

意味的ワードクラウド アルゴリズム解説

意味的ワードクラウド (Semantic Word Cloud) アルゴリズム解説

1. 概要

従来のワードクラウドとの違い
このプログラムが行うこと

2. 処理フロー

3. キーコンセプト

- 3.1 埋め込みベクトル (Embedding Vector)
- 3.2 コサイン類似度 (Cosine Similarity)
- 3.3 PCA (主成分分析)
- 3.4 初期配置: PCA vs t-SNE
- 3.5 Force-directed Layout (力指向レイアウト)

4. 各処理の詳細

- 4.1 ストップワードの読み込み
- 4.2 形態素解析
- 4.3 埋め込み取得とキャッシュ
- 4.4 Force-directed Layout の詳細
- 4.5 色の計算

5. パラメータの影響

- コマンドラインオプション
- 内部パラメータ
- パラメータ調整のヒント

6. 数学的背景

- 6.1 コサイン類似度の幾何学的意味
- 6.2 PCAの数学
- 6.3 Force-directed Layout の物理モデル
- 6.4 異方性スケールの意味

付録: コード構造

参考資料

- 関連研究と本実装の位置づけ
 - 既存の研究・実装
 - 本実装の特徴
 - 結論

意味的ワードクラウド (Semantic Word Cloud) アルゴリズム解説

項目	内容
作成者	荻 多加之
作成支援	Claude Code (Anthropic)
作成日	2024-12-25
最終更新	2025-12-28
バージョン	1.1

このドキュメントでは、`semantic_wordcloud.py` の動作原理を詳しく解説します。

1. 概要

従来のワードクラウドとの違い

項目	従来のワードクラウド	意味的ワードクラウド
配置基準	ランダム or スパイラル	意味的類似度
単語間の関係	考慮しない	類似した単語が近くに配置
技術	単純な衝突回避	埋め込みベクトル + Force-directed layout

このプログラムが行うこと

```
入力テキスト → 形態素解析 → 頻出単語抽出 → 埋め込みベクトル取得  
→ 意味的距離計算 → Force-directed配置 → 画像出力
```

2. 処理フロー



3. キーコンセプト

3.1 埋め込みベクトル (Embedding Vector)

単語を高次元の数値ベクトルに変換したもの

```
「機械学習」 → [0.012, -0.034, 0.056, ..., 0.023] (1536次元)  
「深層学習」 → [0.015, -0.031, 0.058, ..., 0.019] (1536次元)  
「料理」      → [-0.045, 0.067, -0.012, ..., 0.089] (1536次元)
```

- ・意味が似ている単語 → ベクトルも似ている（近い）
- ・意味が異なる単語 → ベクトルも異なる（遠い）

OpenAI の `text-embedding-3-small` モデルは1536次元のベクトルを出力します。

3.2 コサイン類似度 (Cosine Similarity)

2つのベクトルの「向き」の類似度を測る指標

$$\cos(\theta) = \frac{\mathbf{A} \cdot \mathbf{B}}{|\mathbf{A}| \times |\mathbf{B}|}$$

値の範囲: -1 ~ 1

- 1: 完全に同じ方向（非常に類似）
- 0: 直交（関連なし）
- 1: 正反対の方向（対義的）

実際の単語埋め込みでは、ほとんどの値が 0.3~0.9 の範囲に収まります。

```
# コード内での計算  
sim_matrix = cosine_similarity(embeddings) # 類似度行列  
dist_matrix = 1 - sim_matrix # 距離行列に変換
```

3.3 PCA (主成分分析)

高次元データを低次元に圧縮する手法

1536次元 → 2次元

目的：初期配置の決定

方法：データの分散が最大になる方向を見つけて射影

```
pca = PCA(n_components=2)  
initial_coords = pca.fit_transform(embeddings)
```

PCAは「できるだけ情報を保持しながら次元を減らす」ため、意味的に近い単語は2次元でも近くに配置されます。

3.4 初期配置: PCA vs t-SNE

本プログラムではPCAを初期配置に使用していますが、t-SNEも選択肢の一つです。

比較表

項目	PCA	t-SNE
計算速度	高速	遅い（反復計算）
決定性	決定的（毎回同じ結果）	確率的（毎回異なる）
大域構造	保持される	失われやすい
局所構造	部分的に保持	よく保持される
クラスタ分離	緩やか	明確に分離

PCAを選んだ理由

1. Force-directedとの相性

- PCAは大域的な構造を保持 → Force-directedで微調整
- t-SNEは既に局所最適化済み → Force-directedと役割が重複

2. 決定性

- PCAは同じ入力に対して常に同じ出力
- ランダムシードの管理が単純

3. 計算効率

- PCAは線形変換で高速
- t-SNEはO(N²)～O(N log N)の反復計算

t-SNEを使う場合のメリット・デメリット

メリット: - クラスタがより明確に分離される - 局所的な類似関係がよく表現される

デメリット: - 計算時間が長い - 大域的な距離関係が歪む（遠い点同士の距離が信頼できない） - Force-directedレイアウトの効果が薄れる（t-SNE自体が配置最適化） - perplexityパラメータの調整が必要

コード例 (t-SNEを使う場合)

```
from sklearn.manifold import TSNE

# PCAの代わりにt-SNEを使用
# tsne = TSNE(n_components=2, perplexity=30, random_state=42)
# initial_coords = tsne.fit_transform(embeddings)
```

PCAの限界と、それでも問題にならない理由（重要）

PCAの本質的な限界:

1. 線形手法である

- 1536次元の埋め込み空間は非線形構造を持つ
- PCAは線形射影なので、非線形な類似関係を捉えきれない

2. 最大分散 ≠ 意味的類似度

- PCAは「分散が最大になる方向」を探す
- これが「意味的に近い単語を近くに配置する」と一致するとは限らない

しかし、この設計では問題にならない:

PCAの役割 = 「だいたいの初期配置」を提供するだけ	
↓	
Force-directed = 高次元距離を使って本格的に最適化	

核心となるコード:

```

# Force-directedは高次元の距離を直接使う
ideal_distances = cosine_distance(embeddings) # 1536次元から計算

# PCAの出力は使わない！ただの初期位置として利用
spring_force = attraction_strength * (current_dist - ideal_dist)

```

つまり： - PCAの出力 → **初期位置だけ**（500回の反復で書き換わる） -
 最終配置 → **高次元コサイン距離が決める**

検証：初期配置を完全ランダムにしても、十分な反復で同様の結果が得られる：

```

# 極端な例：完全ランダム初期配置
word.x = center_x + random.uniform(-100, 100)
word.y = center_y + random.uniform(-100, 100)
# → 反復を増やすべ収束する（ただし収束が遅い）

```

まとめ：

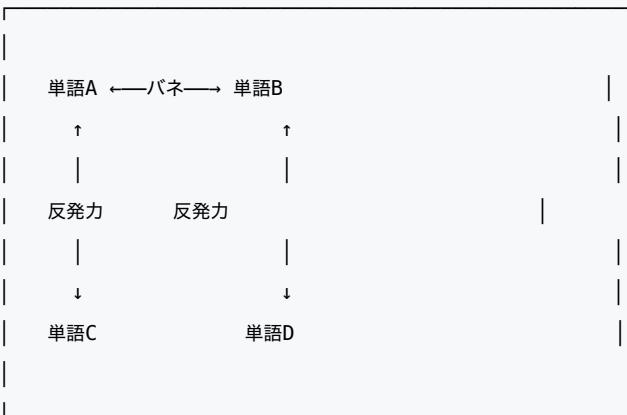
観点	評価
PCAは最適か？	いいえ、非線形構造には不向き
問題になるか？	いいえ、Force-directedが本質的な最適化を担う
改善の余地	ある（UMAPなど）が、実用上は十分

結論：本プログラムでは、Force-directedレイアウトが主役であり、 初期配置は「おおまかな配置」を提供すれば十分なため、高速で決定的なPCAを採用しています。 PCAの限界は、Force-directedの反復最適化によって補われます。

3.5 Force-directed Layout (力指向レイアウト)

物理シミュレーションによるグラフ配置アルゴリズム

各単語をノード、意味的関係をエッジとみなし、 バネと反発力でノードを動かして最適な配置を見つけます。



バネ: 理想距離に近づけようとする力

反発力: 重なりを避けるための力

4. 各処理の詳細

4.1 ストップワードの読み込み

```
def load_stopwords(custom_filepath: str = "stopwords.txt") -> set:
```

3つのソースからストップワードを収集:

1. **stopwords-iso** 日本語 (~134語)
 - 「これ」 「それ」 「ある」 「いる」 など
2. **stopwords-iso** 英語 (~1298語)
 - “the”, “and”, “is”, “of” など
3. **カスタムファイル** (stopwords.txt)
 - ユーザー定義の除外単語

キャッシュ機構: - 初回: URLからダウンロード → `stopwords_cache.json` に保存 - 2回目以降: キャッシュから読み込み (高速)

4.2 形態素解析

```
def extract_words(text: str) -> list[str]:
    tokenizer = Tokenizer() # Janome
    for token in tokenizer.tokenize(text):
        pos = token.part_of_speech.split(',')
        if pos[0] == '名詞' and pos[1] not in ['非自立', '代名詞', '数']:
            # 名詞かつ、非自立・代名詞・数詞でない
        ...
```

Janomeの出力例:

```
「機械学習の研究」
↓
機械学習 / 名詞, 固有名詞, 一般, *
の / 助詞, 連体化, *, *
研究 / 名詞, サ変接続, *, *
```

フィルタリング条件: - 品詞が「名詞」である - 名詞の中でも「非自立」「代名詞」「数」は除外 - 1文字の単語は除外 - ストップワードに含まれていない

4.3 埋め込み取得とキャッシュ

```
def get_embeddings(words: list[str], api_key: str = None,
                   cache_file: str = "embeddings_cache.json") -> np.ndarray:
```

キャッシュ構造:

```
{
    "機械学習": [0.012, -0.034, ...],
    "深層学習": [0.015, -0.031, ...],
    "ロボット": [0.023, 0.045, ...]
}
```

処理フロー:

1. キャッシュファイルを読み込み
2. 要求された単語のうち、キャッシュにある/ないを分離
3. キャッシュにない単語だけAPIで取得
4. 新しい単語をキャッシュに追加・保存
5. 要求された順序で埋め込みを返す

4.4 Force-directed Layout の詳細

```
def force_directed_layout(words, canvas_width, canvas_height, iterations, verbose):
```

Step 1: 異方性スケールの計算

```
aspect_ratio = canvas_width / canvas_height # 例: 900/700 = 1.286
scale_x = np.sqrt(aspect_ratio)           # 例: 1.134
scale_y = 1.0 / np.sqrt(aspect_ratio)     # 例: 0.882
```

これにより、正方形ではなく橿円形のレイアウトになります。

Step 2: 距離行列の計算

```
sim_matrix = cosine_similarity(embeddings) # N×N の類似度行列
dist_matrix = 1 - sim_matrix             # 距離に変換 (0~2)

# 理想距離にスケーリング
ideal_distances = (dist_matrix / max_dist) * ideal_scale
```

例:

	機械学習	深層学習	料理
機械学習	0.0	0.15	0.85
深層学習	0.15	0.0	0.82
料理	0.85	0.82	0.0

→ 機械学習と深層学習は近く、料理は遠く配置される

Step 3: PCAによる初期配置

```
pca = PCA(n_components=2)
initial_coords = pca.fit_transform(embeddings)

# 異方性スケールを適用
word.x = center_x + initial_coords[i, 0] * scale * scale_x
word.y = center_y + initial_coords[i, 1] * scale * scale_y
```

Step 4: 力の計算と更新

3種類の力と計算次元（重要）：

このアルゴリズムの核心は、**高次元の意味的距離を2次元配置で再現すること**です。各力がどの次元で計算されるかを理解することが重要です。

力	理想/目標	現在/実際	目的
スプリング力	1536次元（コサイン距離）	2次元（キャンバス）	意味的関係の再現
反発力	-	2次元のみ	視覚的重なり防止
中心引力	-	2次元のみ	キャンバス内収束

1. スプリング力 (attraction) ← 高次元→2次元マッピングの核心

```
# 理想距離: 1536次元空間でのコサイン距離から算出
ideal_dist = ideal_distances[i, j] # 高次元由来

# 現在距離: 2次元キャンバス上の距離
current_dist = np.sqrt(dx**2 + dy**2) # 2次元

spring_force = attraction_strength * (current_dist - ideal_dist)
```

- 現在距離 > 理想距離 → 引き寄せる
- 現在距離 < 理想距離 → 押し離す
- 意味的に近い単語は2Dでも近くに配置される

2. 反発力 (repulsion) ← 純粋に2次元の視覚的問題

```

# すべて2次元で計算
min_sep = (words[i].width + words[j].width) / 2 + 5 # 単語幅 (2D)
actual_dist = np.sqrt(dx**2 + dy**2) # 2D距離

if actual_dist < min_sep:
    repulsion = repulsion_strength / (actual_dist**2 + 1)

```

- 単語が重なりそうなとき発生
- 距離が近いほど強い
- 意味的距離とは無関係、視覚的な配置のみ考慮

3. 中心への引力 (center force) ← 2次元

```
center_force = 0.002 * dist_to_center
```

- 単語が散らばりすぎないように中心に引き寄せる

シミュレーションループ:

```

for iteration in range(iterations): # デフォルト500回
    forces = compute_forces()
    temperature = max(0.1, 1.0 - iteration / iterations) # 冷却

    for word in words:
        word.vx = (word.vx + force) * damping * temperature
        word.vy = (word.vy + force) * damping * temperature
        word.x += word.vx
        word.y += word.vy

```

温度 (temperature): - 初期: 1.0 → 大きく動く - 最終: 0.1 → 微調整のみ - 徐々に冷却することで安定した配置に収束

4.5 色の計算

色の計算は、意味的に近い単語に似た色を付けるための処理です。

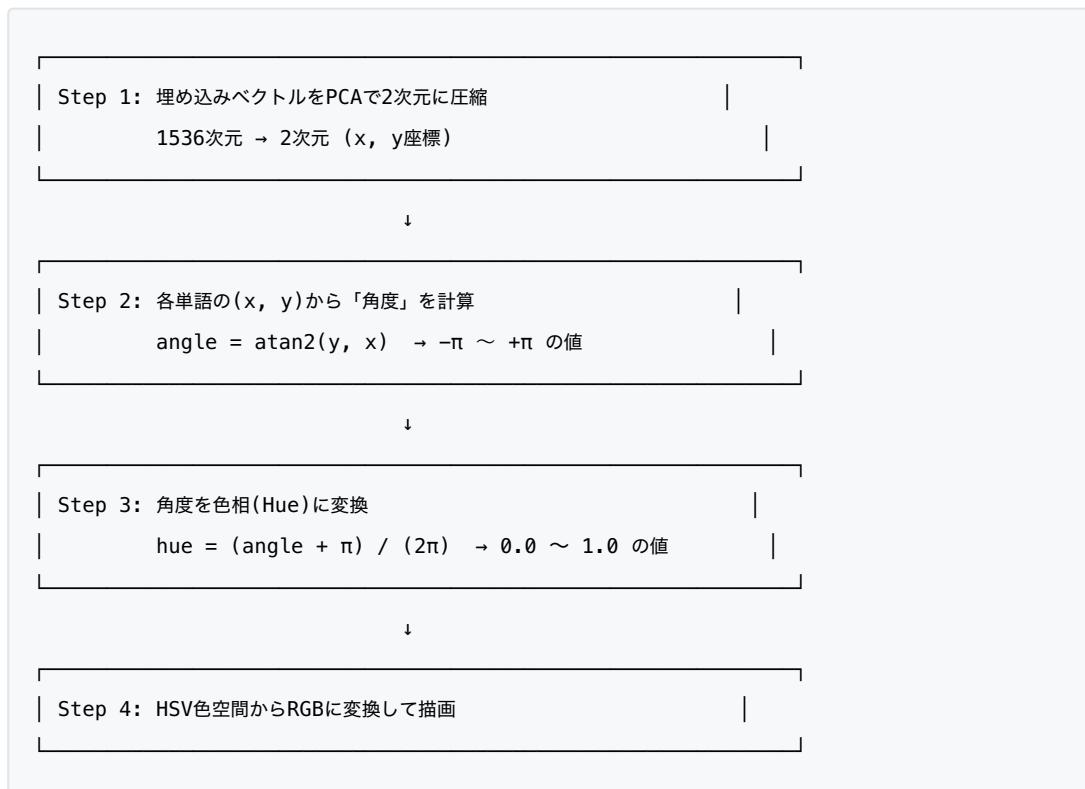
目的

意味的に近い単語 → 似た色

意味的に遠い単語 → 異なる色

これにより、ワードクラウドを見たときに「このグループは似た話題」と直感的にわかります。

処理の流れ



Step 1: PCAで2次元に圧縮

```
pca = PCA(n_components=2)
coords_2d = pca.fit_transform(embeddings)
```

1536次元の埋め込みを2次元に圧縮します。 意味的に近い単語は、2次元でも近い位置に配置されます。

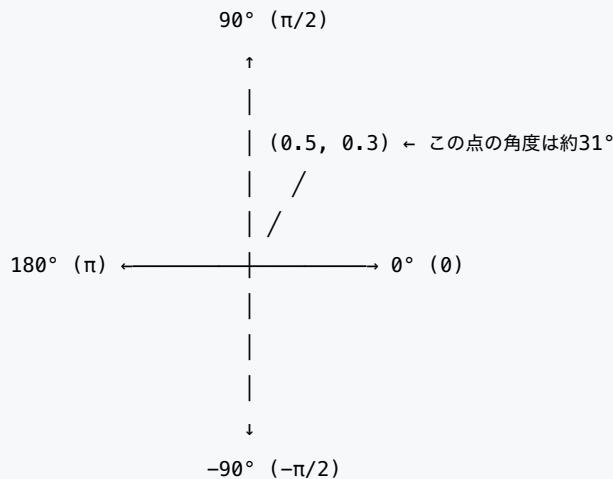
例：

「機械学習」 \rightarrow (0.5, 0.3)
「深層学習」 \rightarrow (0.6, 0.4) ← 近い！
「料理」 \rightarrow (-0.8, -0.2) ← 遠い

Step 2: 角度の計算

```
angle = np.arctan2(y, x)
```

`atan2(y, x)` は、原点から点(x, y)への角度を返します。



戻り値の範囲: $-\pi \sim +\pi$ (約 $-3.14 \sim +3.14$)

Step 3: 角度を色相に変換

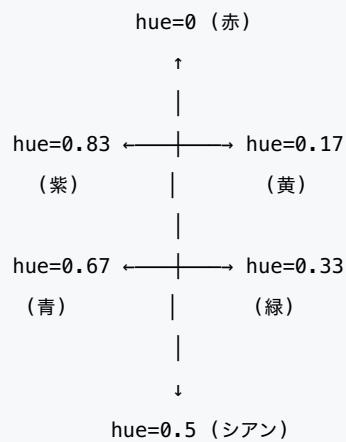
```
hue = (angle + np.pi) / (2 * np.pi)
```

角度 ($-\pi \sim +\pi$) を色相 (0 ~ 1) に変換します。

変換式の意味：

$\text{angle} = -\pi \rightarrow \text{hue} = (-\pi + \pi) / (2\pi) = 0.0$ (赤)
 $\text{angle} = 0 \rightarrow \text{hue} = (0 + \pi) / (2\pi) = 0.5$ (シアン)
 $\text{angle} = +\pi \rightarrow \text{hue} = (+\pi + \pi) / (2\pi) = 1.0$ (赤)

色相環 (Hue Circle)



Step 4: HSVからRGBへの変換

```
saturation = 0.65 + 0.3 * min(dist / max_dist, 1.0) # 彩度  
value = 0.85 # 明度  
rgb = colorsys.hsv_to_rgb(hue, saturation, value)
```

HSV色空間: - **H (Hue/色相):** 色の種類 (0~1で赤→黄→緑→シアン→青→紫→赤) - **S (Saturation/彩度):** 色の鮮やかさ (0=灰色、1=鮮やか) - **V (Value/明度):** 明るさ (0=黒、1=明るい)

彩度の調整:

```
saturation = 0.65 + 0.3 * min(dist / max_dist, 1.0)
```

- 原点から遠い単語 (dist大) → 彩度が高い (鮮やか)
- 原点に近い単語 (dist小) → 彩度が低い (くすんだ色)

これにより、中心にある「一般的な単語」は控えめな色、外側にある「特徴的な単語」は鮮やかな色になります。

実例

PCA座標と色の対応：

単語	(x, y)	角度	色相	色
機械学習	(0.5, 0.3)	31°	0.59	シアン系
深層学習	(0.6, 0.4)	34°	0.59	シアン系（似た色）
ロボット	(0.7, -0.2)	-16°	0.46	緑系
細胞	(-0.3, 0.6)	117°	0.82	紫系
料理	(-0.8, -0.2)	-166°	0.04	赤系

なぜPCAを使うのか（配置と別に計算する理由）

色の計算では、配置用のPCAとは別に、色専用のPCAを実行しています。

```
def compute_semantic_colors(words: list[Word]) -> list[tuple]:  
    # 新たにPCAを実行（配置とは独立）  
    embeddings = np.array([w.embedding for w in words])  
    pca = PCA(n_components=2)  
    coords_2d = pca.fit_transform(embeddings)
```

理由： - 配置は異方性スケールやForce-directedで変形されている - 色は純粋な「意味的な位置関係」を反映させたい - PCAの第1・第2主成分が意味的なクラスタを捉える

5. パラメータの影響

コマンドラインオプション

オプション	デフォルト	説明
-n, --num-words	80	表示する単語数
--cache-words	200	キャッシュする単語数
--iterations	500	シミュレーション反復回数
--seed	None	ランダムシード
-o, --output	semantic_wordcloud.png	出力ファイル名

内部パラメータ

```
# force_directed_layout 内
attraction_strength = 0.01      # スプリングの強さ
repulsion_strength = 700        # 反発力の強さ
damping = 0.9                  # 減衰係数

# 重なり解消パラメータ
padding = 3                     # 重なり検出のパディング
nudge = 3.5                     # 重なり解消時の移動量
overlap_iterations = 150         # 重なり解消の反復回数

# 回転制限
# 5文字以上の単語は回転させない（長い単語の回転はレイアウトを崩しやすい）

# キャンバス
canvas_width = 900
canvas_height = 700

# フォントサイズ
font_size = 12 + 28 * (ratio ** 0.5)  # 12~40の範囲
```

パラメータ調整のヒント

症状	調整
単語が散らばりすぎ	attraction_strength を上げる
単語が密集しすぎ	repulsion_strength を上げる
配置が安定しない	iterations を増やす
重なりが多い	repulsion_strength、padding、nudge を上げる
隙間が多すぎる	repulsion_strength、padding、nudge を下げる
長い単語の回転で崩れる	回転制限の文字数を調整

6. 数学的背景

6.1 コサイン類似度の幾何学的意味

2つのベクトル A, B のコサイン類似度:

$$\cos(\theta) = (\mathbf{A} \cdot \mathbf{B}) / (|\mathbf{A}| \times |\mathbf{B}|)$$

ここで:

$$\mathbf{A} \cdot \mathbf{B} = \sum(a_i \times b_i) \quad (\text{内積})$$

$$|\mathbf{A}| = \sqrt{\sum(a_i^2)} \quad (\text{ノルム})$$

なぜコサイン類似度を使うのか: - ベクトルの「大きさ」ではなく「向き」を比較 - 単語の出現頻度に依存しない - 値が $-1 \sim 1$ に正規化される

6.2 PCAの数学

目標: 分散を最大化する方向を見つける

1. データの平均を引く (中心化)
2. 共分散行列を計算: $C = (1/n) \times X^T \times X$
3. 固有値分解: $C \times v = \lambda \times v$
4. 最大固有値に対応する固有ベクトルが第1主成分

1536次元 → 2次元の場合、上位2つの固有ベクトルを使用。

6.3 Force-directed Layout の物理モデル

フックの法則 (バネ):

$$F = -k \times (x - x_0)$$

k : バネ定数 (attraction_strength)

x : 現在の距離

x_0 : 理想距離 (ideal_distance)

クーロンの法則 (反発):

$$F = k \times q_1 \times q_2 / r^2$$

簡略化: $F = repulsion_strength / (r^2 + 1)$

運動方程式:

```
v(t+1) = (v(t) + F) × damping × temperature  
x(t+1) = x(t) + v(t+1)
```

6.4 異方性スケールの意味

```
aspect_ratio = width / height = 900 / 700 ≈ 1.286  
  
scale_x = √(aspect_ratio) ≈ 1.134  
scale_y = 1 / √(aspect_ratio) ≈ 0.882  
  
scale_x × scale_y = 1 (面積保存)
```

これにより: - x方向に1.134倍伸ばす - y方向に0.882倍縮める - 全体の「面積」は変わらない - 結果として横長の橢円形になる

付録: コード構造

```
semantic_wordcloud.py
|
├── 定数・設定
│   ├── FONT_PATHS
│   ├── JAPANESE_STOPWORDS_URL
│   ├── ENGLISH_STOPWORDS_URL
│   └── STOPWORDS_CACHE_FILE
|
├── クラス
│   └── Word (単語の状態を保持)
│       ├── text, freq, font_size, embedding
│       ├── x, y, vx, vy, rotation
│       ├── width, height (プロパティ)
│       ├── get_bbox()
│       └── overlaps()
|
├── ユーティリティ関数
│   ├── load_stopwords()
│   ├── get_font_path()
│   ├── extract_words()
│   ├── load_text_from_excel()
│   └── load_text()
|
└── コア処理
    ├── get_embeddings()
    ├── force_directed_layout()
    │   ├── compute_forces()
    │   └── count_overlaps()
    ├── compute_semantic_colors()
    └── render_wordcloud()
|
└── main()
    └── 引数解析・処理フロー制御
```

参考資料

- OpenAI Embeddings
- scikit-learn: PCA
- Force-directed graph drawing (Wikipedia)
- stopwords-iso

- Janome
-

7. 関連研究と本実装の位置づけ

7.1 既存の研究・実装

意味的ワードクラウドは新しいアイデアではなく、いくつかの先行研究・実装が存在します。

学術研究

研究	手法
Semantic word cloud generation based on word embeddings (2016)	Word2Vecで意味的距離を計算、グラフベースのレイアウト
Semantic Word Clouds with t-SNE	t-SNEで意味的配置を生成
ReCloud	文法的依存関係からセマンティックグラフを構築、Force-directedレイアウト

既存ツール・実装

プロジェクト	特徴
Arizona大学 Semantic Word Cloud	Webツール、複数アルゴリズム対応
nlp-chula/swordcloud	Python、タイ語NLP研究グループ
ttavni/SemanticWordClouds	PKE + 事前学習埋め込み
WordCloud.jl	Julia、t-SNE対応

7.2 本実装の特徴

既存手法との共通点

- 埋め込みベクトルによる意味的距離の計算
- Force-directedレイアウト
- PCAによる初期配置・色付け

本実装の独自の組み合わせ

要素	本実装	一般的な既存実装
埋め込み	OpenAI text-embedding-3-small	Word2Vec, GloVe, FastText
ベクトル次元	1536次元	100-300次元
距離制約	高次元コサイン距離を直接使用	2D投影後の距離
レイアウト形状	異方性（橿円形）対応	多くは正方形のみ
言語対応	日本語 (Janome + stopwords-iso)	英語中心が多い

技術的な差別化ポイント

1. OpenAI Embeddings の利用

- 最新の大規模言語モデルベースの埋め込み
- 1536次元の高密度な意味表現
- 既存実装の多くはWord2Vec/GloVe (100-300次元)

2. 高次元距離の直接利用

- 多くの実装: 先にt-SNE/PCAで2Dに落としてから配置
- 本実装: 高次元のコサイン距離を理想距離として保持し、Force-directedで2D再現
- これにより、次元削減による情報損失を最小化

3. 異方性Force-directed

- 横長・縦長など任意のアスペクト比に対応
- 多くの実装は正方形レイアウトのみ

4. 日本語ネイティブ対応

- Janomeによる形態素解析
- stopwords-isoの日本語ストップワード
- 英語中心の既存ツールとの差別化

7.3 結論

個々の技術（埋め込み、Force-directed、PCA）は既存のものですが、「OpenAI Embeddings + 高次元距離制約 + 異方性レイアウト + 日本語対応」という組み合わせは、調査した範囲では前例が見つかりませんでした。