jest nieznaczna poprawa, około 1 punkt procentowy dla modelu o długości sekwencji równej 5, oraz około 10 punktów procentowych dla modelu o długości wejściowej równej 10. Mimo poprawy modele te wciąż osiągają znacznie gorsze wyniki od pozostałych badanych modeli.

Zastosowanie warstwy GRU

Zastosowanie warstwy GRU poprawiło działanie wyróżnionego modelu w kategoriach Top 3 oraz Top 5, natomiast pogorszyło w Top 1. Różnice te są jednak bardzo niewielkie, wynoszą około 2 punktów procentowych i najprawdopodobniej wynikają z losowości procesu uczenia. Można wywnioskować zatem, że w tak przedstawionym zadaniu rodzaj zastosowanej warstwy nie ma wpływu na skuteczność modelu. Warto zatem użyć warstwy GRU w celu skrócenia czasu treningu.

4.5. Najlepsza znaleziona architektura

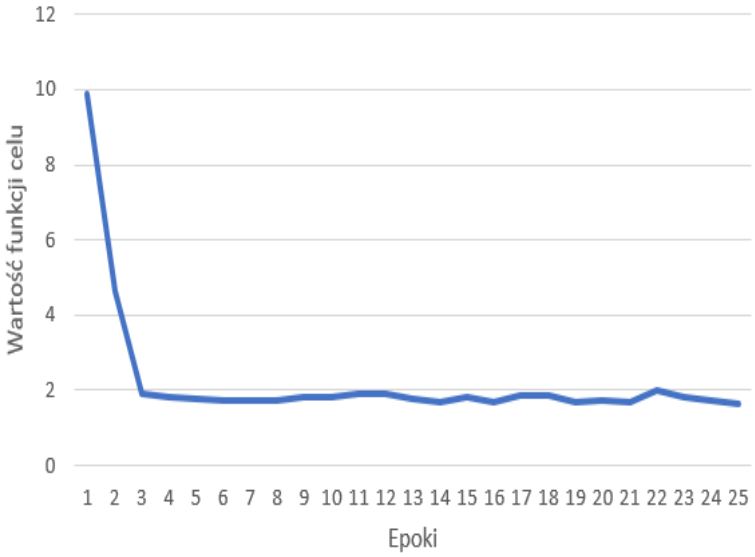
Jako, że model trenowany jest z myślą o stworzeniu wtyczki ułatwiającej pisanie kodu, do wybrania najlepszej architektury posłużę się metryką uwzględniającą 5 najlepszych podpowiedzi z uwzględnieniem kolejności wystąpienia. W tym kryterium oba modele scharakteryzowane w sekcjach 4.5, 4.3 uzyskują taką samą skuteczność wynoszącą 46%, jednak dla 5 najlepszych parametrów bez uwzględnia kolejności radzi sobie lepiej model przedstawiony w 4.5 osiągając skuteczność równą 68%, zatem to on zostanie użyty w implementacji wtyczki oraz porównania z architekturami z innych prac. W tabeli 4.6 przedstawiam dokładne parametry tego modelu.

parametr	wartość
liczba warstw rnn	2
rodzaj rnn	GRU
liczba neuronów w warstwie	128
długość sekwencji wejściowej	10
liczba trenowanych parametrów	3,434,177
rozmiar słownika	20000
rozmiar wektora zanurzeń	32
rozmiar porcji danych	128
liczba epok	25
optymalizator	adam
funkcja celu	entropia krzyżowa

Tabela 4.6. Wybrany najlepszy model

Na wykresie 4.1 została przedstawiona wartość funkcji celu w kolejnych epokach treningu.

Rysunek 4.1. Wartość funkcji celu w kolejnych epokach



4.6. Generowane wyniki

Rysunek 4.2. Przykładowy wynik wygenerowany przez model

json.py

```
from django . utils import simplejson
from django . forms . util import ErrorDict , ErrorList
class ErrorJSONEncoder ( simplejson . JSONEncoder ) :
def default ( self , o ) :
if issubclass ( o . __class__ , ErrorDict ) :
return dict ( o )
if issubclass ( o . __class__ , ErrorList ) :
return list ( o )
else :
return super ( ErrorJSONEncoder , self ) . default ( o )
def dumps ( o ) :
return simplejson . dumps ( o , cls = ErrorJSONEncoder , default = str )
```

74/110 = 0.6727272727272727

Na rysunku 4.2 został przedstawiony przykładowy program którego uzupełnienie realizował model. Na zielono zostały zaznaczone tokeny, które znalazły się w 5 pierwszych predykcjach modelu, pozostałe tokeny oznaczone zostały kolorem czerwonym. Można zaobserwować, że model w bardzo dobrym stopniu radzi sobie z przewidywaniem struktury programu, uzupełnianiem nawiasów, wykrywaniem końca linii, stawianiem słów kluczowych. Gorzej radzi sobie z uzupełnianiem nazw pochodzących z bibliotek oraz definiowanych zmiennych.

4.7. Porównanie uzyskanych wyników

Modele pochodzące z przytoczonych prac pełnią to samo zadanie, jednak trenowane są na innych zbiorach danych. Mimo to, uzyskany przeze mnie model osiąga bardzo zbliżone do nich wyniki, oraz warto zestawić je ze sobą.

Autor [15] uzyskał model o skuteczności wynoszącej 69.7%, dla 5 najlepszych wyników. Jest to niemal identyczny wynik z modelem uzyskanym przeze mnie. Mimo to architektury modeli znacznie różnią się, gdyż w pracy zastosował jedną warstwę LSTM o 512 neuronach. Różnica ta może wynikać z narzucenia innych hiperparametrów.

Autor [2] uzyskał model o skuteczności wynoszącej 66.3% dla 3 najlepszych predykcji, bez uwzględniania kolejności ich wystąpienia. Skuteczność mojego modelu w tej kategorii jest mniejsza, gdyż wynosi ona 0.59%. Różnica ta najprawdopodobniej wynika z tego, że model przedstawiony w pracy [2] jest trenowany oraz testowany na pojedyńczych bibliotekach, zatem realizuje on znacznie prostsze zadanie. Modele te również wykorzystują całkowicie inną architekturę. Wykorzystany zostaje model ze skupieniem uwagi składający się z 3 nieliniowych warstw.

W pracy autorzy [10] uzyskują model o skuteczności wynoszącej 67.9%, dla 10 najlepszych predykcji z uwzględnieniem kolejności ich wystąpienia. Mój model osiąga w tej metryce gorszy wynik, równy 47%. Różnice te mogą wynikać z znacznie dłuższego czasu treningu wynoszącego około 3 dni, oraz z bardziej złożonego mechanizmu predykcji, uwzględniającego zagnieżdżenia w kodzie.

4.8. Możliwe błędy

Wszystkie oceniane przeze mnie modele trenowane były tylko raz. Ze względu na długi czas wykonania pojedyńczego eksperymentu, zdecydowałem się na porównanie ze sobą wiele różnych modeli zamiast kilku modeli kilkakrotnie trenowanych. Efektem tego mogą być nieprecyzyjne wyniki wynikające z losowości procesu uczenia, gdyż rozmiar próbki równy 1, jest zdecydowanie za mały aby dokładnie określić działanie modelu. Jako, że nie posiadamy jeszcze wystarczającej wiedzy na temat cech wspólnych pomiędzy językami programowania a językami naturalnymi, sugerowanie się modelami pochodzącymi z tej dziedziny nie przynosi odpowiednich rezultatów.

Wszystkie zastosowane miary traktują przewidziane tokeny równo. Nie oddają one zatem dokładnie w jakim stopniu wtyczka usprawnia pracę programisty, gdyż przewidzenie długiej nazwy oszczędza znacznie więcej czasu niż np. pojedyńczego przecinka.

Relacja pomiędzy kryteriami, którymi są oceniane modele, a faktyczną skutecznością tych modeli w praktyce jest nie znana. Zbyt duża liczba sugestii proponowanych przez model również negatywnie wpływa na pracę programisty, ponieważ sprawia, że pole zawierające sugestie staje się nieczytelne. Ocena z uwzględnieniem kolejności zakłada, że predykcja znajdująca się na drugim miejscu jest tylko w połowie tak użyteczna, jak ta znajdująca się na miejscu pierwszym, natomiast sugestia na trzecim miejscu tylko w jednej trzeciej itd. Założenie to wynika z tego, że wybranie niższej oceny wymaga więcej

pracy od programisty. Mimo to łatwo pominąć ten problem np. stosując skróty klawiszowe, zatem nie jesteśmy w stanie określić, czy to założenie jest poprawne.

5. Implementacja wtyczki

Zintegrowane środowisko programistyczne Sublime Text 3

Sublime Text 3 [17] jest wieloplatformowym, rozbudowanym i wysoce konfigurowal-

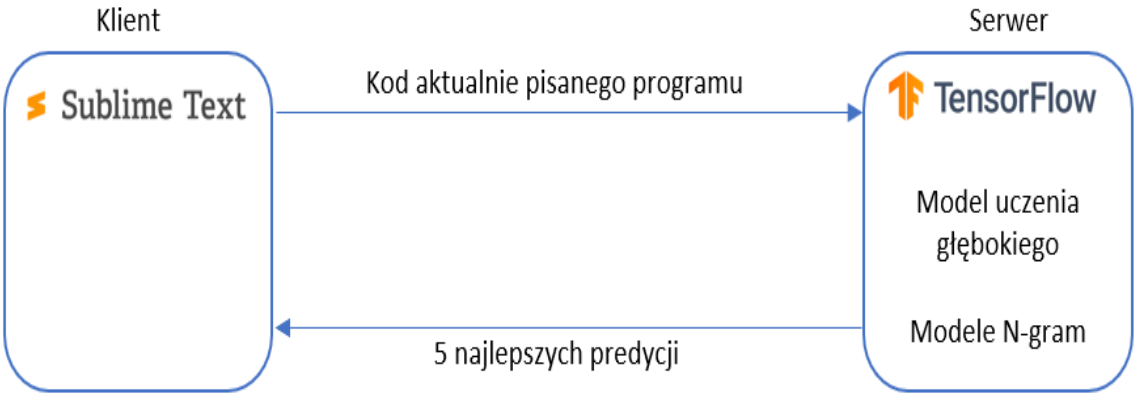
Rysunek 5.1. Sublime Text 3

nym edytorem teksty stworzonym z myślą o programistach. Udostępnia ono interfejs programistyczny w języku Python pozwalający na proste tworzenie własnych wtyczek lub instalacje wtyczek stworzonych przez społeczność użytkowników aplikacji, jak i modyfikowanie samego środowiska. Natywnie wspiera obsługę wszystkich najpopularniejszych języków programowania oraz języków znaczników (markup language) co czyni je bardzo uniwersalnym oraz zawdzięcza temu swoją dużą popularność. Na liście najczęściej wybieranych środowisk programistycznych [18] ocenianych pod kątem częstości odwiedziń strony pobierania, aktualnie zajmuje 9 miejsce. Wyprzedają je głównie środowiska tworzone pod kątem rozwoju w konkretnym języku programowania takie jak pyCharm lub Eclipse.
Sublime Text jest zaimplementowane w języku Python oraz C++.

Architektura wtyczki

W celu uniknięcia konieczności doinstalowywania przez użytkowników dodatkowych modułów do wirtualnego środowiska wykonawczego programu Sublime Text, zaimplementowałem wtyczkę w architekturze klient-serwer. Wtyczka przy uruchomieniu programu Sublime Text uruchamia zewnętrzny proces serwera przyjmującego zapytania pod lokalnym adresem komputera na porcie 8000. Serwer składa się najlepszego znalezionego przeze mnie modelu uczenia głębokiego oraz kombinacji unigramu i bigramu. Podejście to pozwala również na uniknięcie kombinacji związanych z niezgodnościami wersji kompilatora Python środowiska Sublime Text (Python 3.6) z wersjami użytych przeze mnie bibliotek

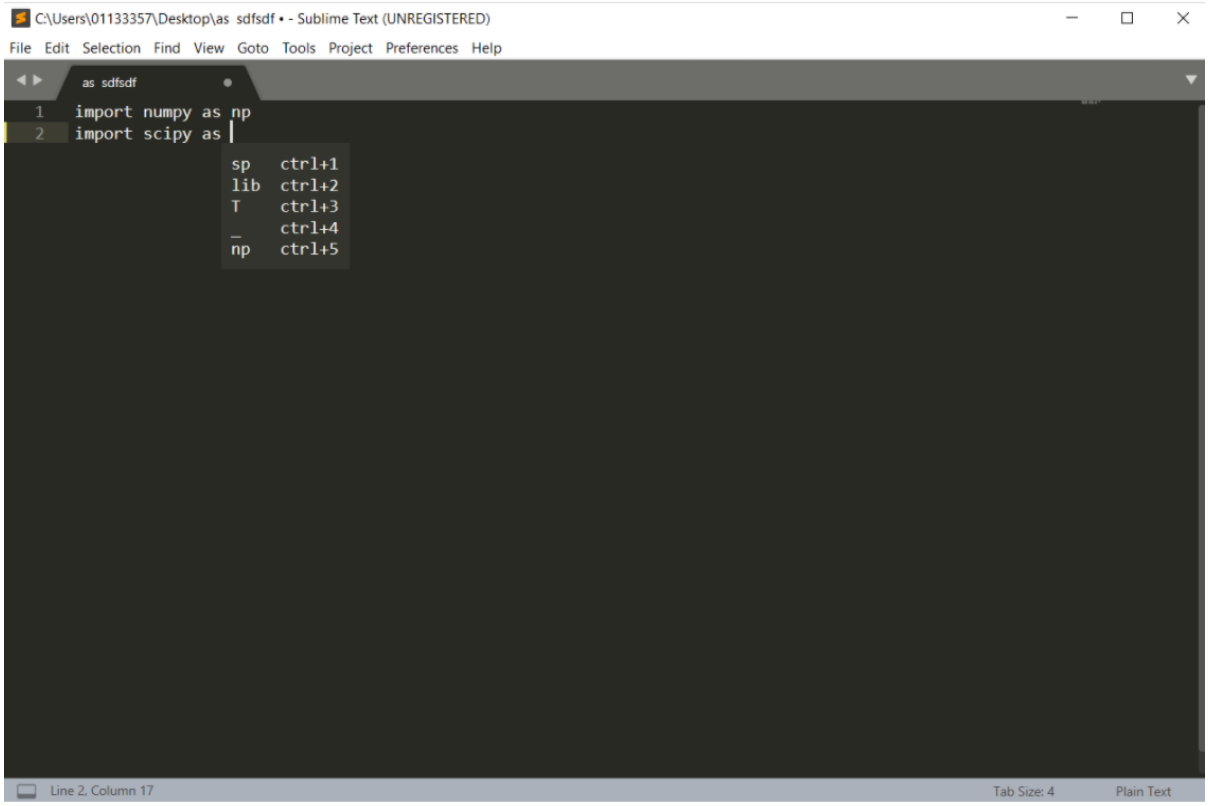
Rysunek 5.2. Architektura wtyczki



5.1. Obsługa

Wtyczka po każdym naciśnięciu spacji, enter lub któregoś ze znaku specjalnego wysyła zapytanie do serwera składające sie z całego kodu aktualnie modyfikowanego programu, a w odpowiedzi uzyskuje wiadomość składającą sie z 5 najlepszych predykcji wykonanych przez model. Następnie użytkownik może wybrać którąś z sugestii poprzez naciśnięcie kombinacji klawiszy $ctrl + i, i \in \{1,2,3,4,5\}$.

Rysunek 5.3. Działanie wtyczki



5.2. Czas odpowiedzi modelu

Po wykonaniu 1000 zapytań do serwera odpowiedzialnego za wykonywanie predykcji, średni czas odpowiedzi wyniósł 18.8ms. W skład tej operacji wchodzi wysłanie zapytania do serwera lokalnego, tokenizacja danych, wykonanie predykcji oraz odesłanie odpowiedzi. Czas ten jest na tyle krótki, że nie ma żdanego wpływu na efektywność pisania kodu.

6. Podsumowanie

Omówienie

W swojej pracy przedstawiam zastosowanie rekurencyjnych sieci neuronowych do autouzupełniania kodu źródłowego. W swoich eksperymentach uzyskałem model osiągający skuteczność równą 68% dla 5 pierwszych predykcji oraz 46% po uwzględnieniu kolejności ich wystąpienia. Badam również wpływ hiperparametrów takich jak długość sekwencji wejściowej, rodzaje warstw sieci rekurencyjnych, układ warstw sieci, rozmiar słownika i ilość neuronów w warstwie. Pokazuję, że w zadaniu przewidywania kodu lepiej działają modele, w których skład wchodzi kilka warstw o małej ilości neuronów oraz długości sekwencji wejściowych liczące 10 słów.

Napotkane problemy

Głównym wyzwaniem oraz detalem różniącym języki programowania od języków naturalnych, jest możliwość nadawania dowolnych nazw obiektom oraz metodom, przez co nie można objąć wszystkich słów w słowniku danych treningowych. Słowa tego typu nazywane są słowami poza słownikiem. Zwiększanie wielkości słownika nigdy nie obejmie wszystkich możliwych nazw, natomiast bardzo spowolni ostatni krok algorytmu, którym jest obliczenie wyznaczenie funkcji softmax. Jak zostało pokazane w publikacji [10] od pewnego momentu większy rozmiar słownika zaczyna wpływać negatywnie na skuteczność modelu.

Nadmierne dopasowanie modelu do danych treningowych może wystąpić przy zbyt długim treningu. Taki model zacznie dawać bardzo dobre predykcje na zbiorze treningowym jednak bardzo słabo poradzi sobie na zbiorze walidacyjnym. Zamiast zgeneralizować problem model nauczy się danych treningowych 'na pamięć'.

Ostatnim napotkanym przeze mnie problemem były ograniczenia wynikające ze sprzętu. Aby trening sieci skończył się w sensownym czasie musi się on odbywać na karcie graficznej. Nie ma serwisu udostępniającego moc obliczeniową tych kart za darmo na wystarczająco długi czas.

Dalsze prace

Poprawa modelu

Dalsze badania dotyczące hiperparametrów np. długości sekwencji wejściowych z przedziału (5, 10) i (10, 15) jak i wielokrotne treningi modelu mogą znacznie wpłynąć na poprawę uzyskanych w tej pracy wyników. Ze względu na ograniczenia czasowe, modele testowane w tej pracy były trenowane przez względnie krótki czas. Wydłużenie czasu treningu również powinno pozytywnie wpłynąć na model

końcowy.

Inne języki programowania

Ze względu na bardzo duże podobieństwa między językami programowania, warto zbadać zachowanie przedstawionych modeli na językach innych niż Python. Zastosowane w tej pracy rozwiązania bardzo łatwo adaptują się do eksperymentów z innymi danymi wejściowymi. Możemy oczekiwać innych rezultatów dla języków statycznych takich jak Java lub języków deklaratywnych np. SQL.

Rozszerzenie dla innych środowisk programowania

Wtyczka w tej pracy została zaimplementowana tylko dla środowiska SublimeText 3, jednak ze względu na jej prostą oraz rozszerzalną budowę bardzo łatwo wprowadzić ją do innych środowisk programistycznych. Dzięki architekturze serwerowej może obsługiwać wiele klientów jednocześnie, oraz po przeniesieniu do chmury nie wymagałaby pobierania sieci neuronowej i bibliotek uczenia maszynowego na komputer użytkownika.

Bibliografia

[1] *ml4code*, Dostęp zdalny: https://ml4code.github.io/papers.html.

[2] C. S. Subhasis Das, "Contextual Code Completion Using Machine Learning", prac. mag., Stanford, 2015.

[3] A. Svyatkovskiy, S. Fu, Y. Zhao i N. Sundaresan, "Pythia: AI-assisted Code Completion System", Microsoft, 2019.

[4] *github*, Dostęp zdalny: https://github.com.

[5] Dostęp zdalny: https://tabnine.com.

[6] Dostęp zdalny: https://openai.com.

[7] S. Hochreiter i J. Schmidhuber, "Long Short-term Memory", *Neural computation*, t. 9, s. 1735–80, grud. 1997. DOI: 10.1162/neco.1997.9.8.1735.

[8] K. Cho, B. van Merrienboer, Ç. Gülçehre, F. Bougares, H. Schwenk i Y. Bengio, "Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation", *CoRR*, t. abs/1406.1078, 2014. arXiv: 1406.1078. adr.: http://arxiv.org/abs/1406.1078.

[9] srilab, *150k Python Dataset*, Dostęp zdalny: https://www.sri.inf.ethz.ch/py150.

[10] V. J. Hellendoorn i P. Devanbu, "Are Deep Neural Networks the Best Choice for Modeling Source Code?", w *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017. Paderborn, Germany: ACM, 2017, pp. 763–773, 2017.

[11] A. Shewalkar, D. Nyavanandi i S. A. Ludwig, "Performance Evaluation of Deep Neural Networks Applied to Speech Recognition: RNN, LSTM and GRU", *Journal of Artificial Intelligence and Soft Computing Research*, t. 9, nr. 4, s. 235–245, 1Oct. 2019. DOI: https://doi.org/10.2478/jaiscr-2019-0006. adr.: https://content.sciendo.com/view/journals/jaiscr/9/4/article-p235.xml.

[12] J. J. Hopfield, "Neural Networks and Physical Systems with Emergent Collective Computational Abilities", *Proceedings of the National Academy of Sciences of the United States of America*, t. 79, nr. 8, s. 2554–2558, 1982, ISSN: 00278424. adr.: http://www.jstor.org/stable/12175.

[13] Y. Kim, Y. Jernite, D. Sontag i A. M. Rush, *Character-Aware Neural Language Models*, 2015. arXiv: 1508.06615 [cs.CL].

[14] H. Peng, L. Mou, G. Li, Y. Liu, L. Zhang i Z. Jin, "Building Program Vector Representations for Deep Learning", w *Knowledge Science, Engineering and Management*, S. Zhang, M. Wirsing i Z. Zhang, red., Cham: Springer International Publishing, 2015, s. 547–553, ISBN: 978-3-319-25159-2.

[15] E. van Scharrenburg, "Code Completion wiht Recurrent Nerual Networks", prac. mag., University of Amsterdam, 2018.

[16] *tokenize*, Dostęp zdalny: https://docs.python.org/3/library/tokenize.html.

[17] *Sublime Text 3*, Dostęp zdalny: https://www.sublimetext.com/.

[18] *pypl*, Dostęp zdalny: `https://pypl.github.io/IDE.html`.

Wykaz symboli i skrótów

EiTI – Wydział Elektroniki i Technik Informacyjnych
PW – Politechnika Warszawska

Spis rysunków

Spis tabel

Spis załączników