Raport

Narzędzie do generacji tytułów **publikacji naukowych**

Autorzy: Katarzyna Poręba, Aleksandra Rąpała, Michał Rudy

1. Wstęp

Zagadnienie, które rozwiązywaliśmy to: stworzenie narzędzia do generacji tytułów publikacji naukowych.

W źródłach znajdował się odnośnik do projektu **csinva/gpt-paper-title-generator** znajdującego się na portalu github.

Projekt ten rozwiązywał trzy główne problemy:

- Generowanie tytułów specyficznych dla autora, korzystając z bazy danych arXiv,
- Dopracowanie tytułów prac,
- Ocenienie, czy tytuł pasuje do pracy

Po analizie oraz testowaniu istniejącego produktu bardzo brakowało nam możliwości wygenerowania tytułu na bazie słów kluczowych. Funkcjonalność ta byłaby wyjątkowo atrakcyjna dla użytkownika końcowego.

Zdecydowaliśmy więc, że właśnie tą funkcjonalność spróbujemy zaimplementować. Projekt gpt-paper-title-generator korzysta z **GPT-3**, czyli produktu firmy OpenAI. Ten model języka wykorzystuje deep learning w celu tworzenia tekstu podobnego do ludzkiego. GPT-3 jest dostępny dla każdego. Działa on jednak na bazie API OpenAI, co może być dosyć kosztowne. Postanowiliśmy więc użyć sieci neuronowych na bazie biblioteki **keras**.

2. Rozwinięcie, przebieg pracy

2.1. Przygotowanie danych

Do wygenerowania tytułów naukowych korzystamy z bazy danych **arXis**. Jest to repozytorium udostępniające 2,179,363 artykułów naukowych. Gromadzi on artykuły z takich dziedzin jak: astronomia, matematyka, statystyka, informatyka, biologia [1].

Do pozbycia się z niechcianych znaków używamy funkcji clean_title. Bazuje ona na kodzie z wcześniej wspomnianego projektu gpt-paper-title-generator.

```
import re
import tensorflow as tf
import pandas as pd
import numpy as np
import os
import time

def clean_title(s):
    s = s.replace('\n', '')
    s = s.replace('\t', '')
    s = re.sub(' +', ' ', s)
    return s

def get_metadata():
    # check if file with clean data exists
    if os.path.exists('./data/arxiv_metatadata_2022_clean.pkl'):
        df = pd.read_pickle('./data/arxiv_metatadata_2022_clean.pkl')

    # prepare this file if it doesn't exist
    else:
        df = pd.read_pickle('./data/arxiv_metadata_2022.pkl')
        df = df.drop(columns=['versions', 'update_date'])

        # remove replicated submissions and clean titles
        df = df.drop_duplicates(subset=['title', 'authors'],
        keep='first')
        df['title'] = df['title'].apply(clean_title)

        # calculate title length
        df['title_len'] = df['title'].str.split(' ').apply(len)
        df = df[~df['title'].str.lower().str.startswith('comment')]
```

```
        df = df.to_pickle('./data/arxiv_metatadata_2022_clean.pkl')
    return df
```

Opis funkcji def clean_title(s):

- W przekazanym do funkcji ciągu s, za pomocą metody **replace()** stary ciąg zostaje zastąpiony nowym, a mianowicie każdy znak nowej linii, a także znak tabulacji zostanie usunięty.
- Kolejną ważną metodą jest **sub()**, która pozwala na zmianę wzorca na inny ciąg znaków. Znak "+" wskazuje, że poprzedni znak może wystąpić raz, lub więcej razy.

Opis funkcji def get_metadata():

- Na samym początku sprawdziliśmy za pomocą **os.path.exists()**, czy wybrana ścieżka do pliku z "clean data" istnieje.
- Następnie, metoda **read_pickle()** posłużyła nam do serializacji danego obiektu. Jako argument przekazaliśmy właściwą ścieżkę do pliku, który należało załadować.
- W przypadku, gdy funkcja **os.path.exists()** zwróciła w programie wartość "false", przystąpiliśmy do przygotowania potrzebnego pliku ze wspomnianymi "clean data".
- Istotnym elementem okazało się użycie **df.drop()**, dzięki czemu dokonaliśmy usunięcia z kolumn określonych etykiet 'versions' oraz 'update_date'.
- W następnym kroku należało pozbyć się ewentualnie zduplikowanych wierszy, do czego szczególnie przydatnym narzędziem okazała się metoda **drop_duplicates().**
- W ostatnim kroku zrealizowaliśmy obliczanie długości odpowiednich tytułów. W tym miejscu szczególną rolę odegrała metoda **str.split(),** która dzieli łańcuch znaków według podanego seperatora (w naszym przypadku ogranicznikiem jest znak spacji). Z kolei funkcja **str.lower()** skonwertowała wszystkie litery w ciągu na małe znaki, dzięki czemu zapobiegliśmy przewidywalnych problemów związanych z porównywaniem tych samych liter. Istotną rolę w naszym kodzie odegrała również metoda **str.startswith()**, która zwraca wartość "true", jeśli podany przez nas łańcuch znakowy zaczyna się od ciągu - w naszym przypadku "comment".

2.2. Prezentacja przetworzonych danych

Dzięki prawidłowemu przygotowaniu funkcji get_metadata() w poprzednim kroku, mogliśmy zrealizować wyświetlenie z pliku tylko kolumny tytułowej.

```
data = get_metadata().title
```

Należy podkreślić, że baza danych **arXis** (z roku 2022) ma **158586**

wierszy, więc aby wykorzystać je do zastosowania sieci neuronowej, pobraliśmy tylko 1000 pierwszych wierszy.

```
# get first 500 rows
data = data.iloc[0:1000]
print(get_metadata())
```

2.3. Wektoryzacja tekstu

```
# creact a text from a list of strings
fullText = " ".join(data.tolist())

vocab = sorted(set(fullText))
print(vocab)
changeCharToIndex = {u:i for i, u in enumerate(vocab)}
changeIndexToChar = np.array(vocab)
changeTextToInt = np.array([changeCharToIndex[c] for c in fullText])
examples_per_epoch = len(fullText)//(101)

# convert the text vector into a stream of character indices
char_dataset = tf.data.Dataset.from_tensor_slices(changeTextToInt)
sequences = char_dataset.batch(101, drop_remainder=True)

# shuffle the data and pack it into batches
dataset = sequences.map(split_input_target)
BATCH_SIZE = 10
BUFFER_SIZE = 10000
dataset = dataset.shuffle(BUFFER_SIZE).batch(BATCH_SIZE,
drop_remainder=True)
```

W pierwszej kolejności pobrane przez nas tytuły są scalane, a następnie powstały tekst jest sortowany. W ten sposób otrzymujemy unikalne znaki.

Warto podkreślić, że naszym głównym zamiarem jest **wektoryzacja** tekstu, czyli konwersja ciągów znaków na listę wartości. W tym celu kolejny realizowany przez nas krok to konwersja elementów utworzonej tablicy na odpowiadające jej liczby.

Drugim etapem jest konwersja wektora tekstowego na strumień indeksów z wykorzystaniem metody ***from_tensor_slices()*** biblioteki TensorFlow.

Ostatnim etapem jest upakowanie danych. Metoda **batch()** pozwala nam przekształcić pojedyncze znaki w sekwencje o pożądanej wielkości.

2.4. Uczenie maszynowe

Nasz generator zamiast GPT-3 korzysta z sieci neuronowych. Dokładna nazwa użytej przez nas sieci to **RNN** (Recurrent neural network)**.** Sieci rekurencyjne posiadają połączenia do wcześniejszych warstw. Jej stan obecny zależy pośrednio od stanów z przeszłości [2]. Jej głównym zadaniem w naszym generatorze jest uzyskanie najbardziej prawdopodobnego ciągu na bazie określonego ciągu wejściowego.

W tym celu stworzona została (na bazie kodu TensorFlow) funkcja **build_model**. Model ten składa się z trzech warstw: **Embedding** – warstwy wejściowej, **GRU** – warstwy RNN, **Dense** – warstwy wyjściowej.

```
def build_model(vocab_size, embedding_dim, rnn_units, batch_size):
    model = tf.keras.Sequential([
        # The input layer. A trainable lookup table that will map each
character-ID to a vector with embedding_dim dimensions
        tf.keras.layers.Embedding(vocab_size, embedding_dim,
        batch_input_shape=[batch_size, None]),
        # A type of RNN with size units=rnn_units
        tf.keras.layers.GRU(rnn_units, return_sequences=True,
        stateful=True, recurrent_initializer='glorot_uniform'),
        # The output layer, with vocab_size outputs
        tf.keras.layers.Dense(vocab_size)
    ])
    return model

# Length of the vocabulary in StringLookup Layer
vocab_size = len(vocab)
# The embedding dimension
embedding_dim = 100
# Number of RNN units
rnn_units = 100
```

```
model = build_model(vocab_size=vocab_size, embedding_dim=embedding_dim,
rnn_units=rnn_units, batch_size=BATCH_SIZE)
# model.summary()
```

2.5. Trenowanie modelu

Następnie wykonywane jest trenowanie modelu. Na bazie poprzedniego stanu RNN oraz danych wejściowych możemy przewidywać klasę następnego znaku. Jako funkcję utraty stosujemy funkcję **sparse_categorical_crossentropy**.

```
# defining loss function
def loss(labels, logits):
    return tf.keras.losses.sparse_categorical_crossentropy(labels,
logits, from_logits=True)
```

Podczas trenowania zapisujemy również punkty kontrolne. Znajdują się one w folderze **training_checkpoints.**

```
# configure the training procedure
model.compile(optimizer='adam', loss=loss, run_eagerly=True)

# configure checkpoints during training
checkpoint_prefix = os.path.join('./training_checkpoints',
"ckpt_{epoch}")
checkpoint_callback =
tf.keras.callbacks.ModelCheckpoint(filepath=checkpoint_prefix,
save_weights_only=True)
```

Następnym krokiem jest przejście do trenowania modelu. W naszym przypadku wybraliśmy ilość **50** Epochs. Przy tej liczbie uzyskiwaliśmy interesujące wyniki oraz trenowanie nie zajmowało dużo czasu (ok. 18s na epoch). Podczas trenowania wartości każdej epoki malały. Następnie wybieramy najlepszy model spośród tych przez nas wygenerowanych.

2.6. Testowanie

Aby wygenerować tekst, musimy uruchomić podany, jako argument funkcji model w pętli oraz w ramach testowania śledzić jego stan w czasie działania programu.

```
def text_generator(model, start_string, temperature):
    num_generate = 40

    # vectorize starting string
    input_eval = [changeCharToIndex[s] for s in start_string]
    input_eval = tf.expand_dims(input_eval, 0)

    text_generated = []
    model.reset_states()
    iterTmp = 0
    while 1:
        iterTmp += 1
        # remove the batch dimension, apply temperature
        predictions = tf.squeeze(model(input_eval), 0) / temperature

        # predict the character using a categorical distribution
        predicted_id = tf.random.categorical(predictions,
num_samples=1)[-1,0].numpy()

        # change the predicted characters
        input_eval = tf.expand_dims([predicted_id], 0)
        text_generated.append(changeIndexToChar[predicted_id])
        if (iterTmp > num_generate and changeIndexToChar[predicted_id]
== " "):
            break

    return (start_string + ''.join(text_generated))
```

Bardzo istnotny jest również parametr funkcji o nazwie **temperature**. Określa on jak bardzo przewidywalny jest nasz tekst, gdzie wartość 1 oznacza tekst składający się z zupełnie losowych słów.

W naszym projekcie zastosowaliśmy pętle while. W celu kontrolowania ilości wykonanych iteracji, wprowadziliśmy zmienną “iterTmp” zainicjowaną początkową wartością 0. Przy każdym

wykonaniu pętli zmienna jest inkrementowana, a na końcu wykonania każdego kolejnego przejścia pętli sprawdzamy, czy zmienna ta nie jest przypadkiem większa od zadanej przez nas wartości 40 oraz czy znak, który jest dodawany nie jest spacją. Pozwala to na dokończenie słowa. W sytuacji uzyskania tej wartości, pętla przestaje być wykonywana, a program kończy swoje działanie, wyświetlając oczekiwany rezultat.

Przed wykonaniem pętli jest budowany wektor startowy.

Wykonanie pętli składa się głównie z trzech etapów:

Remove the batch dimension, apply temperature
W tym etapie najważniejszą rolę odgrywa funkcja squeeze(), która dokonuje operacji, polegającej na zwracaniu tensora (obiektu matematycznego, będącego uogólnieniem pojęcia wektora), który nadal jest tego samego typu, ale pozbawiony wszystkich wymiarów o rozmiarze 1. Ważne jest również to, że w tym etapie dokonuje się uwzględnienie wspomnianego wyżej istotnego dla naszego modelu - parametru **temperature**.

Predict the character using a categorical distribution
W tym miejscu zostaje wykorzystane wcześniej wyliczone “predictions”, czyli wspomniany już tensor, pozbawiony wymiarów o rozmiarze 1. W tym etapie zostaje ono przekazane jako argument do funkcji random.categorical(), która odpowiada za generowanie próbek z rozkładu kategorycznego. Zwracany rezultat jest zapisywany pod zmienną “predicted_id”.

Change the predicted characters
Predicted_id jest w tym etapie przekazywana jako argument funkcji o nazwie expand_dims(). Funkcja odpowiada za zwracanie tensoru z osią długości 1 wstawioną w indeksie podanego jako drugi argument.

3. Podsumowanie

Projekt generujący tytuły publikacji naukowych został wykonany i zwraca te tytuły. Zastosowanie rozwiązanie sieci neuronowych wpłynęło pozytywnie. Sieci rekurencyjne posiadają połączenia do wcześniejszych warstw. Na bazie poprzedniego stanu RNN oraz danych wejściowych możemy przewidywać klasę następnego znaku.

4. Bibliografia

[1] „Wikipedia,” [Online]. Available: https://pl.wikipedia.org/wiki/ArXiv.

[2] UMK, „is.umk,” [Online]. Available: chrome-extension://efaidnbmnnnibpcajpcglclefindmkaj/https://www.is.umk.pl/~grochu/wiki/lib/exe/fetch.php?media=zajecia:nn_2018_1:03-rnn.pdf.

[3] „towardsdatascience,” [Online]. Available: https://towardsdatascience.com/generating-scientific-papers-titles-using-machine-learning-98c8c9bc637e.

[4] „Tensorflow,” [Online]. Available: https://www.tensorflow.org/text/tutorials/text_generation.

[5] „Github,” [Online]. Available: https://github.com/csinva/gpt-paper-title-generator.