# Przetwarzanie języka naturalnego w praktyce. PyConPL 2013

dr inż. Krzysztof Dorosz
dorosz@agh.edu.pl

## Cel warsztatów

Celem tych warsztatów jest możliwość zapoznania się z ogólnie ujętą tematyką przetwarzania tekstu osób potencjalnie programujących już w Pythonie na poziomie minimum średniozaawansowanym, które wcześniej nie miały styczności z tymi zagadnieniami. Dobrano taki przykład na którym zademonstrowane zostaną różne podstawowe problemy i terminy występujące w tej dziedzinie (np. słownik fleksyjny, leksem, stop lista). Warsztaty mają charakter zdecydowanie bardziej praktyczny niż naukowy, to znaczy nie będziemy skupiać się na złożonych zagadnieniach ściśle lingwistycznych (jak choćby gramatyki formalne, drzewa rozbioru zdania, warstwa morfologiczna, itp), natomiast postaramy się wykonać bardzo praktyczny algorytm używając do tego ogólnie dostępnych i łatwych do zrozumienia technik wywodzących się z lingwistyki komputerowej. Być może dla niektórych warsztaty te będą pierwszym krokiem w stronę zagłębienia się w bardziej zaawansowaną lingwistykę komputerową.

## Dlaczego Python

Język Python posiada wiele cech, które sprawiają, że jest to język niezwykle dobrze przystosowany do maszynowego przetwarzania języka naturalnego. Wspierane przez język Python struktury danych wyższego poziomu, takie jak listy (zagnieżdżane) czy słowniki, wręcz idealnie nadają się do oddania złożonych, często hierarchicznych struktur, które możemy obserwować analizując tekst. Obecność konstrukcji takich jak filter, map, reduce, a także pojęcie generatorów i iteratorów umożliwiają zarówno dużą ekspresję każdej linii kodu jak i łatwość operowania na tych strukturach danych w praktyce. Pełne wsparcie dla Unicode jak i względna łatwość operowania tekstem w dowolnym kodowaniu, jest także niezwykle pomocna w przypadku pracy z tekstami szczególnie z heterogenicznych źródeł i formatów (jak choćby z Internetu). Warto także wspomnieć o względnej szybkości języka (jak na język interpretowany), a także o reklamowanym przez samo środowisko Pythona pojęciu “battery included” - czyli o ogromnej liczbie bibliotek pomocnych w dużym spectrum problemów (w tym językowych). Na koniec dodać należy także o dwóch często zapominanych w takich porównaniach aspektach, którymi są subiektywna łatwość pisania kodu i ilość efektywnych linii kodu potrzebnych do rozwiązania danego problemu. Metryki te są bezpośrednio związane także z pojęciem czytelności kodu. Okazuje się, że język Python dobrze wypada także w tej kategorii. Prowadząc przedmiot Przetwarzanie Języka Naturalnego na AGH w ramach pierwszego laboratorium semestru zawsze realizuje tzw. laboratorium wprowadzające podczas którego studenci 4 roku Informatyki IET proszeni są o implementację we wszystkich językach programowania, które

znają, prostego algorytmu generowania posortowanej listy frekwencyjnej z korpusu tekstu. Następnie studenci zadane mają poddać te implementacje subiektywnym i obiektywnym ocenom takim jak: łatwość i czytelność programowania w skali o 1-5 i liczba LOC. Zestawiając otrzymane wyniki dla całego roku, wśród języków takich jak C/C++, Java, PHP, Ruby i wiele innych Python z każdym rokiem w średnim ujęciu pokazuje swoją wyższość. Z opinii przedstawianych przez studentów wynika, że Python jest językiem wystarczająco szybkim w przetwarzaniu, zarazem o niezwykle dużej ekspresji pojedynczej linii kodu pozwalającej pisać bardzo krótki kod (a przez to dużo łatwiejszy w utrzymaniu, dużo szybszy do napisania, prawie zawsze łatwiejszy w czytaniu). Przeprowadzenie tego doświadczenia na pierwszych zajęciach sprawia, że co roku bardzo dużo moich studentów wybiera ten właśnie język, jako język implementacji programów na dalszych laboratoriach, mimo że dla wielu jest to pierwszy kontakt z tym językiem.

# Plan warsztatów

Celem zadań w ramach niniejszych warsztatów jest opracowanie rozwiązania, które potrafiło będzie wyceniać wartość produktu na podstawie tytułu aukcji. Skupimy się tutaj na analizie podobieństwa tytułów aukcji, w celu wyznaczenia średniej ceny produktu o zbliżonej nazwie. Zakładamy przy tym, że zbliżona nazwa aukcji opisywać będzie ten sam produkt. W tym celu dowiemy się:

1. Czym jest korpus tekstu i jak będziemy z nim pracować. Poznamy też narzędzie csvkit do wygodnego przetwarzania plików csv.
2. W jaki sposób poznać charakterystykę korpusu (słownik frekwencyjny).
3. Poznamy uniwersalne prawo Zipfa będące czasem określane prawem Pareto dla tekstu.
4. Zapoznamy się z metrykami liczenia odległości napisów i skonstruujemy mechanizm wyceny towarów po tytule aukcji.

# 0. Zanim zaczniesz - przygotowanie do warsztatów

1. Upewnij się, że posiadasz Pythona najlepiej w najnowszej wersji 2.7.x. w środowisku Linux lub MacOS.
2. Upewnij się, że twoja konsola posiada locale ustawione do używania UTF-8.
3. Zainstaluj pakiet gnuplot.
4. Zainstaluj virtualenv oraz virtualenvwrapper (jeśli jeszcze nie posiadasz).
5. Stwórz i aktywuj nowe środowisko wirtualne pythona. Nazwij je pyconpl-nlp:
   $ mkvirtualenv pyconpl-nlp
6. Zainstaluj wymagane pakiety z pliku requirements.pip:
   $ pip install -r requirements.pip

# 1. Korpusy tekstów

Zacznijmy od wyjaśnienia czym jest korpus tekstu. W lingwistyce korpusem tekstu nazywamy po prostu pewien zbiór danych (tekstów), który oprócz nazwy posiada swoją określoną charakterystykę, zawartość tematyczną, czasami dodatkową annotację itp. Możemy

np. mówić o korpusie tekstów z “Rzeczpospolitej”, a więc o zbiorze artykułów prasowych z tej gazety. Korpusy mogą być w bardzo różnych formatach i zawierać, bądź nie, informacje strukturyzowane, czy tzw. annotację i tagowanie. W najprostszym przypadku korpus to zwykły zbiór płaskich plików TXT zawierający czysty tekst. W przypadku artykułów prasowych w takim przypadku nie będzie wiadomo nawet, które elementy tego tekstu to tytuł, data czy autor. Budując np. wyszukiwarkę po takim korpusie tekstu, taka informacja mogła by być bardzo pomocna np. przy rankingowaniu wyników (np. dla hasła Kowalski ważniejsze są teksty, których autorem jest Kowalski, niż te gdzie tylko piszą o Kowalskim). Z tego powodu często dodaje się dodatkowe dane strukturyzujące, które można dodać albo np. za pomocą znaczników SGML (takich jak HTML czy XML) albo w inny własny, ustalony w korpusie sposób - np. w postaci pliku CSV, w których konkretne kolumny odpowiadają konkretnym informacjom. W tekście mogą pojawić się też znaczniki czysto lingwistyczne, takie jak koniec paragrafu, zdania, tagowanie wyrazów (np. część mowy: rzeczownik, przymiotnik, czasownik, …). Przykładem takiego bogatego korpusu dla języka polskiego jest Narodowy Korpus Języka Polskiego (http://www.nkjp.pl/) koordynowany przez IPI PAN. Jak widać korpusy potrafią być bardzo bogate w informacje, jednakże najczęściej mamy do czynienia po prostu z czystym tekstowym korpusem tekstów pozyskanych prosto ze źródła (np. ściągnięte fragmenty stron WWW). Korpusy tekstów mają jeszcze jedną bardzo ważną cechę - poziom zbalansowania. Jest to bardziej przydatne, gdy wykorzystujemy korpus do celów naukowych (statystycznych, porównawczych). Zależy nam wtedy na tym, aby teksty zebrane w danym korpusie statystycznie reprezentowały np. częstotliwość wyrazów używanych w danym języku, a więc obecność wszystkich wyrazów i ich odpowiednia częstość. Jeśli wyobrazimy sobie korpus poświęcony ściśle tylko i wyłącznie tekstom opisujących grę w piłkę nożną, możemy oczekiwać, że korpus taki jest mocno zbiasowany (niezbalansowany), to znaczy najczęściej występujące tam wyrażenia to najprawdopodobniej: sport, bramka, piłka, piłkarz, gol. Porównując to do charakterystyki ogólnej języka polskiego, wyrazy te nie będą tak częste - wręcz przeciwnie, mają one (pomimo dużej popularności tej gry) charakter specjalistyczny. Może także brakować reprezentacji bardzo wielu wyrazów jak np. parlament, okręt, itp. które mają niskie prawdopodobieństwo znalezienia się w takich tekstach. Wniosek z tego jest taki, że każdy korpus jest inny i zanim zaczniemy go przetwarzać warto znać jego charakterystykę, żeby móc tworzyć poprawne założenia dotyczące jego analizy.

## Korpus “telefony gsm”

Podczas warsztatów będziemy posługiwać się prostym korpusem zawierającym tytuły, ceny i stan przedmiotów (nowy, używany) zapisanych w postaci plików CSV. Posiadając taką bazę napisów z dodatkowymi annotacjami w postaci cen czy stany produktu możemy bez problemów spróbować napisać algorytm do wyceny produktów.

## Narzędzie csvkit

Pliki CSV wydają się bardzo proste do przetwarzania, bardzo często stosuje się do nich zwykłe polecenia Linuxa takie jak grep, cut czy sed. Jednak obsługa plików CSV w ten sposób może przysporzyć wiele trudnych do przewidzenia błędów mających źródło z dwóch faktów:

1. W plikach CSV dopuszcza się w przechowywanych wartościach używanie znaków stanowiących separatory - następuje wtedy umieszczanie treści w cudzysłowach.
2. Wartości mogą zawierać znaki końca linii, oznacza to że jeden logiczny wiersz pliku CSV może rozpięty być na wiele wierszy pliku tekstowego.

W praktyce oba przytoczone powody uniemożliwiają używanie bezpośrednio narzędzi typu grep, ponieważ pracują one linia po linii. Dużo lepszym pomysłem jest użycie dedykowanego do przetwarzania plików CSV narzędzia csvkit (http://csvkit.rtfd.org/), którego dodatkową zaletą jest fakt, iż jest on zaimplementowany w Pythonie i można przyłączyć się do jego rozwoju dopisując nowe funkcjonalności. Narzędzie csvkit dostarcza kilka narzędzi konsolowych, które w większości nawiązują nazewnictwem do linuxowych odpowiedników. Z ważniejszych poleceń wymienić należy **csvgrep** do filtrowania plików za pomocą wzorca lub wyrażenia regularnego, **csvcut** do wycinania kolumn z pliku, a także **csvsort** do sortowania po zadanej kolumnie oraz **csvlook** do łatwego podglądu plików.

Przykładowe użycie narzędzia csvkit łatwo przedstawić z użyciem korpusu:

$ csvlook gsm.csv

```
|-----------------------------------------------------+----------+--------|
|  name                                               | price    | state  |
|-----------------------------------------------------+----------+--------|
|  NOWY DOTYKOWY LG T375 Wi-Fi DUAL SIM + GRATISY FV  | 238.0    | new    |
|  NOWY SAMSUNG GALAXY NOTE 2 N7100 WHITE FV 23%      | 1799.0   | new    |
|  Samsung Galaxy S II S2 SII **gwarancja**PROMOCJA!! | 819.0    | used   |
|  ...
```

Wycinanie samej zawartości tekstowej z korpusu gsm:

$ csvcut -c name gsm.csv | tail -n +2

```
NOWY DOTYKOWY LG T375 Wi-Fi DUAL SIM + GRATISY FV
NOWY SAMSUNG GALAXY NOTE 2 N7100 WHITE FV 23%
Samsung Galaxy S II S2 SII **gwarancja**PROMOCJA!!
...
```

Odfiltrowanie tylko nowych produktów z korpusu gsm:

$ csvgrep -c state -m new gsm.csv | csvcut -c name,state | csvlook

```
|-----------------------------------------------------+--------|
|  name                                               | state  |
|-----------------------------------------------------+--------|
|  NOWY DOTYKOWY LG T375 Wi-Fi DUAL SIM + GRATISY FV  | new    |
|  NOWY SAMSUNG GALAXY NOTE 2 N7100 WHITE FV 23%      | new    |
|  NOKIA 3510 12miesięcy  GWARANCJI   Dostawa 24h     | new    |
|  ...
```

### Do wykonania

1. Zapoznaj się z komendami csvkit i/lub uruchom przykładowy skrypt prezentujące ich działanie:
   $ ./z01_corpus.sh

# 2. Słownik frekwencyjny - czyli częstotliwość wyrazów w korpusie

Najprostszym sposobem poznania charakterystyki korpusu jest zbudowanie słownika frekwencyjnego. Słownik frekwencyjny jest to lista unikalnych wyrazów z korpusu posortowana względem częstości występowania w danym korpusie.

## Tokenizacja

Żeby policzyć wyrazy w tekście należy najpierw je wyodrębnić z ciągłego tekstu. Taki proces nazywa się tokenizacją, a konkretnie w Pythonie będzie to operacja przeprowadzająca napis unicode w listę napisów unicode. Przykładowo:

u”Ala ubrała biało-czerwoną bluzkę, którą kupiła za 99 zł w P&P.”

W zależności od przyjętej strategii tokenizacji uzyskamy zupełnie inne tokeny.

1) Tokenizacja od spacji do spacji

```
>>> u"Ala ubrała biało-czerwoną bluzkę, którą kupiła za 99 zł w P&P.".split(' ')
[u'Ala', u'ubra\u0142a', u'bia\u0142o-czerwon\u0105', u'bluzk\u0119,', u'kt\xf3r\u0105',
u'kupi\u0142a', u'za', u'99', u'z\u0142', u'w', u'P&P.']
```

Zauważmy, że taka tokenizacja w przypadku normalnych zdań jest niezbyt korzystna, skleja nam interpunkcję z wyrazami uniemożliwiając ich rozpoznanie. Wyrazy wielosegmentowe są sklejone (biało-czerwony). Nazwa własna P&P prawidłowo rozpoznana.

2) Tokenizacja typu - znajdź wszystko co należy do wyrazu:

```
>>> re.findall(u'\w+', u"Ala ubrała biało-czerwoną bluzkę, którą kupiła za 99 zł w P&P." , re.U)
[u'Ala', u'ubra\u0142a', u'bia\u0142o', u'czerwon\u0105', u'bluzk\u0119', u'kt\xf3r\u0105',
u'kupi\u0142a', u'za', u'99', u'z\u0142', u'w', u'P', u'P']
```

Wydaje się, że druga tokenizacja podzieliła w tym przypadku tekst lepiej. Wyraz wielosegmentowy biało-czerwony rozbity na dwie jednostki (zwykle jest to korzystne). Jednakże taka tokenizacja wprowadziła inny problem. Nazwa własna P&P podzielona została na dwa segmenty P, P. Niestety zwykle takie sytuacje są nie do uniknięcia, musielibyśmy mieć słownik

nazw własnych i rozpoznać ją explicite, ponieważ znaku & do listy znaków dozwolonych jest złym pomysłem, ze względu na zbyt zachłanne sklejanie innych wyrazów.

To tylko dwa przykłady na prostą tokenizacje. W praktyce proces ten może być dużo bardziej złożony i np. wieloetapowy (związany z przeszukiwaniem słowników).

**Uwaga praktyczna:** bardzo często chcemy podczas tokenizowania jednocześnie sprawdzić napisy do zapisu małymi literami celem pozbycia się niepotrzebnych różnic pomiędzy nimi z tego wynikających. W Pythonie bardzo prosto wykonać to za pomocą metody lower() na napisie Unicode.

## Użycie słownika fleksyjnego

Podstawowym problemem z przetwarzaniem tekstu naturalnego jest fakt, że języki naturalne zwykle są fleksyjne. Oznacza to, że jeden **wyraz** jest reprezentowany w tekście może być przez wiele **form**. Na przykład można napisać “Nie mam psa”, w ten sposób użyć formy fleksyjnej “psa” wyrazu “pies”. Zliczenie ile razy wystąpił “pies” w tekście wymaga więc dodatkowej wiedzy, że napis “psa” jest jednym z potencjalnych określeń (form) znaczących tyle samo co “pies”. Inne języki (jak np. angielski) mają bardzo zredukowaną fleksję, np. rozróżnia ona tylko liczbę mnogą od pojedynczej “dogs”-> “dog”. Najprostszą operacją, którą możemy więc zrobić jest próba sprowadzenia dowolnej formy wyrazu znalezionej w tekście do formy podstawowej. To pozwala na znaczącą redukcję ilości różnych rozpoznanych napisów w tekście identyfikując całe wyrazy, a nie osobne formy.

Istnieją generalnie dwa podejścia do tego problemu: stemming oraz użycie słownika fleksyjnego. Nie będziemy tutaj zajmować się stemmingiem, ale warto wiedzieć czym jest stemming. Ogólnie stemmer jest to takie narzędzie, które posiada odpowiednie reguły umożliwiające obcięcie końcówki (najczęściej formy fleksyjnej) tak, aby powstał **stem** (o stemie możemy myśleć jak o formie podstawowej, a precyzyjniej mówiąc jak o wspólnym charakterystycznym początku danej grupy wyrazów). Np. używając reguły, aby obcinać ”s” z końca wyrazów w języku angielskim, możemy zamienić “computers” na “computer”. Reguły mogą być także silniejsze i np. sprowadzać różnego rodzaju wyrazy do wspólnego początku: “computing” -> “comput”, “computer” -> “comput”. Tym sposobem różne wyrazy kojarzące się z obliczeniami/komputerem sprowadzono do wspólnego początku “comput”, który od tej pory reprezentuje ich wystąpienie. Taka redukcja wyrazów do stemu jest trochę bliższa podejściu reprezentowania **pojęcia**.

Innym podejściem do sprowadzania formy fleksyjnej do podstawowej jest explicite posiadanie słownika fleksyjnego dla danego języka, który udostępnia wiedzę o wyrazach i potrafi je odmienić. Łatwym do pozyskania i darmowym do użytku (LGPL, MIT) jest słownik udostępniany przez SJP.pl. Możemy użyć go poprzez napisany do tego celu prosty moduł pydic (dokumentacja https://pydic.readthedocs.org/en/latest/PyDic.html, kod https://github.com/agh-glk/pydic, pydic jest już zainstalowany z pliku reqirements.pip).

Pobranie słownika ze strony http://sjp.pl/slownik/odmiany/

```
$ wget http://sjp.pl/slownik/odmiany/sjp-odm-20130802.zip
$ unzip sjp-odm-*.zip
```

Po ściągnięciu i rozpakowaniu mamy plik odm.txt. Użyjemy teraz narzędzia z pakietu pydic do wyczyszczenia tego pliku i zbudowania słownika w formacie pydic.

Przygotowanie odpowiedniego formatu pliku TXT:

```
$ cat odm.txt | sjp2pydic.py > sjp.txt
```

Budowa słownika:

```
$ pydic_create.py -v -f sjp.txt
```

W wyniku tej operacji w bieżącym katalogu utworzony zostanie katalog sjp.pydic, który reprezentuje słownik fleksyjny. Przetestujmy jego działanie w konsoli Pythona:

```
$ python
>>> from pydic import PyDic
>>> sjp = PyDic('sjp.pydic')
```

identyfikujemy wyraz na podstawie formy:

```
>>> sjp.id(u'telefonował')
[PyDicId(u'163327@sjp')]
```

znajdujemy formę bazową na podstawie formy:

```
>>> sjp.word_base(u'telefonował')
[u'telefonowa\u0107']
>>> print sjp.word_base(u'telefonował')[0]
telefonować
```

Możesz także spotkać się w tym tekście z pojęciem **leksem.** Przez leksem rozumie się po prostu konkretny wyraz stanowiący jednostkę słownika, a więc najczęściej mający swój identyfikator w tym słowniku oraz wektor odmiany. Np w SJP leksem “telefonować” ma ID 163327@sjp oraz następujący wektor odmiany:

```
>>> sjp.id_forms(u'163327@sjp')
[u'telefonowa\u0107', u'telefonowali', u'telefonowaliby', u'telefonowaliby\u015bcie',
u'telefonowaliby\u015bmy', u'telefonowali\u015bcie', u'telefonowali\u015bmy',
u'telefonowa\u0142', u'telefonowa\u0142a', … ]
```

W ten sposób słownik ten rozpozna wszystkie z w/w form i będzie umiał powiązać je z formą podstawową telefonować.

Co da nam użycie słownika fleksyjnego w praktyce? Zapoznaj się z tytułami dwóch przykładowych aukcji:
1) “Telefony Nokii dla seniora z gwarancją.”
po tokenizacji, zmiana wielkości liter: telefony, nokii, dla, seniora, z, gwarancją
2) “Telefon Nokia. Wersja senior. Gwarancja!”
po tokenizacji, zmiana wielkość liter: telefon, nokia, wersja, senior, gwarancja

Liczba wspólnych tokenów w przypadku obu list wynosi 0. Gdyby jednak użyć słownik fleksyjny i sprowadzić wszystkie wyrazy do formy podstawowej:
1) telefon, nokia, dla, senior, z, gwarancja
2) telefon, nokia, wersja, senior, gwarancja
okazuje się, że teksty te mają aż 4 wspólne tokeny. Uzyskujemy więc ogromny zysk jakościowy przy porównaniach wyrazów.

## Do wykonania

1. Przygotuj słownik sjp w formacie pydic (jeśli jeszcze nie wykonałeś/aś tego czytając opis powyzej):
   $ cat odm.txt | sjp2pydic.py > sjp.txt
   $ pydic_create.py -v -f sjp.txt
2. W pliku z02_frequence.py uzupełnij funkcję tokenize_line(line). Funkcja powinna podzielić każdą linię na listę wyrazów. Uzyj do tego generatora, który będzie generował kolejne wyrazy z linii - oszczędzając w ten sposób pamięć.
3. Podczas tokenizacji sprowadź napisy do zapisu małymi literami - metoda lower().
4. Sprowadź tokeny do formy podstawowej wyrazu, zrób to w ten sposób:
   a. użyj pydic, metoda word_base(unicode_string); metoda ta zwróci listę potencjalnych form podstawowych dla podanej formy,
   b. odfiltruj wszystkie wyrazy z tej listy, które zaczynają się z wielkiej litery,
   c. z pozostałej listy wybierz jako formę bazową zawsze najkrótszy wyraz,
   d. jeśli lista jest pusta, użyj tokenu jako formy podstawowej.

5. Testuj program uruchamiając go w ten sposób:
   $ csvcut -c name gsm.csv | python z02_frequence.py

   Otrzymasz tego typu listę frekwencyjną:

   14026 nowy
   11876 nokia
   8931 samsung
   8747 bez

…

6. Zapisz uzyskane wyniki do pliku gsm.freq
   $ csvcut -c name gsm.csv | python z01_frequence.py > gsm.freq

# 3. Prawo Zipfa (i uogólnienie Mandelbrota)

George Kingsley Zipf w swojej książce "Human Behavior and the Principle of Least Effort" zamieścił prawo, które można by określić mianem zasady Pareto w lingwistyce. Postulował on, że częstotliwość występowania wyrazu w tekście jest odwrotnie proporcjonalna do numeru rankingu powstałego przez uporządkowanie wyrazów względem ich częstości występowania. Przykład:

| | | | |
|---|---|---|---|
| częstość | 205 | 115 | 56 |
| wyraz | kot | pies | mysz |
| ranking | 1 | 2 | 3 |

Inaczej mówiąc, wyraz na 50. pozycji w rankingu będzie występował trzykrotnie częściej niż wyraz na pozycji 150. Zatem jeśli f określa częstotliwość, a r pozycję w rankingu, to winna istnieć stała k taka, że:

$$f \approx \frac{k}{r}$$

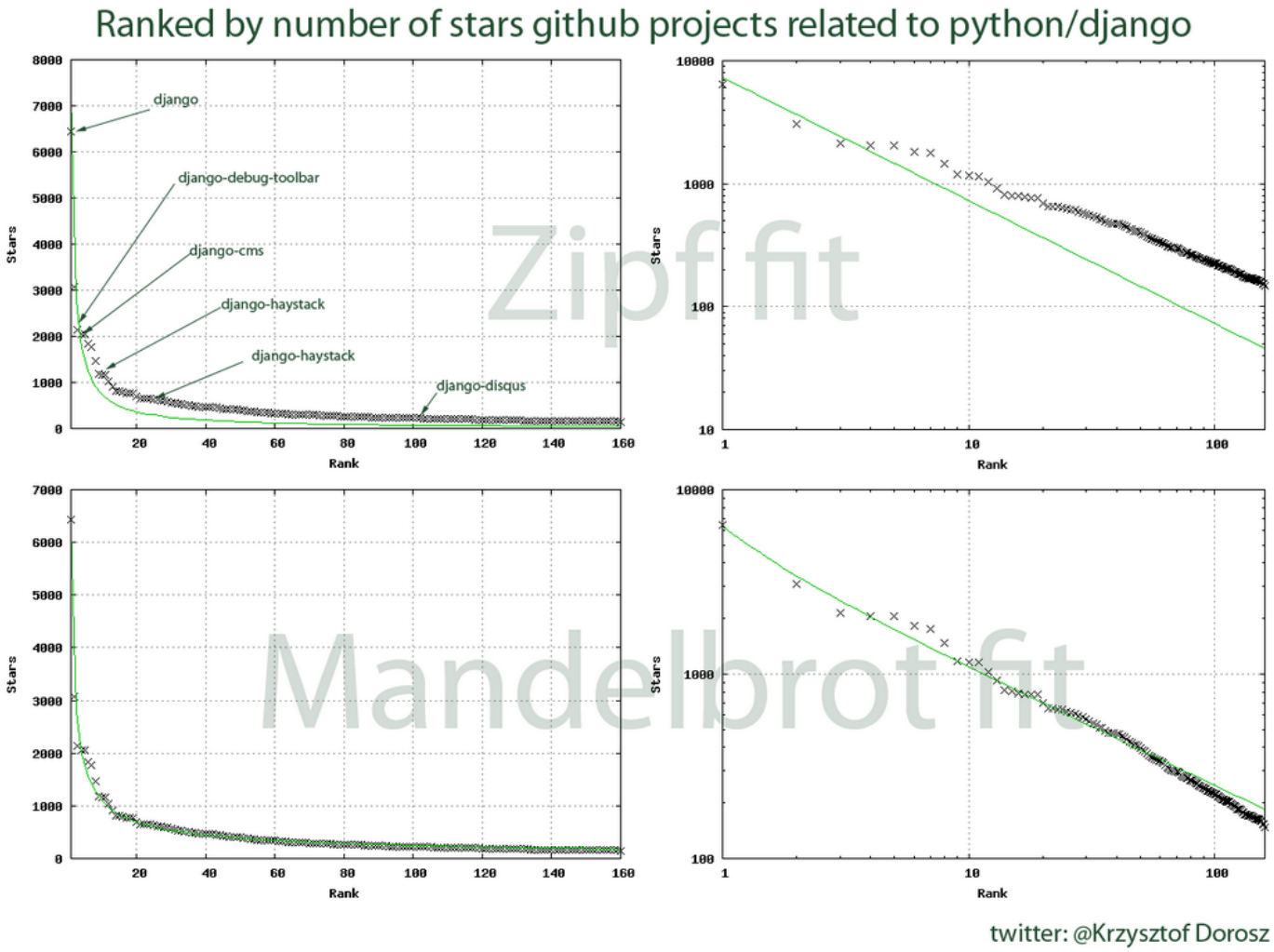
Prawo Zipfa oddaje pewien charakter statystyczny wielu problemów związanych z modelowaniem zachowań ludzkich (np. działa także w stosunku do ilości i wielkości miast wyrażonej w liczbie mieszkańców, i wiele innych...) natomiast nie jest możliwe precyzyjne odwzorowanie na całej dziedzinie problemu. Mandelbrot podał uszczegółowienie tego prawa wyrażając go za pomocą modelu relacji opartego na funkcji wykładniczej.

Dokładniej, dla pewnych stałych B, d, P relacja między r a f wynosi:

$$\log(f) = \log(P) - B \cdot \log(r + d)$$

Prawo to działa w bardzo wielu aspektach związanych z ludzkim profilem zachowania. Gdyby w podobny sposób ułożyć miasta posortowane pod względem ilości mieszkańców, ich wykres byłby zbieżny do Prawa Zipfa. Okazuje się, że prawo te spełniają nawet najbardziej wymyślne systemy związane z zachowaniem ludzi, np. rozkład projektów Django na githubie względem przydzielonych im gwiazdek także spełnia prawo Zipfa!

Z prawa Zipfa płynie jeszcze jedna nauka - zwykle najczęściej używamy tylko niewielkiego zasobu słownictwa. Wyrazy najczęściej stosowane niosą małą wartość informatywną - ponieważ najprawdopodobniej znajdziemy je w większości tekstów. Nie dyskryminują więc one w znaczący sposób te teksty. Okazuje się, że najwięcej informacji niosą te wyrazy, które umieszczone są w ogonie wykresu krzywej Zipfa.




## Stop lista

Bardzo często stosuje się więc **stop listę**, stanowiącą zbiór wyrazów z głowy wykresu krzywej Zipfa, żeby nie zaburzały one analizy pozostałych bardziej informatywnych wyrazów. Sytuacja ta ulega trochę zmianie w naszym zbiasowanym korpusie tekstu. Tam w najbardziej popularnych wyrazach znajdziemy nazwy marek telefonów, które akurat są nam bardzo potrzebne do oceny wartości. Możemy za to spróbować odrzucić inne wyrazy, które są częste a nie wydają się istotne na rozpoznanie produktu (np.  faktura, vat, itp).

## Do wykonania

1. Wygeneruj wykresy krzywej Zipfa i Mandelbrota dla korpusu gsm w następujący sposób:
   $ ./z03_zipf.sh
   skrypt sam użyje poprzednio wygenerowanego pliku gsm.freq
2. Zapoznaj się z wygenerowanymi wykresami znajdującymi się po wygenerowaniu w plikach z03_*.png.
3. Zbuduj stop listę w pliku gsm.stop szybko przeglądając początek pliku gsm.freq ręcznie. Umieść na niej linia po linii w kodowaniu UTF-8 wszystkie wyrazy, które mają charakter czysto marketingowy, a nie opisują rodzaj/cechę produktu wpływającego na cenę: np. faktura, fv, czarny, biały, sklep, nowość, nowy, szybko, okazja, zadbany, kurier, 24h itp.

# 4. Metryki odległości tekstu

Do realizacji zadania na warsztatach potrzebujemy mechanizmu, który potrafił będzie wskazać jak podobne są do siebie dwa napisy. Podobieństwo możemy wyznaczać zależnie od różnych aspektów, stad będziemy mówić, że używamy konkretnych metryk w kontekście których mówimy o podobieństwie (odległości) napisów.

## Odległość edycyjna

Najbardziej znaną i powszechnie używaną metryką podobieństwa jest odległość Levenshteina, zwana także odległością edycyjną. Odległość w tej metryce wyrażana jest za pomocą liczby naturalnej. Jej intuicyjna interpretacja jest taka, że jest to minimalna ilość operacji takich jak usunięcie, dołożenie lub nadpisanie znaku, aby przeprowadzić jeden napis w drugi. Na przykład odległość edycyjna dwóch napisów:
A) ROMEK
B) ATOMEK
wynosi 2 ponieważ do napisu ROMEK wystarczy dodać literę A (na początku), uzyskując AROMEK, a następnie podmienić literę R na T, uzyskując finalnie ATOMEK.
Dużą wadą tej metody jest niska odporność na przestawienia wyrazów w tekście. Przykładowo:
A) ZEGAREK TISSOT
B) TISSOT ZEGAREK
Odległość edycyjna wynosi aż 14, a czasami chcielibyśmy (np gdy porównujemy tytuły aukcji), aby takie dwa napisy były dla nas tożsame (bo przekazują tą samą treść). Można np. przed wyliczaniem odległości edycyjnej posortować wyrazy alfabetycznie, tak aby zniwelować ten efekt.

## LCS - Longest common substring

Stosunkowo prostą metryką jest Longest Common Substring, która porównuje dwa napisy A i B na podstawie długości ich najdłuższego wspólnego podciągu. Dla f(A,B) - funkcji zwracającej najdłuższy wspólny podciąg napisów A i B, |A| - długości napisu A to w/w metrykę zdefiniujemy jako:

$$\mathrm{LCS}(\mathrm{A},\mathrm{B}) = 1 - \frac{|\mathrm{f}(\mathrm{A},\mathrm{B})|}{\max(|\mathrm{A}|,|\mathrm{B}|)}$$

Metryka powyższa jest podobnie jak odległość edycyjna dla pewnych zastosowań zbyt wrażliwa na przestawienia elementów napisu - np. jeśli w adresie przestawimy miejscami miejscowość z ulicą - to nadal będzie to ten sam adres, ale metryka LCS pokaże nam znaczną zmianę.

## Współczynnik Dice'a

Metoda ta pozwala uniknąć problemu wrażliwości na kolejność występowania wyrazów w tekście. Bada ona bowiem charakterystykę zbitek literowych - zwanych n-gramami. N-gram jest to po prostu dowolny podciąg N znaków z tekstu. Przykładowo dla tekstu "ATOMEK" można wypisać następujące unikalne 3-gramy: ATO, TOM, OME, MEK. Jeśli rozbijemy napisy A i B na odpowiadające im zbiory n-gramów Ngrams(A) i Ngrams(B) to możemy porównać te zbiory przy pomocy tzw. współczynnika Dice'a:

$$\mathrm{DICE}(\mathrm{A},\mathrm{B}) = 1 - \frac{2 \cdot |\mathrm{Ngrams}(\mathrm{A}) \cap \mathrm{Ngrams}(\mathrm{B})|}{|\mathrm{Ngrams}(\mathrm{A})| + |\mathrm{Ngrams}(\mathrm{B})|}$$

Innymi słowy liczymy ile unikalnych n-gramów jest w napisie A, ile w B, ile jest takich n-gramów że znajdują się zarówno w A i B, a potem podstawiamy do zbioru. Dzięki takiej operacji nie ma dużego znaczenia gdzie leży dany wyraz w tekście, przy odpowiednio małej wielkości n-gramu wygenerują się takie same n-gramy. Niestety metoda ta ma także swoje słabości, np. przy użyciu 2-gramu takie teksty są idealnie podobne:
A) aaaaaaaaabbaaaaaaaaaaa
B) bbbbbbbbbaabbbbbbbbbbb
wygenerują łącznie zbiór identycznych 2-gramów: aa, ab, ba, bb

## Do wykonania

1. Zapoznaj się z plikiem z4_distance.py. Program ten wymaga podania 3 parametrów:
    a. -m - rodzaj funkcji podobieństwa: edit, lcs lub dice
    b. -t - próg od, którego uznajemy podobieństwo, w przypadku opcji edit, próg może być od 0 (idealnie podobne) do dowolnej dodatniej liczby, w przypadku dwóch pozostałych musi to być liczba z przedziału od 0 (nie podobny) do 1 (idealnie podobny). Próg będziemy musieli próbować wyznaczać empirycznie i testować różne ustawienia zależnie od algorytmów przygotowania danych do porównania
    c. -n - nazwa towaru dla którego szukamy podobieństw
2. Program działa w ten sposób, że stara się odnaleźć w korpusie wszystkie podobne przedmioty zgodnie z wybraną metryką. Program wyświetli je, a potem zsumuje i uśredni ich cenę. Wszystkie metryki są już zaimplementowane, jednak będziemy musieli

zaimplementować funkcję przygotowujące napisy do porównania, ponieważ takie zwykłe podejście jak szybko zobaczysz nie daje dobrych rezultatów.

3. Twoim zadaniem jest edycja funkcji prepare_for_levenshtein(), prepare_for_dice(), prepare_for_lcs(). Funkcje te przetwarzają wstępnie dwa napisy - nazwę aukcji wzorcowej i nazwę aukcji porównywanej w danej chwili). Chodzi o to aby tak przeformatować te napisy, aby jak najlepiej wykorzystać właściwości metryk podobieństwa.
4. Możesz spróbować użyć wielu metryk - albo skupić się tylko na jednej, implementując wskazówki poniżej.
5. prepare_for_levensthein i prepare_for_lcs:
    a. domyślnie funkcja wykonuje jedynie operacje lower() na tekście, oprócz tego w żaden sposób go nie optymalizuje,
    b. spróbuj stokenizować funkcją z z02 tekst, posortować wyrazy alfabetycznie i zwrócić jako napis rozdzielony spacjami.
    c. spróbuj użyć stoplisty, odfiltruj wyrazy które się na niej znajdują
    d. przetestuj różne wagi przy filtrowaniu, dla różnych przykładowych tytułów aukcji
6. prepare_for_dice:
    a. domyślnie funkcja zwraca pythonowy zbiór (set) wyrazów, uzyskanych z Twojej funkcji tokenizującej z z02.
    b. spróbuj użyć stoplisty, odfiltruj wyrazy które się na niej znajdują
    c. spróbuj zamiast wyrazów użyć n-gramów, musisz w tym celu wyliczyć dla każdego napisu zbiór unikalnych podciągów o długości n (przyjmij n=2, 3 lub 4)
7. Pomocne przy pracy programu będzie używanie opcji --verbose (-v).
8. Program uruchamiaj np. tak:

    $ cat gsm.csv | python z04_distance.py -m edit -t 3 -n "NOWY SAMSUNG GALAXY NOTE 2 N7100 WHITE FV 23%" -v
9. Możesz także wcześniej posegmentować swoją bazę na produkty nowe i używane, aby uzyskać bardziej precyzyjną wycenę!

    $ csvgrep -c state -m new gsm.csv > gsm.new.csv
    oraz
    $ csvgrep -c state -m used gsm.csv > gsm.used.csv

10. Powodzenia!