Spelling corrector

Projekt PJN

Autor: Jan Kozieł

1. Abstrakt

Niniejszy dokument ma za zadanie przedstawić projekt pt. *Spelling corrector* który został utworzony w ramach przedmiotu PJN. W dokumencie zostały zawarte informacje o problemach które zostały rozwiązane w ramach projektu, jak sam projekt został utworzony, jakie technologie zostały zastosowane oraz jak projekt działa w jego finalnej wersji.

2. Wstęp

Jak tytuł projektu wskazuje, w ramach projektu powinien zostać utworzony system który pozwala na poprawianie wpisanego przez użytkownika tekstu. Oczywiście sama tematyka jest bardzo obszerna, dlatego postawione zostały pewne założenia:

- Rozwiązaniem projektu będzie oprogramowanie które pozwoli użytkownikowi na wpisanie dowolnego tekstu oraz poprawienie ostatnio wpisanego wyrazu
- Program będzie obsługiwał wyłącznie język angielski
- Poprawa tekstu będzie polegać na wyświetleniu przycisków z sugerowanymi, poprawionymi wersjami wyrazu który ostatnio został wpisany. Po kliknięciu dowolnego z przycisków ostatnio wpisany wyraz zostanie zastąpiony tym poprawnym
- Jako dodatkowa funkcjonalność, zostanie też dodana możliwość podpowiadania kolejnego wyrazu na podstawie już wpisanych. Funkcjonalność ta zostanie uaktywniona gdy użytkownik wpisze już co najmniej jeden wyraz i wyraz który aktualnie jest wpisywany ma długość co najwyżej jednego znaku.

Problem poprawiania pisowni pojawia się w wielu miejscach - od edytorów tekstowych poprzez wyszukiwarki internetowe aż po wypełnianie formularzy. Sama zasada działania oprogramowania do poprawiania tekstu jest bardzo prosta - w dużym skrócie, program musi znaleźć wyraz (lub zdanie, w bardziej rozbudowanych projektach) którego prawdopodobieństwo pojawiania się (biorąc pod uwagę już wpisanych tekst) jest większe niż wyrazu który aktualnie jest wpisany.

Ze względu na to, projekt w dużej mierze operuje na tzw. Modelowaniu języka naturalnego (ang. *Language modeling, LM*), który, z definicji, jest modelem języka która określa prawdopodobieństwa danych kombinacji wyrazów[6]. Model ten jest zazwyczaj tworzony na podstawie różnych metod szacunkowych.

Jednym z najpopularniejszych rodzajów modeli realizujących to zadanie są modele oparte na n-gramach. Inne , przykładowe, modele które należą do tej rodziny to: *Transformer-based Language Models*[7], *Recurrent Neural Networks* i *Recursive Neural Networks*[8].

3. Rozwinięcie

Projekt można podzielić w większości na dwie części:

- część generującą możliwe poprawne słowa
- część obliczającą jak wysokie prawdopodobieństwo jest, że dane słowo zostanie użyte
- część generująca sugerowane słowa

W wyniku połączeniu tych części, program będzie mógł podpowiedzieć najbardziej prawdopodobne słowo, efektywnie działające jako *Spelling corrector*.

3.1. Generowanie możliwych słów

Ta część w dużej mierze bazuje na kodzie który został przedstawiony w *How to Write a Spelling Corrector*[1]. Autor kodu sugeruje podejście które można streścić jako:

1. Generuj ciągi znaków dopóki nie wygenerowane zostanie istniejące słowo
2. Posortuj listę wygenerowanych słów w zależności od częstości występowania słów w przykładowym zestawie danych
3. Zwróć słowo z największym prawdopodobieństwem

To podejście ma jednak kilka sporych wad. Po pierwsze zestaw danych który został użyty podczas sprawdzania czy dane słowo istnieje, a także z jakim prawdopodobieństwem występuje, to jedynie konkatenacja kilkudziesięciu najpopularniejszych książek. Ze względu na to, w projekcie, do tego został użyty inny zestaw danych - lista 20 000 najpopularniejszych słów oszacowanych na podstawie analizy częstotliwościowej korpusu używanego przez Google[2].

Po drugie, kod zakładał, że jeśli zostało wygenerowane jakieś istniejące słowo, to nie ma już potrzeby generować innych, możliwych poprawek. Autor podjął taką decyzję, ze względu na to, że jako jedno z kryteriów prawdopodobieństwa które słowo powinno zostać podpowiedziane, przyjął ile to teoretycznych błędów użytkownik mógł popełnić pisząc dane słowo. Ze względu na to, że w tym projekcie, ocena prawdopodobieństwa danego słowa odbywa się w inny sposób (patrz rozdział 3.2), przed sortowaniem generowane jest jak najwięcej możliwych słów.

Dodatkową zmianą która została wprowadzona względem kodu proponowanego przez Petera Norviga, jest poprawa funkcji generującej wyrazy, pod względem poziomu zagnieżdżenia. Poprawiona została ona w taki sposób aby mogła przyjmować poziom zagnieżdżenia edycji zamiast definiować osobne funkcje na każdy z nich.

```
def _get_all_possible_edits(self, word, nesting_level):
    if nesting_level <= 1:
        return self._generate_edits(word)
    return flatten_list(self._get_all_possible_edits(edit, nesting_level - 1) for edit in self._generate_edits(word))
```

Zdj. 1 - rekurencyjna funkcja pozwalająca na generowanie ciągów znaków o dowolnym zagnieżdżeniu

Sam kod został także przepisany na bardziej klasowe podejście, tak aby interfejs korzystania z niego był trochę bardziej przejrzysty. Kluczowe elementy jednak zostały takie same. Warto więc je omówić:

```
def _generate_edits(self, word):
    "All edits that are one edit away from `word`."
    letters    = 'abcdefghijklmnopqrstuvwxyz'
    splits     = [(word[:i], word[i:])    for i in range(len(word) + 1)]
    deletes    = [L + R[1:]               for L, R in splits if R]
    transposes = [L + R[1] + R[0] + R[2:] for L, R in splits if len(R)>1]
    replaces   = [L + c + R[1:]           for L, R in splits if R for c in letters]
    inserts    = [L + c + R               for L, R in splits for c in letters]
    return list(set(deletes + transposes + replaces + inserts))
```

Zdj. 2 - funkcja generująca ciągi znaków które mogą być wyrazami[1]

Powyższy fragment kodu przedstawia funkcję która jest sercem programu - generuje ona różne ciągi znaków na podstawie wpisywanego aktualnie słowa. Wygenerowane ciągi znaków niekoniecznie są rzeczywistymi słowami (to walidowane jest w kolejnym etapie). Funkcja ta generuje różne rodzaje typowych pomyłek które użytkownik może zrobić podczas wpisywania wyrazu, czyli np. pominięcie jakiegoś znaku, wpisanie nowego znaku, przestawienie znaków, etc.

```
def _known(self, words):
    "The subset of `words` that appear in the dictionary of self._words."
    return list(set(w for w in words if w in self._words))
```

Zdj. 3 - funkcja sprawdzający czy dany ciąg znaków jest słowem

Powyższy fragment kodu używany jest w kolejnym etapie, zaraz po generacji. Pozwala on na odfiltrowanie ciągów znaków które nie są poprawnymi słowami.

```
def candidates(self, word):
    "Generate possible spelling corrections for word."
    return list(set(self._known([word]) + self._known(self._get_all_possible_edits(word, 2))))
```

Zdj. 4 - funkcja zwracająca możliwe poprawki dla danego słowa

Powyższy funkcja jest już elementem publicznego interfejsu obiektu - zwraca ona wszystkie unikalne, sugerowane poprawki dla danego, wpisanego wyrazu. Różni się ona trochę od funkcji która została zaprezentowana w kodzie Petera Norviga - zwracane są wszystkie możliwe poprawki, z powodów wspomnianych wcześniej a także można dowolnie ustawić poziom zagnieżdżenia poprawki.

```
def suggest_corrections(self, word, prev):
    suggestions = list(self.candidates(word))
    suggestions.sort(key = lambda word: self._P(word, prev), reverse = True)

    return suggestions
```

Zdj. 5 - funkcja zwracająca posortowane poprawki dla danego słowa

Powyższy kod też trochę różni się od analogicznej funkcji Petera Norviga - podczas sortowania sugestii, brane pod uwagę jest też całe poprzednie zdanie.

3.2. Obliczanie prawdopodobieństwa wystąpienia słowa

```python
def _P(self, word, prev = None):
    "_Probability of `word`."

    base_score = self._words[word] / self._N if word in self._words else 0

    if prev is None:
        return base_score

    model_score = self._model.score(word, prev)
    return model_score * model_score_multiplier if model_score != 0 else base_score / model_score_multiplier
```

Zdj. 6 - funkcja obliczająca prawdopodobieństwo, że dane słowo jest tym poprawnym

Powyższy kod też ponownie jest trochę zmodyfikowanym kodem oryginalnie znajdującego się na stronie Petera Norviga - dodany został element w którym to wytrenowany model ocenia prawdopodobieństwo wystąpienia tego słowa. Jeśli model oceni poprawnie dane słowo (może się zdarzyć, że model zwróci 0, ze względu np. że w danych treningowych, dana kombinacja nie wystąpiła), to ocena ta jest wielokrotnie wzmacniana, aby przy sortowaniu dominowała ocenę na podstawie częstotliwości występowania słowa w języku angielskim.

Tak jak można się domyślić, celem modelu jest oszacowanie prawdopodobieństwa jakie dane słowo ma na wystąpienie biorąc pod uwagę poprzednie słowa. Sam model opiera się o bibliotekę *nltk*[9] i jest to *Maximum likelihood estimation model* oparty o n-gramy. Model ten został stworzony głównie w oparciu o artykuł *N-gram Language Model with NLTK*[5].

N-gramy to po prostu sekwencje n elementów w relacji do języka naturalnego (zazwyczaj tym elementem są wyrazy ale mogą nim też być inne elementy języka np. sylaby). Dla przykładu, ze zdania “Ala ma kota”, można utworzyć następujący n-gramy o rozmiarze 2 (bigramy):

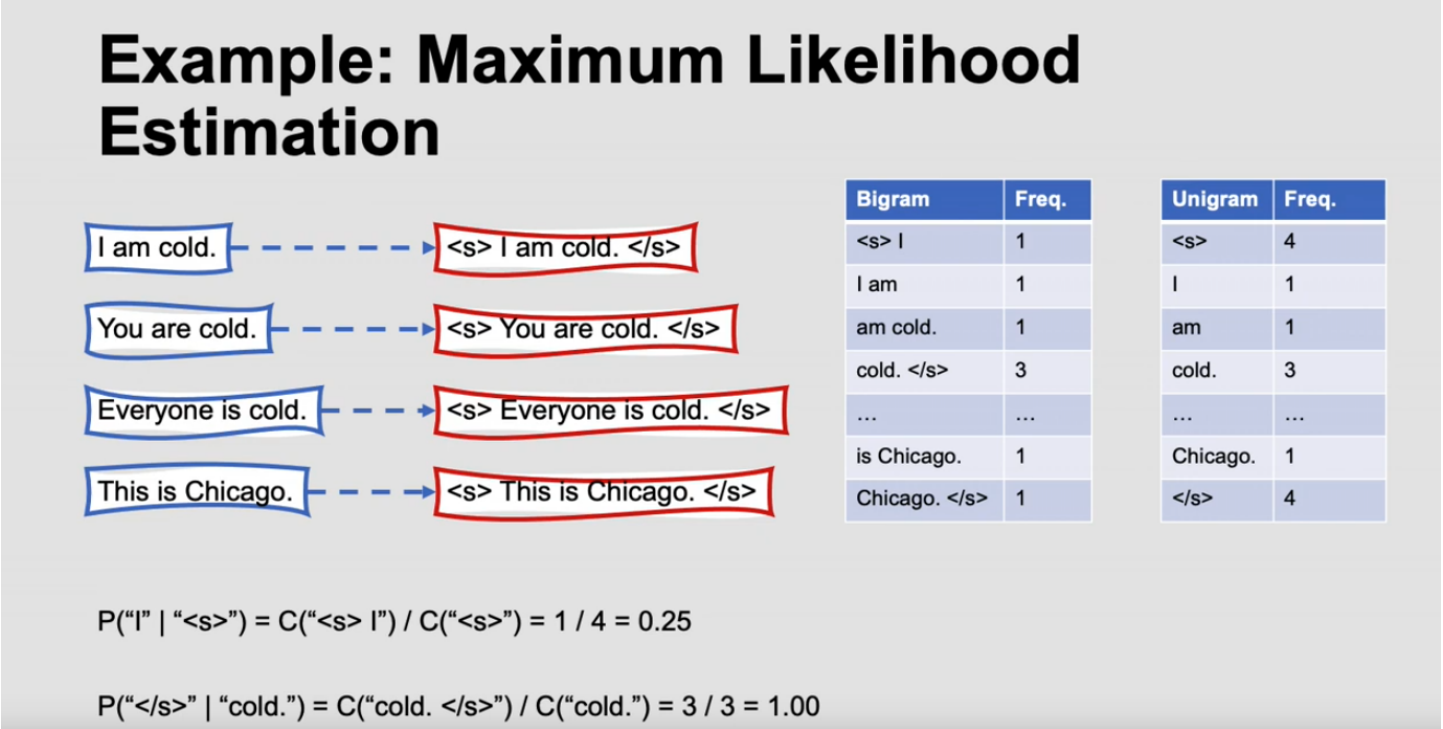
- ala ma
- ma kota

Oczywiście, im dłuższe zdanie tym więcej n-gramów. Teoretycznie najbardziej “poprawnym” modelem byłby model który zapisuje ile dokładnie takich samych zdań pojawiło się w ogólnej historii języka. Jednak ze względu na to, że język naturalny jest bardzo swobodny, trudno przewidzieć każdą możliwą kombinację wyrazów która może być użyta. Przez to, modele używające n-gramów, opierają się o tzw. *Markov’s assumption*, które mówi, że prawdopodobieństwo wystąpienia danego słowa, biorąc pod uwagę całe zdanie, można przybliżyć do prawdopodobieństwa wystąpienia danego słowa biorąc jedynie kilka słów pod uwagę. Dzięki temu łatwo można obliczyć prawdopodobieństwa, ze względu na to, że np. trójki danych wyrazów występują znacznie częściej niż inne.

Model MLE opiera się na n-gramach. Jak łatwo można się domyślić, model ten oblicza prawdopodobieństwo wystąpienia danego słowa bazując na n poprzednich słowach. Działa on w bardzo prosty sposób:

- wygeneruj wszystkie możliwe n-gramy na podstawie dostępnych danych
- oblicz ile razy dany n-gram występuje
- podziel tą wartość przez ilość wystąpień n-gramu o n mniejszym o 1.

To działanie w prosty sposób pozwala na ocenienie prawdopodobieństwa wystąpienia danego słowa na podstawie poprzednich.



Zdj. 7 - działanie modeli MLE[3]

Sam model oczywiście musiał zostać wytrenowany. Jako dane wejściowe został użyty zestaw 900 000 tweetów (z kategorii jako *blog*)[4].

```
n = 3
def get_model():
    if path.isfile('model.pkt'):
        with open('model.pkt', 'rb') as input:
            return dill.load(input)
    else:
        with open('./en_US.blogs.txt', encoding="utf8") as source:
            tokenized_text = []
            word_pattern = re.compile("^[a-zA-Z].*[a-zA-Z]$")

            for (index, tweet) in enumerate(list(source)):
                for sentence in sent_tokenize(tweet):
                    print(f"Parsing sentence {index}...")
                    parsed_sentence = list(map(str.lower, filter(lambda word: word_pattern.match(word), word_tokenize(sentence))))

                    if len(parsed_sentence) != 0:
                        tokenized_text.append(parsed_sentence)

            train_data, padded_sents = padded_everygram_pipeline(n, tokenized_text)

            model = MLE(n)
            model.fit(train_data, padded_sents)
            model.generate()

            with open('model.pkt', 'wb') as output:
                dill.dump(model, output)

            return model
```

Zdj. 8 - funkcja tworząca / ładująca model

Powyższy kod obrazuje jak dane zostały przygotowane przed tym jak model został do nich dopasowany. Jak widać, usuwane są “słowa” które nie zaczynają i nie kończą literą. Decyzja taka została podjęta ze względu na

to, że dane treningowe zawierały dosyć dużo samych liczb. Oczywiście w kontekście przewidywania następnego słowa, nie ma większego sensu uznawać konkretną liczbę za słowo. Jak widać też, model jest zapisywany do pliku, aby nie musieć trenować go przy każdym uruchomieniu programu.

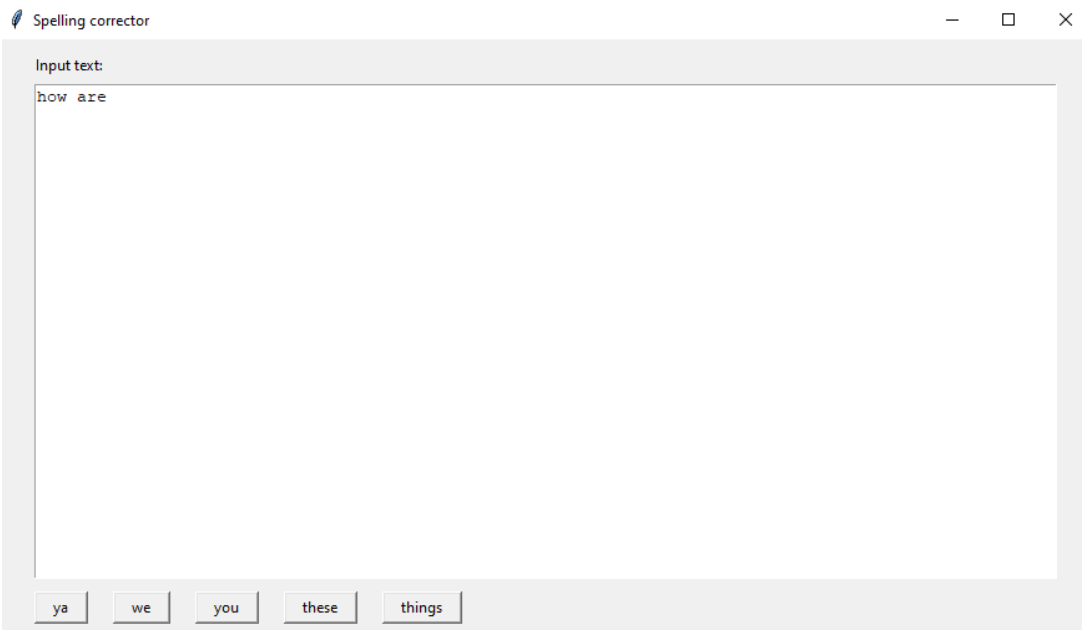
3.3. Predykcja kolejnych wyrazów

Wytrenowany model jest aktualnie stosowany jako jeden ze sposobów sortowania sugerowanych poprawek. Jednak z względu na jego działanie, w prosty sposób można sprawić aby model generował kolejne wyrazy zanim użytkownik zacznie je pisać. Aby to osiągnąć, wystarczy wziąć pod uwagę słowa które model przewiduje, że będą następne (na bazie stworzonych n-gramów)

```python
def predict_new(self, prev, amount):
    predictions = set()
    max = amount * 2

    while len(predictions) < amount and max != 0:
        max -= 1
        new_prediciton = self._model.predict(prev, randrange(0, 10000))

        if new_prediciton != self._padding_token:
            predictions.add(new_prediciton)

    return list(predictions)
```

Zdj. 9 - Funkcja generująca nowe, sugerowane wyrazy.

3.4. Interfejs użytkownika



Zdj. 9 - interfejs użytkownika w trybie podpowiadania słów

Tak jak to zostało wspomniane, interfejs programu jest bardzo prosty - składa się z pola tekstowego oraz przycisków, które w zależności od stanu pola tekstowego albo przewidują kolejne słowo albo sugerują

poprawne słowo. Po kliknięciu przycisku ze słowem albo jest ono dodawane do tekstu albo ostatnio wpisane słowo jest zastępowane tym wybranym, w zależności od trybu w którym aplikacja pracuje

Zdj. 10 - interfejs użytkownika w trybie poprawiania wpisanego słowa

4. Podsumowanie

Przedstawione rozwiązanie spełnia wymagania i założenia postawione na początku projektu. Oczywiście program mógłby być dalej usprawniony, np przez dodanie możliwości poprawienia wyrazów które znajdują się w środku zdania. Podobnie może zostać usprawniona metoda generująca słowa - sam wytrenowany model potrafi także generować słowa ale nie jest on konieczne dostosowany do tego aby generować możliwe poprawione słowa.

Ogólnie efekt działania programu wydaje się zadowalający - tam gdzie celowo popełniany jest błąd, program poprawnie przewiduje jakie słowo powinno zostać użyte.

Wnioski które można wysnuć z projektu to:

- n-gramowa reprezentacja języka naturalnego jest dobra ale nie zawsze poprawnie przewiduje kolejne słowa (jeśli przestawiony zostanie szyk zdania, to podmiot bardzo często może się “zgubić”)
- implementacja *spelling corrector* w dużej mierze zawsze będzie opierać się o oszacowanie prawdopodobieństwa wystąpienia danego słowa. To jak to zostanie oszacowane jest kluczowym elementem który rozróżnia dobre implementacje od gorszych.

5. Bibliografia

1. Peter Norvig - http://www.norvig.com/spell-correct.html - dostęp 14.05.2023
2. Josh Kaufman - https://github.com/first20hours/google-10000-english - dostęp 14.05.2023
3. Natalie Parde https://www.natalieparde.com/teaching/cs_421_fall2020/N-Grams%20and%20Maximum%20Likelihood%20Estimation.pdf - dostęp 14.05.2023

4. Carlos Rafael - https://www.kaggle.com/datasets/crmercado/tweets-blogs-news-swiftkey-dataset-4million - dostęp 14.05.2023
5. Liling Tan - https://www.kaggle.com/code/alvations/n-gram-language-model-with-nltk/notebook - dostęp 14.05.2023
6. Ben Lutkevich - https://www.techtarget.com/searchenterpriseai/definition/language-modeling - dostęp 14.05.2023
7. Yule Wang - https://medium.com/@yulemoon/an-in-depth-look-at-the-transformer-based-models-22e5f5d17b6b - dostęp 14.05.2023
8. Akbar Karimi - https://www.baeldung.com/cs/networks-in-nlp - dostęp 14.05.2023
9. Bird, S., Klein, E. & Loper, E., 2009. *Natural language processing with Python: analyzing text with the natural language toolkit*, " O'Reilly Media, Inc."

Wykonał: Jan Kozieł, 134102, GL01