

# Jak napisać SPELL CORRECTOR

Przedmiot: **Przetwarzanie Języka Naturalnego**

Prowadzący przedmiot: **dr Radosław Kycia**

Autorzy raportu: **Łukasz Cichoń, Mariusz Hodana, Joanna Hornung**

<b>Abstrakt .....</b>	<b>2</b>
<b>1. Wstęp.....</b>	<b>3</b>
1.1. Cel projektu .....	3
1.2. Zakres projektu .....	3
1.3. Metodyka pracy .....	3
<b>2. Autokorekta i jej zastosowania.....</b>	<b>4</b>
2.1. Autokorekta.....	4
2.2. Różne sposoby budowania autokorektorów.....	4
2.3. Inteligentny <i>spell corrector</i> od Google .....	5
2.4. Podstawy teoretyczne niniejszej pracy .....	7
<b>3. <i>Spell corrector</i> na podstawie pracy Petera Norviga.....</b>	<b>8</b>
3.1. Inicjalizacja z dostawcami unigramów oraz opcjonalnie bigramów.....	8
3.2. Generowanie kandydatów .....	9
3.3. Korekcja zdania .....	10
3.4. Wybór najlepszej poprawki .....	11
3.5. Uruchomienie aplikacji uwzględniając argumenty linii komend.....	12
<b>4. Podsumowanie .....</b>	<b>14</b>
<b>Bibliografia .....</b>	<b>14</b>

## 1.1. Cel projektu

Celem projektu było opisanie procesu autokorekty, związanych z nim teorii matematycznych i algorytmów a także przedstawienie działania programu realizującego autokorektę.

## 1.2. Zakres projektu

Zakres projektu obejmował:

- wyszukanie artykułów źródłowych na temat autokorekty,
- przedstawienie znalezionych informacji,
- zbudowanie programu na podstawie dostępnych w Internecie instrukcji,
- opisanie działania programu.

## 1.3. Metodyka pracy

Podczas pracy nad projektem wykorzystano materiały znalezione w Internecie, edytor tekstu *MS Office* oraz środowisko programistyczne *Visual Studio Code*.

Pracę rozpoczęto od wyszukania artykułów źródłowych na temat autokorekty i przedstawienia informacji teoretycznych.

Następnie, na podstawie instrukcji dostępnej w Internecie, zbudowano odpowiedni program, przetestowano go i opisano jego działanie.

Na koniec przedstawiono otrzymane wyniki, wyciągnięto wnioski i podjęto dyskusję na ich temat.

## 2. Autokorekta i jej zastosowania

---

### 2.1. Autokorekta

Określenie *spell checking* oznacza sprawdzanie pisowni i rozpoznawanie tak zwanych literówek<sup>3</sup>. Natomiast *spell correction*, to proces polegający na korygowaniu tego typu błędów. Kiedy literówki, rozpoznawane przez program, poprawiane są automatycznie, mówimy o autokorekcie (ang. *autocorrection*). Autokorekta jest niezwykle przydatna, nie tylko podczas codziennego wpisywania fraz w wyszukiwarkach takich jak Google oraz w edycji tekstu (między innymi tekstu niniejszego raportu), ale też w systemach optycznego rozpoznawania liter (ang. *Optical Character Recognition*).

### 2.2. Różne sposoby budowania autokorektorów

Wyszukiwanie i korygowanie literówek, to jeden z najstarszych problemów procesowych, który przez lata zyskał wiele propozycji rozwiązań. Począwszy od najprostszych, bazujących na manualnie tworzonych leksykonach jako *źródłach prawdy*, po metody coraz bardziej empiryczne.

Najbardziej bezpośrednim podejściem jest modelowanie błędów a następnie odczytywanie ich przez algorytm, w oparciu o utworzoną manualnie bibliotekę wyrazów, tak zwany leksykon. Bazą teoretyczną takiego rozwiązania jest odległość łańcuchowa Damerau-Levenshtein<sup>2</sup>, będąca miarą liczby operacji edycyjnych, potrzebnych do przekształcenia jednego ciągu znaków w drugi. Operacje te obejmują dodanie, usunięcie, zamianę znaku oraz możliwość transpozycji, czyli zamiany miejscami dwóch sąsiednich znaków.

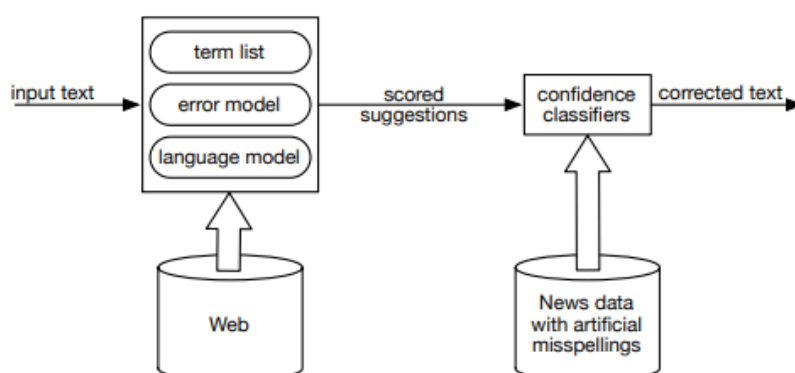
Algorytmy indeksowania fonetycznego, takie jak na przykład *Metaphone*, używany przez GNU Aspell<sup>1</sup>, reprezentują słowa według ich blisko brzmiącej wymowy i umożliwiają korektę w sytuacji, gdy ortografia się nie zgadza. Metoda ta opiera się na plikach danych, zawierających fonetyczne zapisy słów.

### 2.3. Inteligentny *spell corrector* od Google

Korzystanie z leksykonów ma oczywistą wadę, polegającą na tym, że leksykon nigdy nie będzie kompletny, ponieważ liczba słów w językach nowożytnych jest niemal nieograniczona.

Nowoczesnym podejściem do autokorekty jest system zaproponowany w 2009 roku przez czwórkę programistów z *Google*, to znaczy Casey Whitelaw, Bena Hutchin-sona, Grace Y Chung i Gerarda Ellisa<sup>3</sup>. Zespół ten zaprojektował i wdrożył *spell corrector*, z którego obecnie korzysta firma. Innowacja projektu polegała na tym, że autorzy nie zastosowali żadnej manualnie opracowanej biblioteki słów, ale statystyczne modele błędów oraz modele językowe, wraz z klasyfikatorami uczenia maszynowego.

Wcześniejsze *spell correctory*, używające jedynie modelu statystycznego, wymagały dużej liczby danych do wytrenowania systemu. Zastosowanie modeli językowych i uczenia maszynowego, pozwala na wykorzystanie źródeł internetowych do wytworzenia modelu błędów, opartego na punktacji podciągów. Nie ma tutaj biblioteki słów.



Rys. 1. Proces autokorekty i wykorzystywane w nim źródła

Główne elementy procesu autokorekty, zaproponowanego przez zespół *Google*, przedstawia rysunek 1. Pierwszym krokiem do stworzenia systemu było zbadanie materiałów www, pod kątem najczęściej występujących w nich słów. W tym celu zbadano ponad miliard stron www, podzielono je na jednostki (tokeny), przefiltrowano je, odrzucając jednostki niebędące prawdziwymi słowami (zawierające zbyt dużo elementów interpunkcyjnych, zbyt długie lub zbyt krótkie) i wybrano dziesięć milionów naj-

popularniejszych jednostek (ang. *term list*).

Na podstawie powstałej listy, przy użyciu modelu kanału szumów (ang. *noisy channel model*), tworzony jest model błędów (ang. *error model*). Model kanału szumów zakłada, że dla zaobserwowanego słowa  $w$  (słowo wejściowe, ang. *input text*) i potencjalnego słowa korygującego  $s$  (z listy) przyjmuje się, że:  $P(s|w) = P(w|s) \times P(s)$ .

Do wytrenowania modelu błędów potrzebne są trójki danych w postaci: (potencjalne słowo korygujące, słowo wejściowe, licznik). Takie trójki danych treningowych budowane są na podstawie listy słów *term list*. Korzystając z algorytmu odległości Levenshtein-Damerau, do każdego słowa przypisywane jest potencjalne słowo korygujące i licznik występowania takich par. Autorzy pracy uznali, że Internet jest idealnym źródłem do trenowania systemu, ponieważ występują w nim zarówno słowa napisane poprawnie jak i te zawierające literówki. Należy zaznaczyć, że błędy w analizowanych słowach występują w 80% pojedynczo jako nadmiarowy znak, brakujący znak lub zamiana znaku na niewłaściwy. Ponadto słowa z błędami najczęściej zapisane są fonetycznie poprawnie (w przypadku języka angielskiego), ale ortograficznie źle.

Na zakończenie procesu tworzenia trójek danych, słowa odnoszone są do kontekstów, w których występują. Dla każdego słowa arbitralnie tworzony jest zbiór kontekstów. Kontekst występujący najczęściej staje się najbardziej prawdopodobny podczas korekty. Przykładowo, jeśli słowo *accidental* występuje najczęściej w jakimś kontekście, to gdy w tym samym kontekście pojawi się *occidental*, algorytm autokorekty uzna, że jest to literówka. Na podstawie analizy kontekstowej, powstają modele językowe  $n$ -gram (ang. *n-gram language models*), to znaczy zbiory słów stanowiące konteksty o liczebności  $n$ .

W kolejnym etapie, sugerowane słowa korygujące przechodzą weryfikację przy użyciu sztucznie zbudowanego zbioru błędnie zapisanych słów. W efekcie tej weryfikacji otrzymywane są klasyfikatory zaufania dla słów korygujących. Klasyfikatory wyliczyć można na dwa sposoby:

- klasyfikator prosty – polegający na porównaniu logarytmów prawdopodo-

bieństwa najwyższej punktowanych słów korygujących,

- klasyfikator regresji logistycznej, biorący pod uwagę dodatkowo różnicę wyniku modelu językowego dla słowa do korekty i słowa korygującego oraz różnice logarytmów dla niżej punktowanych sugestii a ponadto liczbę dostępnych sugestii, długość tokenu oraz liczbę kontekstów.

Na podstawie otrzymanych klasyfikatorów zaufania, wyłaniane jest najbardziej prawdopodobne słowo korygujące.

Wydajność systemu została potwierdzona poprzez zbiór danych napisanych przez człowieka. Skonstruowany w ten sposób *spell corrector* jest łatwy do zaadaptowania w praktycznie każdym języku naturalnym.

## 2.4. Podstawy teoretyczne niniejszej pracy

*Spell corrector*, zbudowany przez autorów niniejszej pracy, jest systemem uproszczonym względem systemu *Google*, opartym na instrukcji stworzonej przez Petera Norviga, opublikowanej na blogu <https://www.norvig.com/spell-correct.html><sup>4</sup>.

Instrukcja Petera Norviga opiera się na teorii prawdopodobieństwa Bayesa, która zakłada, że  $\text{argmax}_{c \in \text{candidates}} P(c) \times P(w|c)$ , gdzie:

- $w$  to słowo do skorygowania,
- $c$  to potencjalne słowo korygujące,
- $\text{argmax}$  to kandydat o najwyższym prawdopodobieństwie,
- $P(c)$  to model językowy oznaczający prawdopodobieństwo, że  $c$  jest słowem występującym w danym języku, np. słowo *the* stanowi około 7% wystąpień różnych słów w języku angielskim, więc  $P(\text{the}) = 0.07$ ,
- $P(w|c)$  to model błędów, oznaczający prawdopodobieństwo tego, że w miejsce zapisanego przez autora słowa  $w$ , powinno w rzeczywistości zostać wpisane słowo  $c$ , np.  $P(\text{teh}|\text{the})$  będzie relatywnie wysokie, natomiast  $P(\text{theexyz}|\text{the})$  bardzo niskie.

### 3. *Spell corrector* na podstawie pracy Petera Norviga

---

Główną klasą, która odpowiada za działanie algorytmu jest klasa [SpellCorrector](#). Poniżej, na jej podstawie, zostaje omówiony sam algorytm krok po kroku.

#### 3.1. Inicjalizacja z dostawcami unigramów oraz opcjonalnie bigramów

Gdzie unigramy występują w postaci pojedynczych słów, są traktowane jako niezależne jednostki, natomiast bigramy składają się z dwóch kolejnych słów w tekście, pozwalając na określenie kontekstu na podstawie ich współwystępowania.

```
class SpellCorrector(object):
def __init__(self, words_provider, bigrams_provider=None):
    self.wp = words_provider
    self.bp = bigrams_provider
```

Mamy różne klasy które są odpowiedzialne za dostarczanie informacji o znanych słowach oraz bigramach, takie jak [KnownWordsProviderUsingRAM](#), [KnownWordsProviderUsingBigFile](#), [KnownWordsProviderUsingMultipleFiles](#) dla unigramów oraz [BigramsProvider](#) dla bigramów. Każda z tych klas służy do udostępniania informacji o znanych słowach, różnica w nich polega na sposobie zarządzania danymi oraz wymaganiami wydajnościowymi i operacyjnymi:

- klasa [KnownWordsProviderUsingRAM](#), ta klasa ładuje cały zbiór unigramów (słów i częstości ich wystąpienia) bezpośrednio do pamięci RAM. Stosowane przy małej ilości danych,
- klasa [KnownWordsProviderUsingBigFile](#), ta klasa korzysta z jednego dużego pliku do przechowywania unigramów *i czyta z niego dane na żądanie, nie ładując wszystkiego do pamięci RAM*. Stosowane, gdy ilość danych w jednym pliku jest zbyt duża,
- klasa [KnownWordsProviderUsingMultipleFiles](#), ta klasa dzieli dane na wiele plików, organizując je według pierwszej litery każdego słowa, dzięki czemu mo-



zemy operować na dużych zbiorach danych ładując tylko te segmenty które nas interesują.

Klasy te posiadają metody:

- *know* która sprawdza, które z podanych słów są znane, czyli znajdują się w zbiorze danych(unigramowych),
- *P* obliczająca prawdopodobieństwo wystąpienia danego słowa w oparciu o jego częstość.

Klasa *BigramsProvider* nie ładuje wszystkich biogramów do pamięci RAM, zamiast tego odczytuje je z plików na żądanie. Klasa ta także posiada metody:

- *know* która ma za zadanie zidentyfikować, które z podanych słów mogą utworzyć znany bigram z danym poprzedzającym słowem('previous\_word).
- *P*, która oblicza prawdopodobieństwo wystąpienia danej pary słów na podstawie zgromadzonego zbioru tekstowego.

### 3.2. Generowanie kandydatów

```
def _candidates (self, word):
    diacritics_words = self.wp. known (self. _add_diacritics(word))
    known_word = self.wp. known([word])

    if diacritics_words:
        return known_word. union(diacritics_words)
    else:
        return known_word or self.wp. known (self. _edits1(word)) or self.wp.
known(self._edits2(word)) or [word]
```

Jest to metoda odpowiedzialna za generowanie zestawu kandydatów na poprawkę słowa, którzy mogą być odpowiednią wersją błędnie napisanego słowa.

Na samym początku zostają dodane polskie znaki diakrytyczne za pomocą metody *add\_diacritics*.

```
def _add_diacritics(self, word):
    pl_edits1 = self._edit1_diacritics(word)
    pl_edits2 = list(e2 for e1 in pl_edits1 for e2 in self._edit1_diacritics(e1))
    #print(pl_edits1.union(pl_edits2))
    return pl_edits1.union(pl_edits2)
```

```

def _edit1_diacritics(self, word):
    pairs = {
        'a': 'ą',
        'c': 'ć',
        'e': 'ę',
        'l': 'ł',
        'n': 'ń',
        'o': 'ó',
        's': 'ś',
        'z': 'ż' # , 'ź']
    }

    splits = [(word[:i], word[i:]) for i in range(len(word) + 1)]
    e1 = list()
    for orig_letter, new_letter in pairs.items():
        for L, R in splits:
            if R and R[0] == orig_letter:
                e1.append(L + new_letter + R[1:])
    return set(e1)

```

Następnie warianty ze znakami diakrytycznymi są sprawdzane przez *words\_provided* (zbiór poprawnych słów), sprawdzający które słowa występują w słowniku poprawnych słów za pomocą metody *know*, która została wymieniona wyżej.

Jeśli zostały znalezione znane warianty z diakrytykami, to łączone są one z oryginalnymi znanymi słowami za pomocą *union*, która łączy je w zbiór unikalnych elementów i zwraca je jako wynik.

Natomiast jeśli żadne słowo nie jest znane, zostaje ono zmodyfikowane poprzez dodanie, zamianę, usunięcie albo przestawienie liter za pomocą funkcji *edits1* oraz *edits2*, a następnie za pomocą metody *know* sprawdzamy czy po edycji występuje w słowniku poprawnych słów. Jeśli znaleziono jakieś znane edycje, są zwracane jako wynik, w przeciwnym razie zwracamy oryginalne słowo.

### 3.2. Korekcja zdania

```

def sentence_correction(self, sentence, print_words=True):
    words_to_correct = sentence.split()
    corrected_sentence = ""
    corrected_word = None
    for word in words_to_correct:
        corrected_word = self.correction(word, corrected_word)
        if print_words:
            sys.stdout.write(corrected_word + " ")
            sys.stdout.flush()
        corrected_sentence += corrected_word + " "
    return corrected_sentence.rstrip()

```

W metodzie zdanie wejściowe jest podzielone na poszczególne słowa, dla każdego słowa zostaje wywołana metoda `correction` z bieżącym słowem oraz ostatnio poprawionym słowem. Następnie na podstawie flagi (w tym wypadku `true`) skorygowane słowo może zostać wypisane na wyjście. Niezależnie od tego czy słowo jest drukowane, zostaje ono dodane do `corrected_sentence` wraz ze spacją, w ten sposób tworzone jest skorygowane zdanie, które jest zwracane usuwając wszystkie zbędne białe znaki. Jest to metoda zwracająca rezultat poprawionego już ciągu słów.

### 3.4. Wybór najlepszej poprawki

```
def correction(self, word, previous_word=None):
    if previous_word and not isinstance(previous_word, str):
        previous_word = str(previous_word, 'utf-8')
    candidates = self._candidates(word)
    sorted_candidates = list(sorted(candidates, key=self.wp.P, reverse=True))
    if self.bp and previous_word:
        return self._correct_using_bigrams(sorted_candidates, previous_word)
    else:
        return sorted_candidates[0]
```

Metoda `correction` zarządza wyborem najlepszej poprawki, biorąc pod uwagę unigramy jak i opcjonalnie bigramy.

Na samym początku metoda sprawdza czy jest podane `previous_word`, które jest kluczowe w kontekście biogramów, ponieważ przechowuje słowo, które poprzedza obecnie analizowane słowo. Dodatkowo sprawdza czy słowo jest typu `String` jeśli nie, to konwertuje je. Następnie wywołuje wewnętrzną metodę `candidates`, która jak było wcześniej podkreślone generuje listę potencjalnych kandydatów. Lista zostaje posortowana malejąco uwzględniając prawdopodobieństwa występowania słowa w zbiorze danych tekstowych i zwraca kandydata, który najczęściej występuje. Natomiast jeśli istnieje `bigrams_provided` oraz `previous_word` to brane zostaje pod uwagę nie tylko prawdopodobieństwo występowania słowa ale także prawdopodobieństwo występowania połączonych słów za pomocą metody `know` znajdującej się w `correct_using_bigrams`, która jest zwracana jako rezultat metody.

```
def _correct_using_bigrams(self, sorted_candidates, previous_word):
    known_bigrams = self.bp.known(sorted_candidates, previous_word)
    if len(known_bigrams) > 0:
        sorted_bigrams = sorted(known_bigrams, key=lambda w: self.bp.P(
            ".join([previous_word, w])), reverse=True)
        return sorted_bigrams[0]
    else:
        return sorted_candidates[0]
```

Metoda ta jest odpowiedzialna za wykorzystanie bigramów. Na początku sprawdza które z posortowanych kandydatów zgadzają się w połączeniu z danym *previous\_word* tworząc bigramy rozpoznawalne przez bazę tekstową za pomocą metody *know*. Następnie przypisuje je do listy pojedynczych kandydatów na poprawkę. Lista tych kandydatów zostaje posortowana wedle prawdopodobieństwa wystąpienia, które jest obliczane przez funkcję *self.bp.P*. Prawdopodobieństwo zostaje obliczone na podstawie bazy tekstowej czyli *bigrams\_provider*. Tak jak poprzednim razem, sortowanie odbywa się malejąco. Następnie zostaje zwrócone najbardziej prawdopodobne słowo.

Klasa *NGramsUtils* składa się z funkcji pomocniczych, które są używane do przetwarzania tekstu, chodzi o pracę z bazą tekstową w postaci unigramów oraz biogramów.

### 3.5. Uruchomienie aplikacji uwzględniając argumenty linii komend

Poniżej został przedstawiony kod odpowiadający za logikę uruchomienia aplikacji uwzględniając różne parametry argumentów linii komend.

- 1) Na samym początku deklarowane są potrzebne moduły oraz biblioteka do obsługi argumentów linii komend.
- 2) Następnie mamy podane ścieżki do plików dla poszczególnych unigramów oraz biogramów, gdzie przechowywane są dane.

Różne argumenty wykorzystywane w SpellCorector:

- *b* (bigrams) włączenie użycia biogramów,

- w (word) skrypt działa w trybie nieinteraktywnym, gdzie słowo lub zdanie jest korygowane bez dalszej interakcji,
- t (type) pozwala wybrać typ providera dla unigramów:
  - „BigFile”
  - „MultipleFiles”
  - „Ram” który jest domyślną metodą.

```
#!/usr/bin/python
from KnownWordsProvider import KnownWordsProviderUsingRAM, KnownWordsProviderUsingBigFile,
KnownWordsProviderUsingMultipleFiles
from BigramsProvider import BigramsProvider
from SpellCorrector import SpellCorrector
import argparse

UNIGRAMS_FILEPATH = r'D:\Studia_Semestr_8\Przetwarzanie_jezyka_naturalnego\Projekt\Kod\n-
grams\1grams_fixed'
UNIGRAMS_FILES_DIR = r'D:\Studia_Semestr_8\Przetwarzanie_jezyka_naturalnego\Projekt\Kod\n-
grams\1grams_splitted'
BIGRAMS_FILEPATH = r'D:\Studia_Semestr_8\Przetwarzanie_jezyka_naturalnego\Projekt\Kod\n-
grams\2grams_splitted'

if __name__ == '__main__':
    parser = argparse.ArgumentParser()
    parser.add_argument("-b", "--bigrams", help="turn on using 2-grams to spell correction",
action="store_true")
    parser.add_argument("-w", "--word", help="non-interactive mode, correct specified
word(s)")
    parser.add_argument("-t", "--type", help="type of unigrams provider -
RAM/BigFile/MultipleFiles\nRAM is used by default")
    args = parser.parse_args()

    unigrams_path = UNIGRAMS_FILEPATH
    if args.type == "BigFile":
        words_provider = KnownWordsProviderUsingBigFile()
    elif args.type == "MultipleFiles":
        unigrams_path = UNIGRAMS_FILES_DIR
        words_provider = KnownWordsProviderUsingMultipleFiles()
    else:
        words_provider = KnownWordsProviderUsingRAM()
    words_provider.initialize(unigrams_path)

    bigrams_provider = None
    if args.bigrams:
        bigrams_provider = BigramsProvider()
        bigrams_provider.initialize(BIGRAMS_FILEPATH)

    corrector = SpellCorrector(words_provider, bigrams_provider)

    if args.word:
        corrector.sentence_correction(args.word)
        print("")
        exit(0)

    while True:
        text_to_correct = input("> ")
        corrector.sentence_correction(text_to_correct)
        print("")
```

## 4. Podsumowanie

---

W niniejszym projekcie opisano proces autokorekty, związane z nim teorie matematyczne i algorytmy a także przedstawiono działanie programu realizującego autokorektę, napisanego na podstawie tutorialu Petera Norviga. Cele projektu zostały zrealizowane, jednak aplikacja nie jest niezawodna.

## Bibliografia

---

1. Atkinson, 2009, *Gnu aspell*, <http://aspell.net> – data dostępu: maj 2024
2. Damerau, *A technique for computer detection and correction of spelling errors*, Communications of the ACM, 1964 (7): 171–176
3. Whitelaw, Hutchinson, Chung, Ellis, *Using the Web for Language Independent Spell-checking and Autocorrection*, Proceedings of the 2009 Conference on Empirical Methods in Natural Language Processing, 890–899
4. <https://www.norvig.com/spell-correct.html> – data dostępu: maj 2024