Pythonで自然言語処理入門

本日の内容

- NLP基本的な概念の紹介
- 日本語ウィキペディアを処理する実践的な体験
  - 3つの短い実習問題（10分程度）
- 実習問題

NLPとは

自然言語処理(NLP)とは、自然言語の形をした情報を処理する・生成するための手法を研究するコンピュータサイエンスの分野です。言い換えると、NLPの目的はプログラムに人間の言葉で表された情報を理解させることになります。

単語分割 / トークン化

トークナイゼーションとは、与えられたテキストを "トークン" と呼ばれる小さい部分に分けることです。通常、これはテキストを単語トークンに分割することを意味しますが、解析したい内容によっては文や形態素(morphemes)などのような他の単位でトークナイゼーションすることもあります。

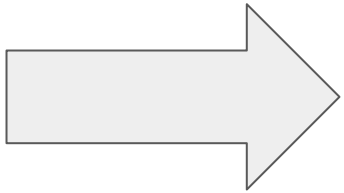
- "**I like to do NLP**"
  => ["I", "like", "to", "do", "NLP"]

- "**単語分割はとても大切です**"
  => ["単語分割", "は", "とても", "大切", "です"]

正規化 (Normalization)

自然言語とその表記体系の特徴として、不規則であることと、同じ概念を異なる方法で表現することが可能であるということが挙げられます。

- 東京大学
- とうきょうだいがく
- トウキョウダイガク
- ﾄｳｷｮｳﾀﾞｲｶﾞｸ
- Tokyo Daigaku
- Toukyou Daigaku
- Tōkyō Daigaku
- tokio daigaku
- etc



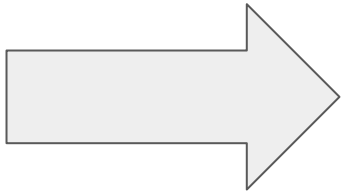
東京大学

こういった単語の複数のバリエーションは全て同じトークンだとみなしたい場合が多い。

正規化 (Normalization): 見出し語変換 (Lemmatization)

自然言語とその表記体系の特徴として、不規則であることと、同じ概念を異なる方法で表現することが可能であるということが挙げられます。

- 食べながら
- 食べた
- 食べさせた
- 食べさせられた
- 食べられた
- 食べている
- 食べましょう



食べる

こういった単語の複数のバリエーションは全て同じトークンだとみなしたい場合が多い。

形態素解析 / Part of Speech Tagging (POS)

文を単語に分割することができたら、各単語の文中での役割を理解することは有用です。POS (Part of Speech) とは各トークンやトークン列に、文中での文法的な役割を表すタグのことです。そのタグ付けする処理をPOS Taggingと言います。日本語では形態素解析(Syntactic Analysis)とも言います。

[名詞]	[助詞]	[副詞]	[形容詞]	[助動詞]	[記号]
NLP	は	とても	楽しい	です	。

MeCab: 日本語形態素解析ツール

Mecabは形態素解析エンジンと呼ばれ、日本語テキストをトークナイゼーションをし、POSタグを与え、見出し語に変換を行います。Mecabにおけるトークンは単語とだいたい等価ですが、より細かい単位にも分割処理を行います。

'NLPはとても楽しいです。'

```
NLP	名詞,固有名詞,組織,*,*,*,*
は	助詞,係助詞,*,*,*,*,は,ハ,ワ
とても	副詞,助詞類接続,*,*,*,*,とても,トテモ,トテモ
楽しい	形容詞,自立,*,*,形容詞・イ段,基本形,楽しい,タノシイ,タノシイ
です	助動詞,*,*,*,特殊・デス,基本形,です,デス,デス
。	記号,句点,*,*,*,*,。,。,。
```

'NLPはとても楽しいです。'

```
NLP     名詞,固有名詞,組織,*,*,*,*
は      助詞,係助詞,*,*,*,*,は,ハ,ワ
とても  副詞,助詞類接続,*,*,*,*,とても,トテモ,トテモ
楽しい  形容詞,自立,*,*,形容詞・イ段,基本形,楽しい,タノシイ,タノシイ
です    助動詞,*,*,*,特殊・デス,基本形,です,デス,デス
。      記号,句点,*,*,*,*,。,。,。
```

['NLP', 'は', 'とても', '楽しい', 'です', '。']

{ 表層形、POS(品詞情報)、POS1、POS2、POS3、活用型、活用形、原形(Lemma)、読み、発音 }

'食べさせてもらえますか?'

食べ	動詞,自立,*,*,一段,未然形,食べる,タベ,タベ
させ	動詞,接尾,*,*,一段,連用形,させる,サセ,サセ
て	助詞,接続助詞,*,*,*,*,て,テ,テ
もらえ	動詞,非自立,*,*,一段,連用形,もらえる,モラエ,モラエ
ます	助動詞,*,*,*,特殊・マス,基本形,ます,マス,マス
か	助詞,副助詞／並立助詞／終助詞,*,*,*,*,か,カ,カ
?	名詞,サ変接続,*,*,*,*,*

食べる　+させる　+て　+もらえる　+ます

'食べさせてもらえますか?'

食べ	動詞,自立,*,*,一段,未然形,食べる,タベ,タベ
させ	動詞,接尾,*,*,一段,連用形,させる,サセ,サセ
て	助詞,接続助詞,*,*,*,*,て,テ,テ
もらえ	動詞,非自立,*,*,一段,連用形,もらえる,モラエ,モラエ
ます	助動詞,*,*,*,特殊・マス,基本形,ます,マス,マス
か	助詞,副助詞／並立助詞／終助詞,*,*,*,*,か,カ,カ
?	名詞,サ変接続,*,*,*,*,*

食べる　+させる　+て　+もらえる　+ます

ハンズオン: 日本語Wikipedia

ハンズオン： 日本語Wikipedia

事前に抽出したウィキペディアのテキストファイル

```
<doc id="5" url="https://ja.wikipedia.org/wiki?curid=5" title="アンパサンド">
アンパサンド

アンパサンド (, &) とは「…と…」を意味する記号である。ラテン語の  の合字で、Trebuchet MS フォントでは、
と表示され "et" の合字であることが容易にわかる。 ampersa、すなわち "and per se and"、その意味は"and [the symbol which] by itself [is] and" である。
その使用は1世紀に遡ることができ (1)、5世紀中葉 (2,3) から現代 (4-6) に至るまでの変遷がわかる。
Z に続くラテン文字アルファベットの 27字目とされた時期もある。
アンパサンドと同じ役割を果たす文字に「の et」と呼ばれる、数字の「 7」に似た記号があった (, U+204A)。この記号は現在もゲール文字で使われている。
記号名の「アンパサンド」は、ラテン語まじりの英語「 & はそれ自身 "and" を表す」(& per se and) のくずれた形である。英語以外の言語での名称は多様である。
日常的な手書きの場合、欧米でアンパサンドは「 ε」に縦線を引く単純化されたものが使われることがある。
また同様に、「t」または「+（プラス）」に輪を重ねたような、無声歯茎側面摩擦音を示す発音記号「」のようなものが使われることもある。
プログラミング言語では、 C など多数の言語で AND 演算子として用いられる。以下は C の例。
PHPでは、変数宣言記号（ $）の直前に記述することで、参照渡しを行うことができる。
BASIC 系列の言語では文字列の連結演算子として使用される。 codice_4 は codice_5 を返す。また、主にマイクロソフト系では整数の十六進表記に
 codice_6 を用い、codice_7 （十進で15）のように表現する。
SGML、XML、HTMLでは、アンパサンドを使って SGML実体を参照する。

</doc>
<doc id="10" url="https://ja.wikipedia.org/wiki?curid=10" title="言語">
言語

この記事では言語（げんご）、特に自然言語について述べる。
・・・
```

ハンズオン： 日本語Wikipedia

事前に抽出したウィキペディアのテキストファイル

<doc id="{記事ID}" url="{記事のURL}" title="{記事タイトル}">

記事の本文

</doc>

<doc id="{記事ID}" url="{記事のURL}" title="{記事タイトル}">

記事の本文

</doc>

...

単語出現頻度

自然言語のデータセットに対する最も簡単な解析：

単語出現頻度(Word frequencies)

単語出現頻度を計算することはNLPにおいてとても重要です。多くの事柄を明らかにしますし、多くのNLPのテクニックは単語出現頻度を基にしています。

また、単語出現頻度をみることで、単語分割や正規化がうまくできているかを確認することができます。NLPにおいて、正規化と単語出現頻度を行ったりしながら、データ中のノイズを減らすということはよくあります。

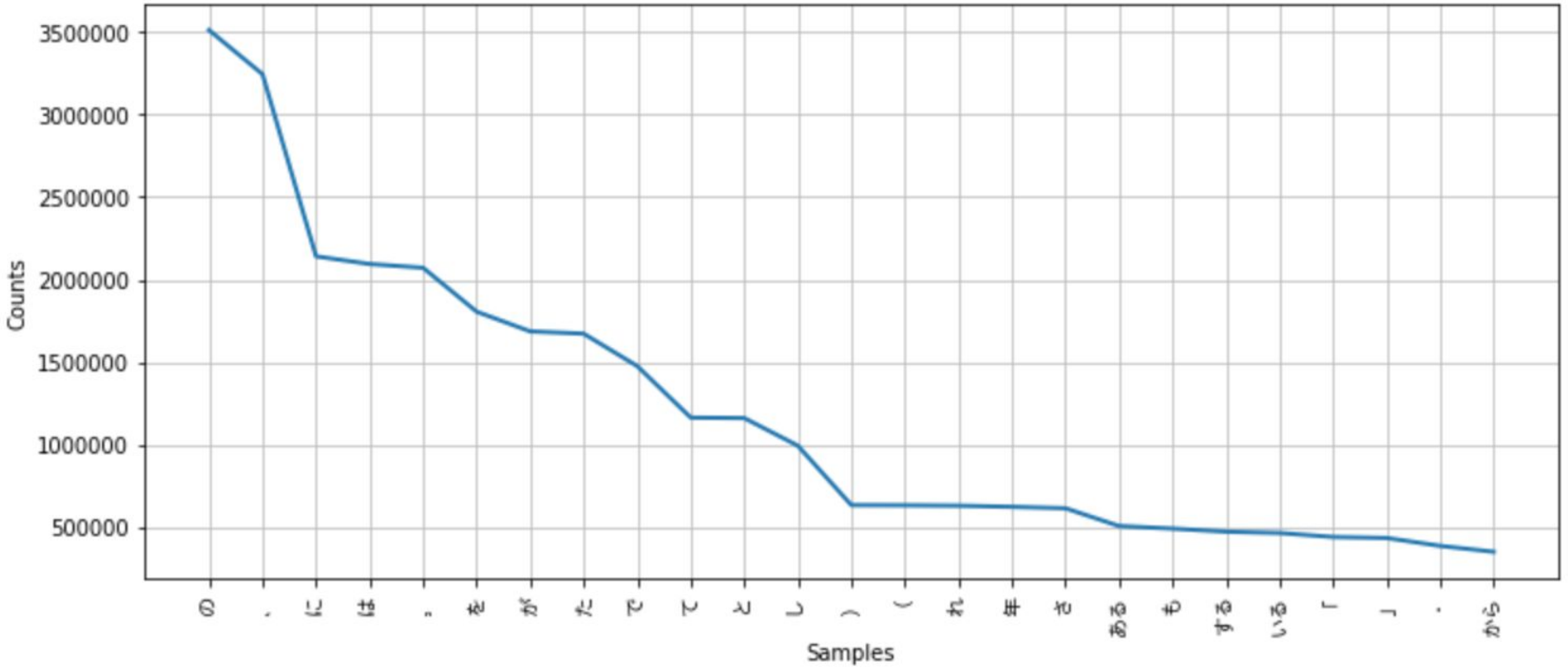
単語出現頻度：計算

```
def count_all_word_frequencies():
  all_words = []
  tagger = MeCab.Tagger()
  with codecs.open("japanese_wikipedia_extracted_articles.txt", "r",'utf-8') as file:
    for line in file:
        node = tagger.parseToNode(line)
        while(node):
          if node.surface != "":
            all_words.append(node.surface.lower())
          node = node.next
          if node is None:
              break
  return Counter(all_words)
```

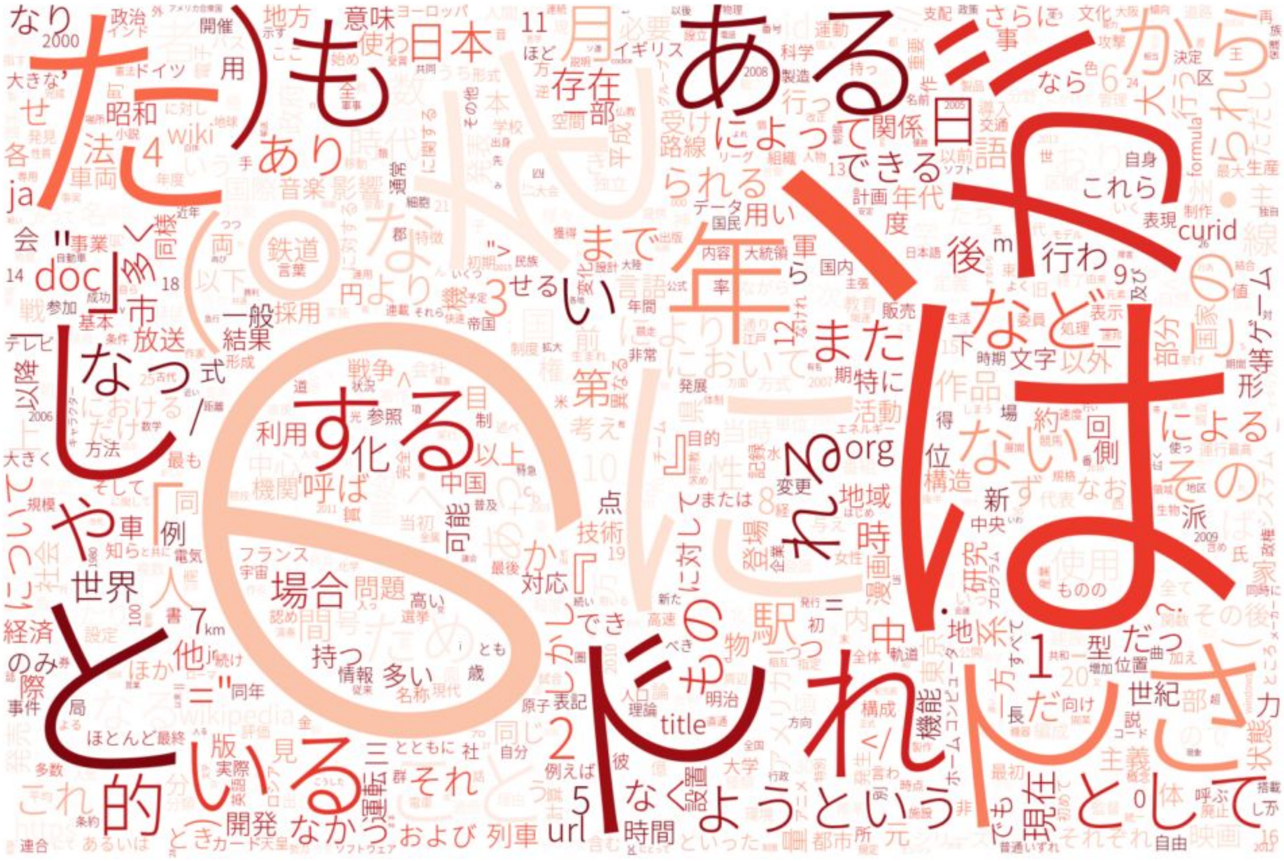
単語出現頻度：計算

```
Word: の, frequency: 3512545
Word: 、, frequency: 3245139
Word: に, frequency: 2141039
Word: は, frequency: 2094666
Word: 。, frequency: 2071763
Word: を, frequency: 1805639
Word: が, frequency: 1686345
Word: た, frequency: 1672139
Word: で, frequency: 1475972
Word: て, frequency: 1163438
Word: と, frequency: 1160679
Word: し, frequency: 993385
Word: ）, frequency: 633112
Word: （, frequency: 631910
Word: れ, frequency: 629469
Word: 年, frequency: 622716
Word: さ, frequency: 613974
Word: ある, frequency: 506361
Word: も, frequency: 490642
Word: する, frequency: 472102
Word: いる, frequency: 464001
```

単語出現頻度：可視化


3500000
3000000
2500000
2000000
1500000
1000000
500000
Counts
の
、
に
は
。
を
が
た
で
て
と
し
）
（
れ
年
そ
ある
も
する
いる
「
」
・
から
Samples

単語出現頻度：可視化



の
は
、
が
た
し
と
で
に
を
て
れ
ある
も
する
年
いる
的
など
また
こと
から
として
という
よう
なっ
ない
さ
れる
ありなる
日本
月
存在
によって
関係
できる
られる
まで
第
場合
世界
特に
作品
使用
現在
時間
列車
開発
鉄道

第１実習問題（１０分）：単語出現頻度・正規化後

以下の点を変更して、もう一度やってみましょう:

- 見出し語のみをカウントする
- 名詞・動詞・形容詞のみをカウントする
- よくでてくるが今回の目的には関連しない語は、
  ブラックリストを作って除外する

単語出現頻度：計算

```
def count_word_frequencies(text, words_blacklist, categories_whitelist, categories_blacklist):
  all_nouns_verbs_adjs = []
  tagger = MeCab.Tagger()
  for line in text:
      node = tagger.parseToNode(line)
      # define a function that given a word returns the lemma and pos, pos2
      while(node):
        lemma, pos, pos2 = get_word_lemma_and_pos_info(node)
        if lemma != '-' and lemma not in words_blacklist and
          pos in categories_whitelist and pos2 not in categories_blacklist:
            all_nouns_verbs_adjs.append(lemma)
        node = node.next
        if node is None:
            break
  return Counter(all_nouns_verbs_adjs)
```

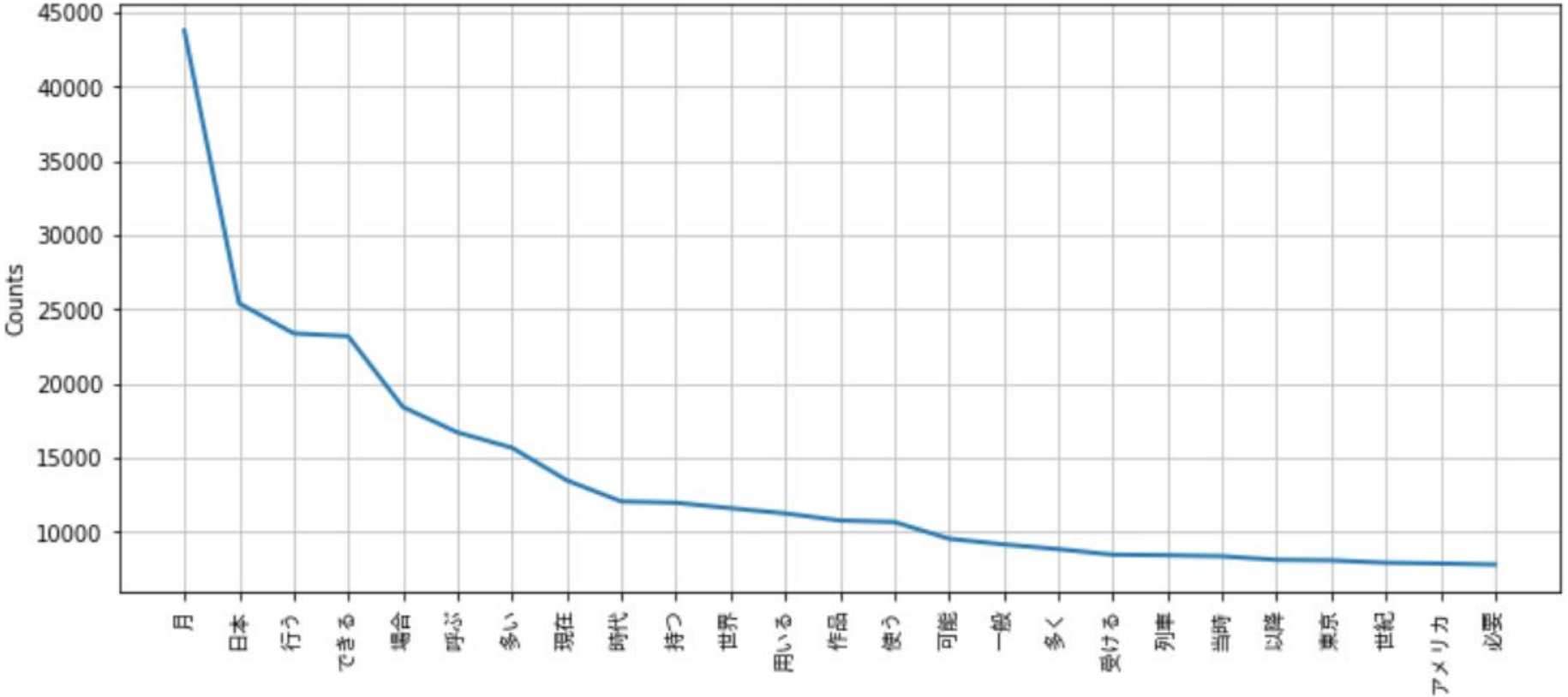
単語出現頻度：計算

```
words_blacklist = ['する', 'なる', 'ない', 'これ', 'それ', 'id', 'ja', 'wiki',
                   'wikipedia', 'id', 'doc', 'https', 'org', 'url', 'いう', 'ある',
                   'curid', 'あれ', 'それら', 'これら', 'それそれ', 'それぞれ',
                   'title', 'その後', '一部', '前', 'よる', '一つ', 'ひとつ', '他',
                   'その他', 'ほか', 'そのほか', 'いる']
whitelist = ['名詞', '動詞', '形容詞']
blacklist = ['非自立', '接尾', 'サ変接続', '数']
with codecs.open("japanese_wikipedia_extracted_articles.txt", "r",'utf-8') as text:
  all_nouns_verbs_adjs = count_word_frequencies(text, words_blacklist, whitelist, blacklist)
for word in all_nouns_verbs_adjs.most_common(25):
  print("Word: {}, frequency: {}".format(word[0], word[1]))
```

単語出現頻度：計算

```
Word: 月, frequency: 43788
Word: 日本, frequency: 25412
Word: 行う, frequency: 23382
Word: できる, frequency: 23191
Word: 場合, frequency: 18425
Word: 呼ぶ, frequency: 16707
Word: 多い, frequency: 15677
Word: 現在, frequency: 13484
Word: 時代, frequency: 12057
Word: 持つ, frequency: 11969
Word: 世界, frequency: 11601
Word: 用いる, frequency: 11252
Word: 作品, frequency: 10774
Word: 使う, frequency: 10669
Word: 可能, frequency: 9547
Word: 一般, frequency: 9162
Word: 多く, frequency: 8844
Word: 受ける, frequency: 8469
Word: 列車, frequency: 8426
Word: 当時, frequency: 8368
Word: 以降, frequency: 8121
Word: 東京, frequency: 8088
```

単語出現頻度：可視化


Counts
45000
40000
35000
30000
25000
20000
15000
10000
月
日本
行う
できる
場合
呼ぶ
多い
現在
時代
持つ
世界
用いる
作品
使う
可能
一般
多く
受ける
列車
当時
以降
東京
世紀
アメリカ
必要

単語出現頻度：可視化

日本
電車
できる
本
呼ぶ
場合
世紀
列車
多い
時代
使う
現在
作品
当時
用いる
可能
昭和
地方
問題
含む
国際
平成
鉄道
必要
社会
多く
路線
シリーズ
都市
機関
同様
以降
言う
構造
状態
言語
文字
政府
名称
重要
映画
漫画
異なる
見る
知る
方法
方式
彼
事業
電子
対象
大統領
時期
最終
認める
内容
番組
基本
会社
人間
少ない
科学
フランス
地域
大きい
イギリス
委員
特徴
高速
以前
要素
規模
名前
自身
formula
歴史
複数
求める
標準
単位
目的
ホーム
システム
通常
続く
地球
宇宙
体制
学校
帝国
概念
述べる
挙げる
続く
国民
国内
人口
民族
自然
区間
分野
憲法
協会
自分
軌道
首都
京都
電気
日共和
仏教
朝
環境
機器
物質
描く
伴う
大阪
経済
江戸
エネルギー
ソフトウェア
基づく

前処理済みのデータベース：インターフェースの定義

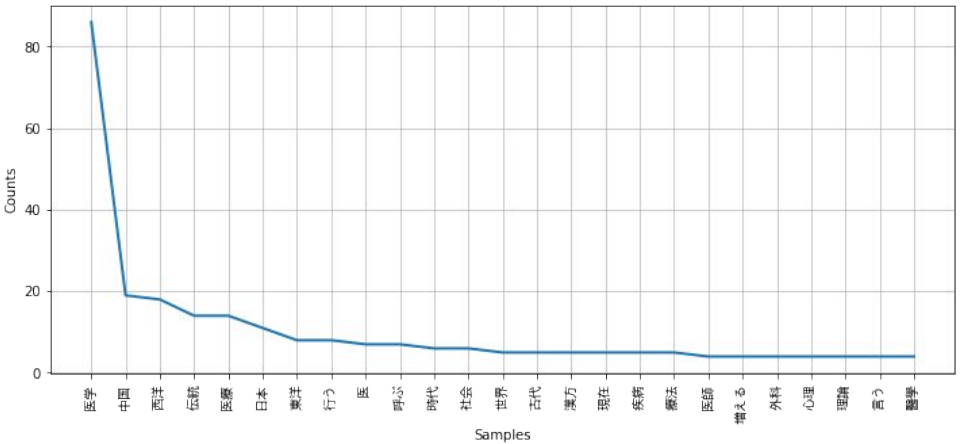
- retrieve_random_articles(db_path, amount)
- retrieve_articles_wordfreqs_by_title(db_path, article_title)
- retrieve_wikipedia_wordfreqs(db_path)
- retrieve_wikipedia_articles_amount(db_path)

ウィキペディア記事ごとの解析

```
title = '医学'
# 「医学」のウィキペディア記事の単語出現の分布
article_word_freqs = dict(retrieve_articles_wordfreqs_by_title(DB_PATH, title))
plot_word_frequency_distribution(article_word_freqs)
# ワードクラウドを作ってみましょう
print(title)
make_word_cloud(article_word_freqs)
```

ウィキペディア記事単位の解析：「医学」

単語出現回数



単語出現回数の分布

単語の重要度：TF-IDF

ある単語がたくさん現れるということだけで、必ず意味をもつということではない。その単語はどこまでそのドキュメントにおいて重要かを確認する必要があります。

〃これはとても面白い記事でしょう？そうでしょう？でしょう？でしょう？こんな記事は見たことないでしょう？でしょう？ 〃

単語の重要度：TF-IDF

TF-IDF (Term frequency-inverse document frequency)はドキュメント中の単語が、そのドキュメントにとってどの程度重要であるかを測る基本的な指標の一つです。

TF-IDFは以下の2つの値の積によって得られます:

- 対象の文章中の単語の出現頻度 (term frequency)
- 対象の単語が現れるドキュメントの割合 (document frequency) を、反転したもの (inverse)

$$\mathrm{tf}(t,d) = \log\left(1 + f_{t,d}\right) \qquad \mathrm{idf}(t, D) = \log \frac{N}{|\{d \in D : t \in d\}|}$$

(t: 単語, d: 文書, N: 文書の総数, D: 文書の集合)

実習問題（１０分）： TF-IDF計算

先ほど紹介されたTF-IDF式を参考して、TF-IDFを計算する関数を実装してください。

$$\mathrm{tf}(t,d) = \log\left(1 + f_{t,d}\right)$$

$$\mathrm{idf}(t, D) = \log \frac{N}{|\{d \in D : t \in d\}|}$$

(t: 単語, d: 文書, N: 文書の総数, D: 文書の集合)

TF-IDF計算

```
def tf_idf(word, doc_word_frequencies, corpus_word_frequencies, dataset_size):
  return tf(word, doc_word_frequencies) * idf(word, corpus_word_frequencies, dataset_size)

def tf(word, doc_word_frequencies):
  return log(1 + doc_word_frequencies[word])

def idf(word, corpus_word_frequencies, amount_of_documents):
  return  log(amount_of_documents / corpus_word_frequencies[word])
```

ウィキペディア記事のTF-IDF計算

```
wikipedia_frequencies = retrieve_wikipedia_wordfreqs(DB_PATH)
wikipedia_size = retrieve_wikipedia_articles_amount(DB_PATH)

def calculate_articles_tfidfs(db_path, title):
  word_freqs = dict(retrieve_articles_wordfreqs_by_title(db_path, title))
  tfidfs_dict = {}
  for word in article_word_frequencies:
    tfidfs_dict[word] = round(tf_idf(word, word_freqs, wikipedia_freqs, wikipedia_size), 5)
  return tfidfs_dict
```

ウィキペディア記事：TF-IDF vs 単語出現回数：「医学」

単語出現回数

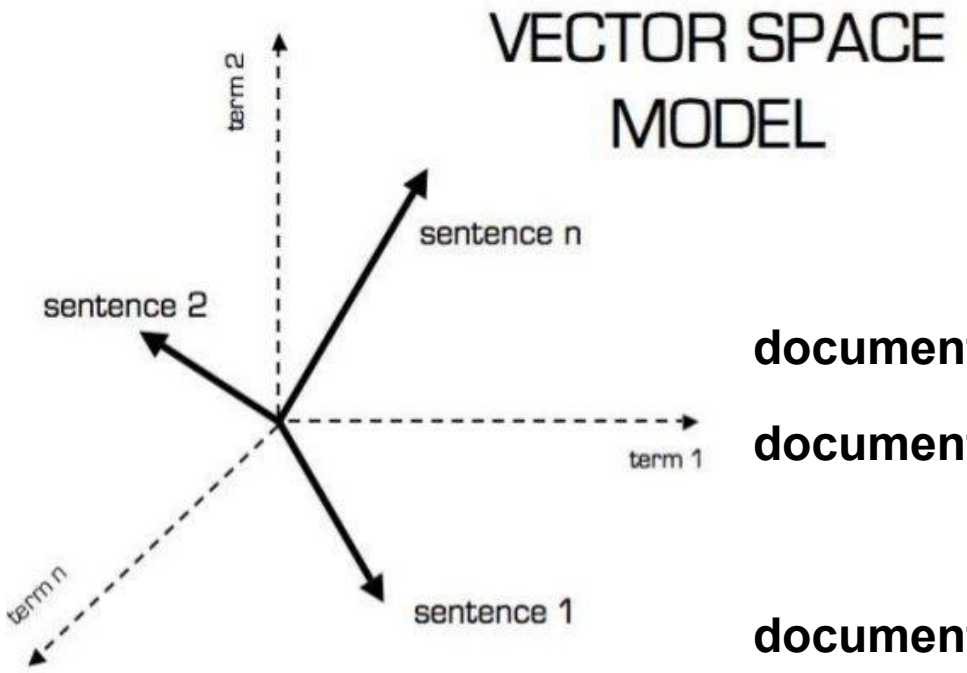
TF-IDF

NLP：高度解析・機械学習

機械学習のアルゴリズムは数値演算を行う関数を学習したい対象に合わせて最適化することで動作します。しかし自然言語は文字列なのでそのままでは数値として扱うことができず、機械学習のアルゴリズムを適用することが困難です。

そのためまず自然言語を数値のベクトルによる表現に変換する必要があります。

TF-IDF：ベクトル化



vocab = {猫, 犬, 鳥, 豚, 馬}

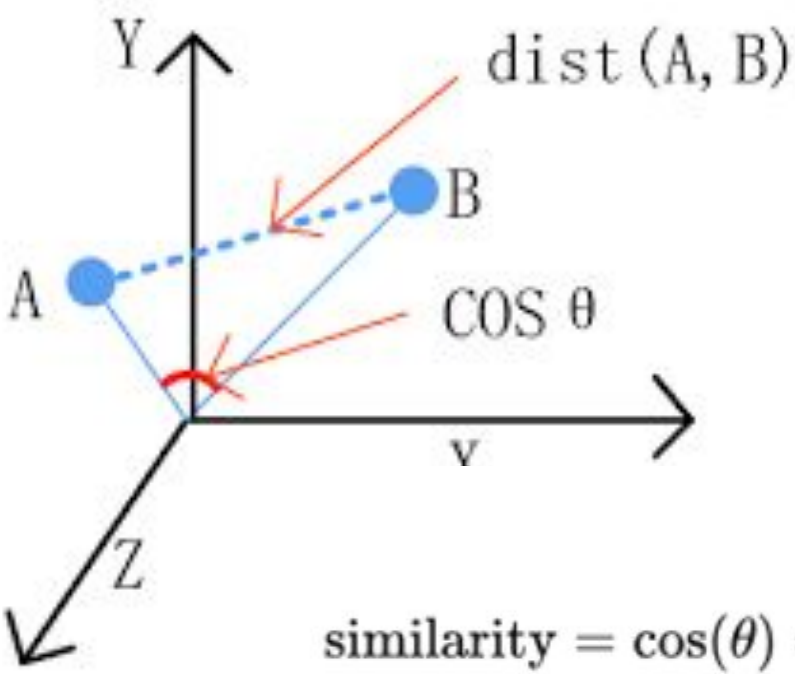
document_A = {猫: 0.65, 鳥: 0.02 }

document_A_vector = [0.65, 0, 0.02, 0, 0]

document_B = {馬: 0.43, 犬: 0.12, 鳥: 0.35}

document_B_vector = [0, 0.12, 0.35, 0, 0.43]

ドキュメントの類似度



document_A_vector = [0.65, 0, 0.02, 0, 0]

document_B_vector = [0, 0.12, 0.35, 0, 0.43]

$$\text{similarity} = \cos(\theta) = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\|\|\mathbf{B}\|} = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2}\sqrt{\sum_{i=1}^{n} B_i^2}}$$

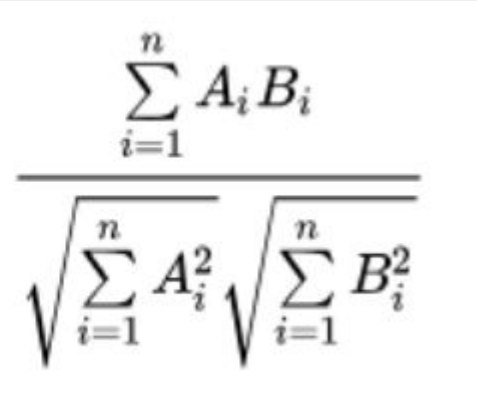
実習問題（１０分）：　類似度計算

先ほど紹介されたコサイン類似度の式を参考して、コサイン類似度を計算する関数を実装してください。

$$\text{similarity} = \cos(\theta) = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\|\|\mathbf{B}\|} = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2} \sqrt{\sum_{i=1}^{n} B_i^2}}$$

ウィキペディア記事間のコサイン類似度

```
# article1 and article2 are dicts of the form {word: tfidf_score}
def similarity(article1, article2):
  numerator_sum = 0
  # calculate the sum only for the words in common
  for word1 in article1:
    for word2 in article2:
      if word1 == word2:
        numerator_sum += article1[word1]*article2[word2]
  # calculate each of the square sums for all the words in each vector
  article1_squared_sum = 0
  for word in article1:
    article1_squared_sum += article1[word]**2
  article2_squared_sum = 0
  for word in article2:
    article2_squared_sum += article2[word]**2
  return numerator_sum / (sqrt(article1_squared_sum)*sqrt(article2_squared_sum))
```



ウィキペディア記事間のコサイン類似度

```
# calculate the similarity between 5 documents
def print_similarity_scores(target_article, articles_list):
  article1 = dict(calculate_articles_tfidfs(DB_PATH, target_article))
  for article_title in articles_list:
    article2 = dict(calculate_articles_tfidfs(DB_PATH, article_title))
    print("Similarity score ({} and {}): {}".format(target_article, article_title, similarity(article1, article2)))
```

```
target = "薬学"
list = ["ピアノ", "医学", "哲学", "物理学"]
print_similarity_scores(target, list )
```

```
Similarity score (薬学 and ピアノ): 0.0204478
Similarity score (薬学 and 医学): 0.198866
Similarity score (薬学 and 哲学): 0.071777
Similarity score (薬学 and 物理学): 0.09976
```

```
target = "アルゼンチン"
list = ["チリ", "言語", "IBM", "亜鉛"]
print_similarity_scores(target, list )
```

```
Similarity score (薬学 and チリ): 0.337001
Similarity score (薬学 and 言語): 0.144934
Similarity score (薬学 and IBM): 0.072863
Similarity score (薬学 and 亜鉛): 0.066898
```

Python自然言語処理入門

おめでとうございます！今日は以下のことを学びました:

- NLPにおけるテキスト処理の基本
- 単語出現頻度の分布のグラフの描画方法
- 文書中の単語をワードクラウドのような目を引くフォーマットで可視化する方法
- TF-IDFを使って二つの文章間の「類似性」を計算する方法

などを学びました。