地理情報と言語処理 実践入門

松村結衣（yuiseki）

株式会社 Helpfeel, Gyazo プロダクトマネージャー
UN Open GIS Initiative, WG7, UN Smart Maps Group

チュートリアルの内容

- 導入 (5 分)
- デモ (10 分)
- ハンズオン (45 分)
- 研究紹介 (15 分)
- まとめ (5 分)
- 質疑応答 (10 分)

自己紹介 (1)

- 名前
  - 松村結衣 (yuiseki)
- 出身大学
  - 慶應義塾大学 政策・メディア研究科
    - 修士論文「街に着目した Twitter 上のメッセージ分析について」(2010)
- 所属
  - 株式会社 Helpfeel, Gyazo プロダクトマネージャー
  - UN Open GIS Initiative, DWG7: UN Smart Maps Group

自己紹介 (2)

株式会社 Helpfeel

- 言語処理や生成 AI を積極的に活用したプロダクトの開発
  - Gyazo
  - Cosense (formerly Scrapbox)
  - Helpfeel
- Gyazo のプロダクトマネージャーを担当
  - Gyazo は画像、動画、音声、テキストを組み合わせたプロダクト
    - マルチモーダルに関心を持っている

自己紹介 (3)

UN Open GIS Initiative

- 国連本部 地理空間情報課 (UN Geospatial) が主導している取り組み
  - 国連を **オープンソースソフトウェア** で支える
    - 現状: Microsoft Azure, Esri ArcGIS, Mapbox
- DWG7 (Domain Working Group 7) : UN Smart Maps Group に所属
  - GIS を中心とした最新技術を調査・研究・開発
    - 例: Raspberry Pi、ベクトルタイル、Web3、PMTiles、生成 AI
  - 月例会議での報告
  - 国連職員向けの Webinar を不定期で開催

チュートリアルのゴール

言語研究分野で地理情報を扱う研究がさらに盛り上がってほしい

- 言語 + 地理という領域で様々な研究アイデアが探求され発表される
    - それを見て刺激を受けて私が捗る

そのために私にできること

- 必要な情報や知識を伝える、実践的なツールやスキルを伝える
- この分野の面白さ・奥深さ、やりがい、可能性を伝える
- この分野のコミュニティの紹介をする

チュートリアルを通じて伝えたいこと

- 地理情報技術のオープンデータや API やライブラリの使い方
- 生成 AI をソフトウェアシステムのパーツとして扱うテクニック
- この 2 つの組み合わせ

デモ

https://trident.yuiseki.net/

デモ：自然言語指示に基づいた地理情報の取得と可視化

- https://trident.yuiseki.net/

デモ：自然言語による地理情報 DB に対する質問応答

- https://trident.yuiseki.net/duckdb

デモ：自然言語指示に基づいた地図スタイルの変更

- https://trident.yuiseki.net/charites

ハンズオン

ハンズオンのゴール

- 地理情報技術と言語処理技術（LLM、生成 AI 含む）を組み合わせて試行錯誤ができる環境を用意して実際に動かす
- ハンズオンを通じて、デモの裏側で使われている技術要素と組み合わせ方が理解できる
- 必要に応じて地理情報技術の概念や用語を解説する

ハンズオンの前提 (1)

- Python を使います
  - Jupyter Notebook を使います
    - Google Colab でも動くはず

```
pip install jupyter notebook
```

ハンズオンの前提 (2)

生成 AI 関係のライブラリ

- LangChain
- Google Gemini
- Chroma

```
pip install langchain langchain-core langchain-common
pip install langchain-google-genai langchain-chroma
```

ハンズオンの前提 (3)

地理情報関係のライブラリ

- Folium
- DuckDB
- mapwidget

```
pip install folium duckdb mapwidget
```

OpenStreetMap の概要

- このチュートリアルでは、OpenStreetMap を随所で使用します
- OpenStreetMap は、フリーな地図データを作成・配布する事を目指したプロジェクトです
  - https://www.openstreetmap.org/
  - https://wiki.openstreetmap.org/wiki/JA:Main_Page

OpenStreetMap 注意事項

https://www.openstreetmap.org/copyright

> あなたは OpenStreetMap とその協力者をクレジットする限り、データを自由にコピー、配布、送信、利用することができます。データを改変したり翻案したりした場合、元データと同じライセンスを適用することによって配布を行うことができます。あなたの権利と責任は リーガルコード で解説されています。

--

要するに

- OpenStreetMap のデータを使う場合は、クレジットを表示すること
- OpenStreetMap のデータを改変して配布する場合は、同じライセンスを適用すること

Overpass API の概要

- Overpass API は、OpenStreetMap のデータを取得するための読み取り専用の API です
  - https://overpass-turbo.eu/
  - https://wiki.openstreetmap.org/wiki/JA:Overpass_API
- Overpass API は、 XML または Overpass QL という専用のクエリ言語を使用してデータを取得します
  - https://wiki.openstreetmap.org/wiki/JA:Overpass_API/Overpass_QL

Overpass API 注意事項

- このチュートリアルでは、 z.overpass-api.de で提供されている Overpass API を使用しています
- z.overpass-api.de で提供されている Overpass API は、 **一日あたり 10,000 クエリまたは 1GB のデータ取得** が目安です
  - この頻度を超えるリクエストが必要な場合は、自分で Overpass API サーバーを立てることができます
  - インストール方法は公式ドキュメントを参照してください
    - https://overpass-api.de/no_frills.html
    - https://overpass-api.de/full_installation.html
    - https://wiki.openstreetmap.org/wiki/Overpass_API/Installation

Natural Earth の概要

- このチュートリアルでは、Natural Earth のデータを使用します
- Natural Earth は、パブリックドメインの地図データセットです
  - https://www.naturalearthdata.com/
  - https://www.naturalearthdata.com/about/terms-of-use/

ハンズオンの内容

- 001: Overpass API と Few-Shot Prompt
  - 001-001: Overpass API による地域情報取得
  - 001-002: Few-Shot Prompt
- 002: DuckDB と SQL による地理空間情報データ分析
  - 002-001: DuckDB と Text-to-SQL
  - 002-002: SQL RAG
- 003: ベクトルタイルと地図スタイルカスタマイズ

001: Overpass API と Few-Shot Prompt

- 001-001: Overpass API による地域情報取得
- 001-002: Few-Shot Prompt

001-001: Overpass API による地域情報取得

ゴール

「生成 AI を使って自然言語から Overpass QL を出力し、 Overpass API で地域情報を取得する」

001-001: Overpass API による地域情報取得

ステップ

- 生成 AI に、自然言語から Overpass QL を出力させる
- 出力された Overpass QL で、 Overpass API からデータを取得する
- 取得したデータを GeoJSON に変換する
- GeoJSON を地図上に表示する

001-001: Overpass API による地域情報取得 | プロンプト

```
from langchain_core.prompts import ChatPromptTemplate

template = """You are an expert in Overpass API Query.
Output the appropriate Overpass API Query from the user input.

You will always reply according to the following rules:
- Output valid Overpass API query.
- The query MUST be out json.
- The query MUST be out geom.
- The query MUST be set timeout as 30000.
- The query will utilize a area specifier as needed.
- The query will search nwr as needed.
- The query MUST be line delimited and surrounded by just three back quote to indicate that it is a code block.

Hints:
- Regardless of the language of the user input, use area["name"="..."]. Because the area names are in the local language.
- If the user input is for a specific type of restaurant, amenity should be restaurant and filter by cuisine.

** Important **
Take a deep breath and carefully check if it is in accordance with the rules and appropriate as an Overpass API Query.

User Input:
{input}
"""

prompt = ChatPromptTemplate.from_template(template)
```

001-001: Overpass API による地域情報取得 | AI モデルの準備

```
from langchain_google_genai import ChatGoogleGenerativeAI

# モデルの準備
model = ChatGoogleGenerativeAI(model="gemini-exp-1206", temperature=0)
```

001-001: Overpass API による地域情報取得 | AI 呼び出し

```
input_text = "長崎県長崎市のカフェを地図に表示してください。"

chain = prompt | model

res = chain.invoke({"input": input_text})
result = res.content.strip()
print(result)
```

001-001: Overpass API による地域情報取得 | AI 出力

````
```
[out:json][timeout:30000];
area["name"="長崎市"]->.searchArea;
(
  nwr["amenity"="cafe"](area.searchArea);
);
out geom;
```
````

```
# 正規表現でresultから ``` を使って Overpass QL のみを抽出
import re
match = re.search(r"```[^\n]*\n(.*?)```", result, re.DOTALL)
overpass_query = match.group(1).strip()
```

001-001: Overpass API による地域情報取得 | Overpass API 呼び出し

```python
import requests

def get_overpass_response(overpass_query: str):
    query_string = f"data={requests.utils.quote(overpass_query)}"
    overpass_api_url = f"https://z.overpass-api.de/api/interpreter?{query_string}"
    response = requests.get(overpass_api_url)
    if response.status_code == 200:
        res_json = response.json()
        return res_json
    else:
        print(f"Error: {response.status_code}")
        print(response.text)
        return None

overpass_response = get_overpass_response(overpass_query)
```

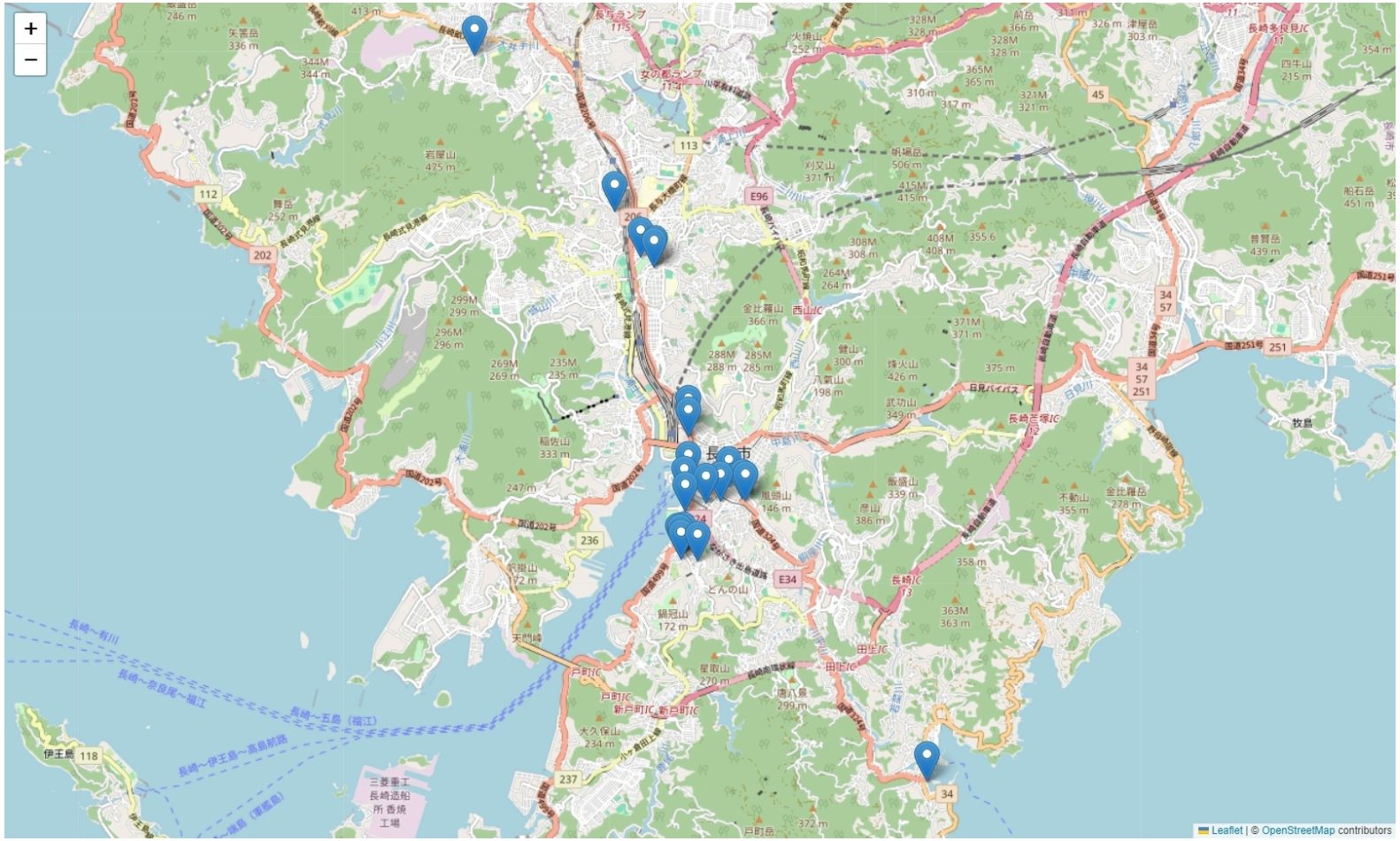
001-001: Overpass API による地域情報取得 | GeoJSON 変換

```
def convert_overpass_to_geojson(overpass_response):
    geojson = {
        "type": "FeatureCollection",
        "features": []
    }

    for element in overpass_response['elements']:
        feature = {
            "type": "Feature",
            "properties": {},
            "geometry": {}
        }
        if 'tags' in element:
            feature['properties'] = element['tags']
        if element['type'] == 'node':
            feature['geometry'] = {
                "type": "Point",
                "coordinates": [element['lon'], element['lat']]
            }
        elif element['type'] == 'way':
            feature['geometry'] = {
                "type": "Polygon" if 'nodes' in element else "LineString",
                "coordinates": [[(n['lon'], n['lat']) for n in overpass_response['elements'] if n['id'] in element['nodes']]]
            }
        elif element['type'] == 'relation':
            # Handle relations as MultiPolygons (simplified)
            feature['geometry'] = {"type": "MultiPolygon", "coordinates": []}
            for member in element['members']:
                if member['type'] == 'way':
                    way_id = member['ref']
                    way = next((w for w in overpass_response['elements'] if w['id'] == way_id and w['type'] == 'way'), None)
                    if way:
                        coordinates = [[(n['lon'], n['lat']) for n in overpass_response['elements'] if n['id'] in way['nodes']]]
                        if coordinates:
                            feature['geometry']['coordinates'].append([coordinates[0]]) #Multipolygonなので、さらにリストで囲む
        geojson["features"].append(feature)
    return geojson

geojson_data = convert_overpass_to_geojson(overpass_response)
```

001-001: Overpass API による地域情報取得 | 地図上に可視化

```
import folium

m = folium.Map(location=[0, 0], zoom_start=2)

folium.GeoJson(geojson_data).add_to(m)

bounds = folium.GeoJson(geojson_data).get_bounds()
if bounds:
    m.fit_bounds(bounds)

display(m)
```

001-001: Overpass API による地域情報取得 | 地図上に可視化 結果



001-002: Few-Shot Prompt

ゴール

「生成 AI に Examples を与えて Few-Shot Prompt で Overpass QL を出力する」

001-002: Few-Shot Prompt

ステップ

- Examples を Embedding しておく
- 生成 AI に自然言語を入力
- Embedding に変換する
- Embedding が距離的に近い Examples を探す
- Examples に基づき Few Shot で Overpass QL を出力する
- Overpass API で地域情報を取得
- GeoJSON で地図上に可視化

001-002: Few-Shot Prompt | Examples を用意

```
examples:
  - input: "長崎県長崎市のカフェを地図に表示してください。"
    output: |
      ```
      [out:json][timeout:30000];
      area["name"="長崎市"]->.searchArea;
      (
        nwr["amenity"="cafe"](area.searchArea);
      );
      out geom;
      ```
  - input: "東京都台東区のラーメン屋を地図に表示してください。"
    output: |
      ```
      [out:json][timeout:30000];
      area["name"="台東区"]->.searchArea;
      (
        nwr["amenity"="restaurant"]["cuisine"="ramen"](area.searchArea);
      );
      out geom;
      ```

  ......
```

001-002: Few-Shot Prompt | Embedding モデル準備

```
from langchain_google_genai import GoogleGenerativeAIEmbeddings

embeddings = GoogleGenerativeAIEmbeddings(model="models/text-embedding-004")
```

001-002: Few-Shot Prompt | Example Selector 準備

```
from langchain_core.example_selectors import SemanticSimilarityExampleSelector
from langchain_chroma import Chroma

example_selector = SemanticSimilarityExampleSelector.from_examples(
    examples,
    embeddings,
    Chroma,
    k=3,
)
```

001-002: Few-Shot Prompt | Few-Shot Prompt 準備

```
from langchain_core.prompts import FewShotPromptTemplate

few_shot_prompt = FewShotPromptTemplate(
    example_selector=example_selector,
    example_prompt=example_prompt,
    prefix="""You are an expert in OpenStreetMap and Overpass QL.
Output the appropriate Overpass QL from the user input.

You will always reply according to the following rules:
- Output valid Overpass QL.
- The query MUST be out json.
- The query MUST be out geom.
- The query MUST be set timeout as 30000.
- The query will utilize a area specifier as needed.
- The query will search nwr as needed.
- The query MUST be line delimited and surrounded by just three back quote to indicate that it is a code block.

** Examples: **
""",
    suffix="""User input:
{input}

Output:""",
    input_variables=["input"],
)
```

001-002: Few-Shot Prompt | AI 呼び出し

```
input_text = "東京都台東区のカフェを地図に表示してください。"

chain = few_shot_prompt | model

res = chain.invoke({"input": input_text})
result = res.content.strip()
print(result)
```

001-002: Few-Shot Prompt | AI 出力

````
```
[out:json][timeout:30000];
area["name"="台東区"]->.searchArea;
(
  nwr["amenity"="cafe"](area.searchArea);
);
out geom;
```
````

001-002: Few-Shot Prompt | 何がすごいの？

- 長崎県長崎市のカフェ
- 東京都台東区のラーメン屋

→

- 東京都台東区のカフェ

002: DuckDB と SQL による地理空間情報データ分析

- 002-001: DuckDB と Text-to-SQL
- 002-002: SQL RAG

002-001: DuckDB と Text-to-SQL

ゴール

「生成 AI を使って自然言語から SQL を出力し、DuckDB で地理空間情報データ分析を行う」

ステップ

- DuckDB でデータベースを作成
- 生成 AI でテーブルスキーマと自然言語から SQL を出力
- DuckDB で SQL を実行
- 結果を GeoJSON で地図上に可視化

002-001: DuckDB と Text-to-SQL | DuckDB 初期化

```
import duckdb

# DuckDBに接続（インメモリで動作）
conn = duckdb.connect()

# DuckDBに拡張機能をインストール
conn.execute("""
INSTALL httpfs;
INSTALL json;
INSTALL spatial;
""")
conn.execute(f"""
LOAD httpfs;
LOAD json;
LOAD spatial;
""")
```

002-001: DuckDB と Text-to-SQL | データベース作成

```
admin_geojson_url = "https://github.com/nvkelso/natural-earth-vector/raw/master/geojson/ne_110m_admin_0_countries.geojson"

# GeoJSON を DuckDBにテーブルとして読み込む
conn.execute(f"CREATE TABLE countries AS SELECT * FROM ST_Read('{admin_geojson_url}')")
```

002-001: DuckDB と Text-to-SQL | テーブルスキーマ準備

```
summary_of_tables = ""

# SHOWによってテーブル一覧を取得
show_result = conn.execute("SHOW").fetchall()
tables = [row[2] for row in show_result]

for table in tables:
    summary_of_tables += f"Table: {table}\n"
    # DESCRIBE TABLEの結果を文字列に変換
    describe_result = conn.execute(f"DESCRIBE TABLE {table}").fetchall()
    for row in describe_result:
        field_name = row[0]
        field_type = row[1]
        summary_of_tables += f"  Field: {field_name}, {field_type}\n"
print(summary_of_tables)
```

002-001: DuckDB と Text-to-SQL | プロンプト

```
template = """You are an expert of PostgreSQL and PostGIS.
You output the best PostgreSQL query based on given table schema and input text.

You will always reply according to the following rules:
- Output valid PostgreSQL query.
- The query MUST be return name, value and geom columns. Use AS to rename columns.
- The query MUST use ST_AsGeoJSON function to output geom column.
- The query MUST be line delimited and surrounded by just three back quote to indicate that it is a code block.
- The Value column should be a value that takes into account the user's intent to the greatest extent possible.

** Table Schema: **
{table_schema}

User Input:
{input}
"""

prompt = ChatPromptTemplate.from_template(template)
```

002-001: DuckDB と Text-to-SQL | AI 呼び出し

```
input_text = "日本よりも人口密度が高い国は？"

chain = prompt | model

res = chain.invoke({"input": input_text, "table_schema": summary_of_tables})
result = res.content.strip()
print(result)
```

002-001: DuckDB と Text-to-SQL | AI 出力

````
```sql
SELECT
  NAME AS name,
  POP_EST / ST_Area(geom) AS value,
  ST_AsGeoJSON(geom) AS geom
FROM
  countries
WHERE
  POP_EST / ST_Area(geom) > (
    SELECT
      POP_EST / ST_Area(geom)
    FROM
      countries
    WHERE
      NAME = 'Japan'
  );
```
````

002-001: DuckDB と Text-to-SQL | SQL 実行

```
# 正規表現でresultから ``` を使って SQL のみを抽出
import re
match = re.search(r"```[^\n]*\n(.*?)```", result, re.DOTALL)
sql_query = match.group(1).strip()
```

```
duckdb_result = conn.execute(sql_query).fetchall()
duckdb_result
```

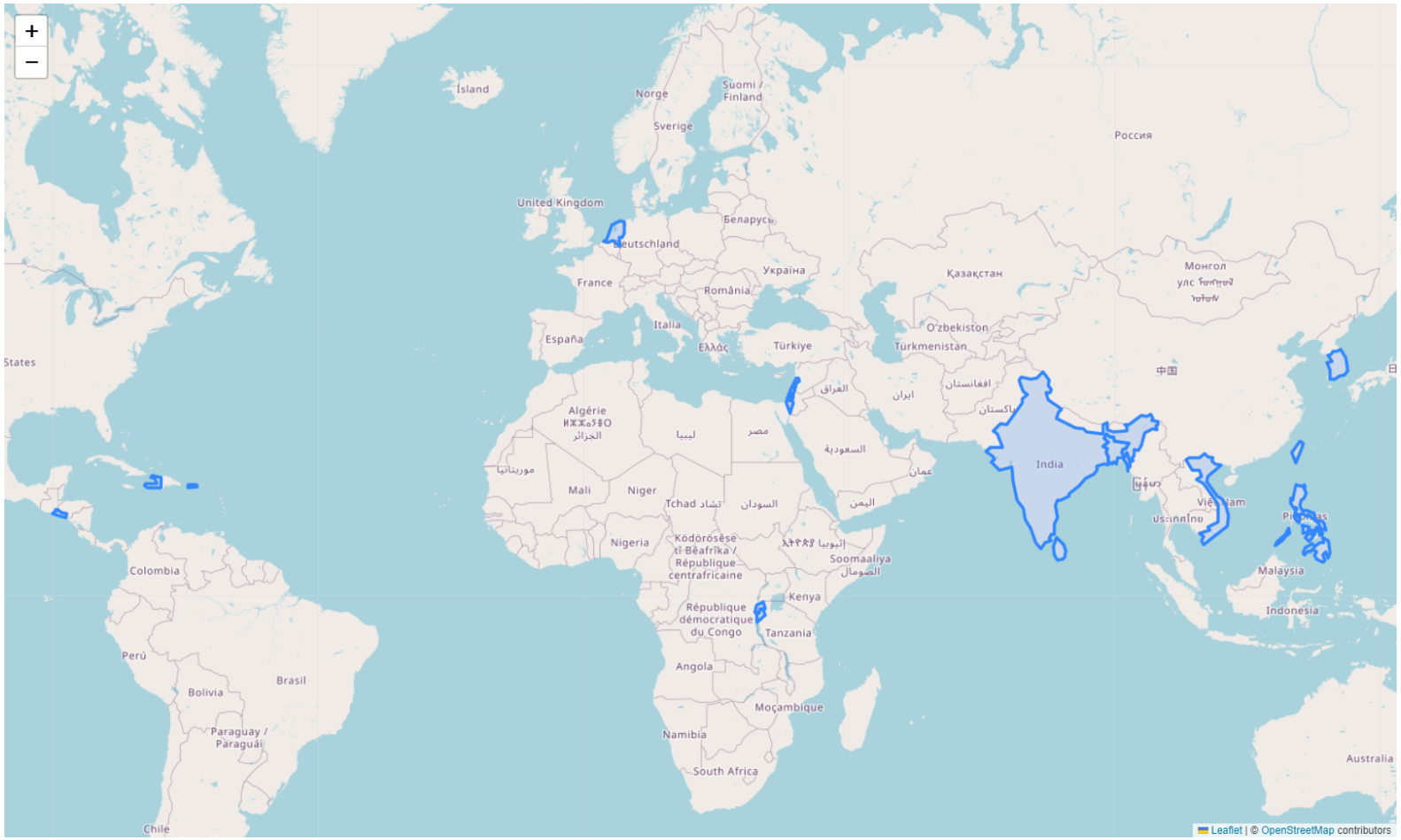
002-001: DuckDB と Text-to-SQL | GeoJSON 変換

```
import json

def duckdb_result_to_geojson(duckdb_result):
    geojson = {
        "type": "FeatureCollection",
        "features": []
    }

    for row in duckdb_result:
        try:
            name, value, geom_str = row
            geom = json.loads(geom_str)
            feature = {
                "type": "Feature",
                "geometry": geom,
                "properties": {
                    "name": name,
                    "value": value
                }
            }
            geojson["features"].append(feature)
        except (ValueError, TypeError, IndexError) as e:
            print(f"Error processing row {row}: {e}")
            return None
        except Exception as e:
            print(f"An unexpected error occurred: {e}")
            return None

    return json.dumps(geojson, indent=2)

geojson_output = duckdb_result_to_geojson(duckdb_result)
geojson_data = json.loads(geojson_output)
```

002-001: DuckDB と Text-to-SQL | 地図上に可視化

```
import folium

m = folium.Map(location=[0, 0], zoom_start=2)

folium.GeoJson(geojson_data).add_to(m)

bounds = folium.GeoJson(geojson_data).get_bounds()
if bounds:
    m.fit_bounds(bounds)

display(m)
```

002-001: DuckDB と Text-to-SQL | 地図上に可視化 結果



002-002: SQL RAG

ゴール

「生成 AI に SQL データベースの情報を踏まえて質問応答させる」

002-002: SQL RAG

ステップ

- DuckDB でデータベースを作成
- 生成 AI で自然言語の質問に答えるための SQL を出力
- DuckDB で SQL を実行
- 自然言語の質問と SQL 実行結果を組み合わせて回答を生成

002-002: SQL RAG | プロンプト 1

```
template = """You are an expert of PostgreSQL and PostGIS.
You output the best PostgreSQL query based on given table schema and input text.

You will always reply according to the following rules:
- Output valid PostgreSQL query.
- The query MUST be line delimited and surrounded by just three back quote to indicate that it is a code block.

** Table Schema: **
{table_schema}

User Input:
{input}
"""

prompt = ChatPromptTemplate.from_template(template)
```

002-002: SQL RAG | AI 呼び出し 1

```
input_text = "日本よりも広い国は世界で何ヶ国ありますか？"

chain = prompt | model
res = chain.invoke({"input": input_text, "table_schema": summary_of_tables})
result = res.content.strip()
```

002-002: SQL RAG | AI 出力 1

````
```sql
SELECT
  COUNT(*)
FROM countries
WHERE
  ST_Area(geom) > (
    SELECT
      ST_Area(geom)
    FROM countries
    WHERE
      NAME = 'Japan'
  );
```
````

002-002: SQL RAG | SQL 実行

```
duckdb_result = conn.execute(query).fetchall()
duckdb_result
```

002-002: SQL RAG | プロンプト 2

```
prompt = (
    "Given the following user question, corresponding SQL query, "
    "and SQL result, answer the user question.\n\n"
    f'Question: {input_text}\n'
    f'SQL Query: {query}\n'
    f'SQL Result: {duckdb_result}\n'
)
```

002-002: SQL RAG | AI 呼び出し 2

```
answer = model.invoke(prompt).content.strip()
print(answer)
```

002-002: SQL RAG | AI 出力 2

日本よりも広い国は世界で59ヶ国あります。

003: ベクトルタイルと地図スタイルカスタマイズ

003: ベクトルタイルと地図スタイルカスタマイズ

ゴール

「生成 AI を使って自然言語による指示に基づいて地図スタイルを変更する」

003: ベクトルタイルと地図スタイルカスタマイズ

ステップ

- Maplibre Style Spec で地図スタイルを定義
- 生成 AI に Maplibre Style Spec と自然言語による指示を入力
    - 生成 AI に Maplibre Style Spec を出力させる

003: ベクトルタイルと地図スタイルカスタマイズ | 地図スタイル定義

Maplibre Style Spec とは

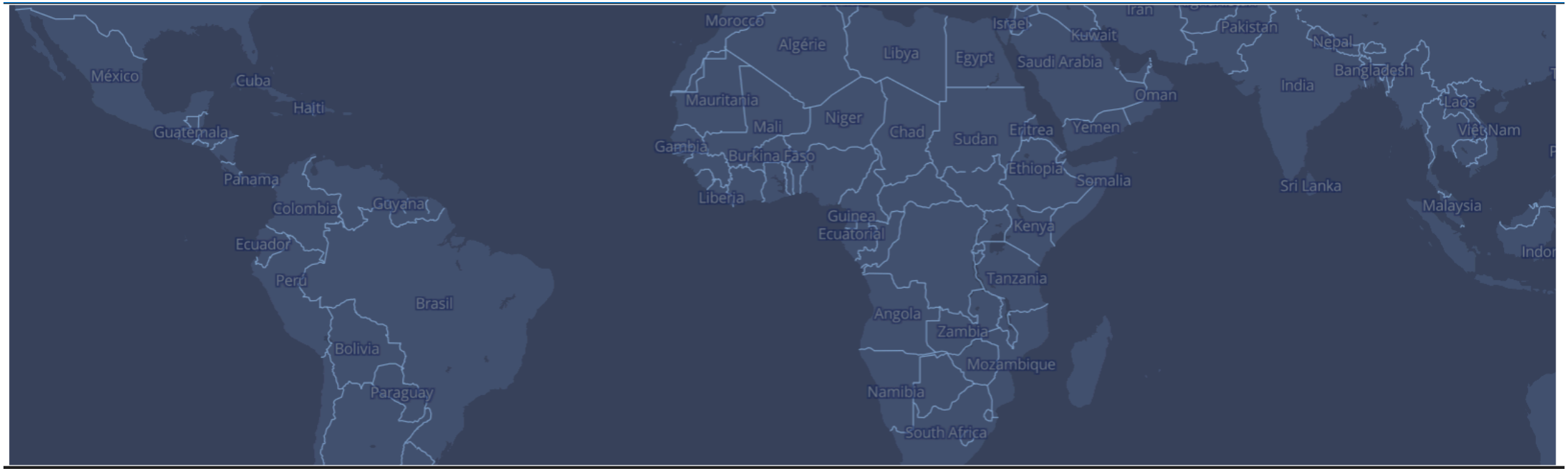
```
{
  "version": 8,
  "sources": {
    "openmaptiles": {
      "type": "vector",
      "url": "https://tile.openstreetmap.jp/data/planet.json"
    }
  },
  "sprite": "https://tile.openstreetmap.jp/styles/osm-bright/sprite",
  "glyphs": "https://tile.openstreetmap.jp/fonts/{fontstack}/{range}.pbf",
  "layers": [
  .....
  ]
}
```

layers には、地図をどう描画するかということが定義されている

003: ベクトルタイルと地図スタイルカスタマイズ | レイヤーの例

```
{
  "id": "place-country-2",
  "type": "symbol",
  "source": "openmaptiles",
  "source-layer": "place",
  "filter": ["all", ["==", "class", "country"]],
  "layout": {
    "text-field": "{name:latin}",
    "text-font": ["Open Sans Regular"],
    "text-max-width": 6.25,
    "text-size": {
      "base": 1,
      "stops": [
        [1, 11],
        [4, 17]
      ]
    }
  },
  "paint": {
    "text-color": "#7d8791",
    "text-halo-blur": 1,
    "text-halo-color": "hsla(228, 60%, 21%, 0.7)",
    "text-halo-width": 1.4
  }
}
```

003: ベクトルタイルと地図スタイルカスタマイズ | Before



© OpenStreetMap contributors

003: ベクトルタイルと地図スタイルカスタマイズ | プロンプト

```
template = """You are an expert of OpenStreetMap and Maplibre GL JS.
You output the best javascript code of map style definition for the given user input.

You will always reply according to the following rules:
- Output valid javascript code.
- The code MUST be line delimited and surrounded by just three back quote to indicate that it is a code block.
- The code MUST be takes into account the user's intent to the greatest extent possible.

** Current style definition: **
{current_style}

User Input:
{input}
"""
prompt = ChatPromptTemplate.from_template(template)
```

003: ベクトルタイルと地図スタイルカスタマイズ | AI 呼び出し

```
input_text = "国の名前を赤色にしてください。"

chain = prompt | model

res = chain.invoke({"input": input_text, "current_style": current_style})
result = res.content.strip()
print(result)
```

003: ベクトルタイルと地図スタイルカスタマイズ | AI 出力

```
{
  "id": "place-country-2",
  "type": "symbol",
  "source": "openmaptiles",
  "source-layer": "place",
  "filter": ["all", ["==", "class", "country"]],
  "layout": {
    "text-field": "{name:latin}",
    "text-font": ["Open Sans Regular"],
    "text-max-width": 6.25,
    "text-size": {
      "base": 1,
      "stops": [
        [1, 11],
        [4, 17]
      ]
    }
  },
  "paint": {
    "text-color": "#ff0000",
    "text-halo-blur": 1,
    "text-halo-color": "hsla(228, 60%, 21%, 0.7)",
    "text-halo-width": 1.4
  }
}
```

003: ベクトルタイルと地図スタイルカスタマイズ | 地図上に可視化 結果

© OpenStreetMap contributors

閑話休題

自然言語と人工言語

自然言語と人工言語 (1)

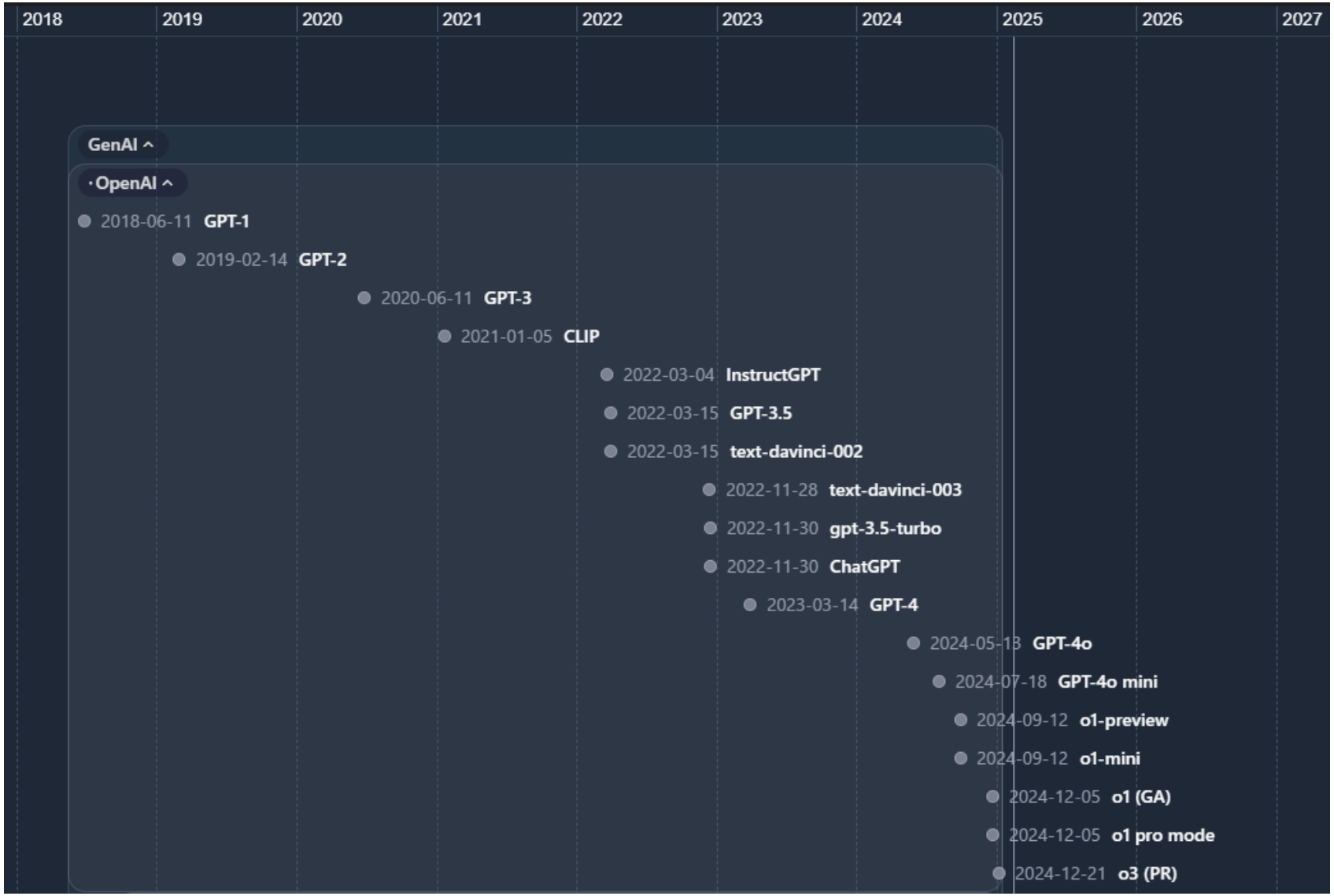
- 人工言語
  - Overpass QL
  - SQL
  - Maplibre Style Spec

自然言語と人工言語 (2)

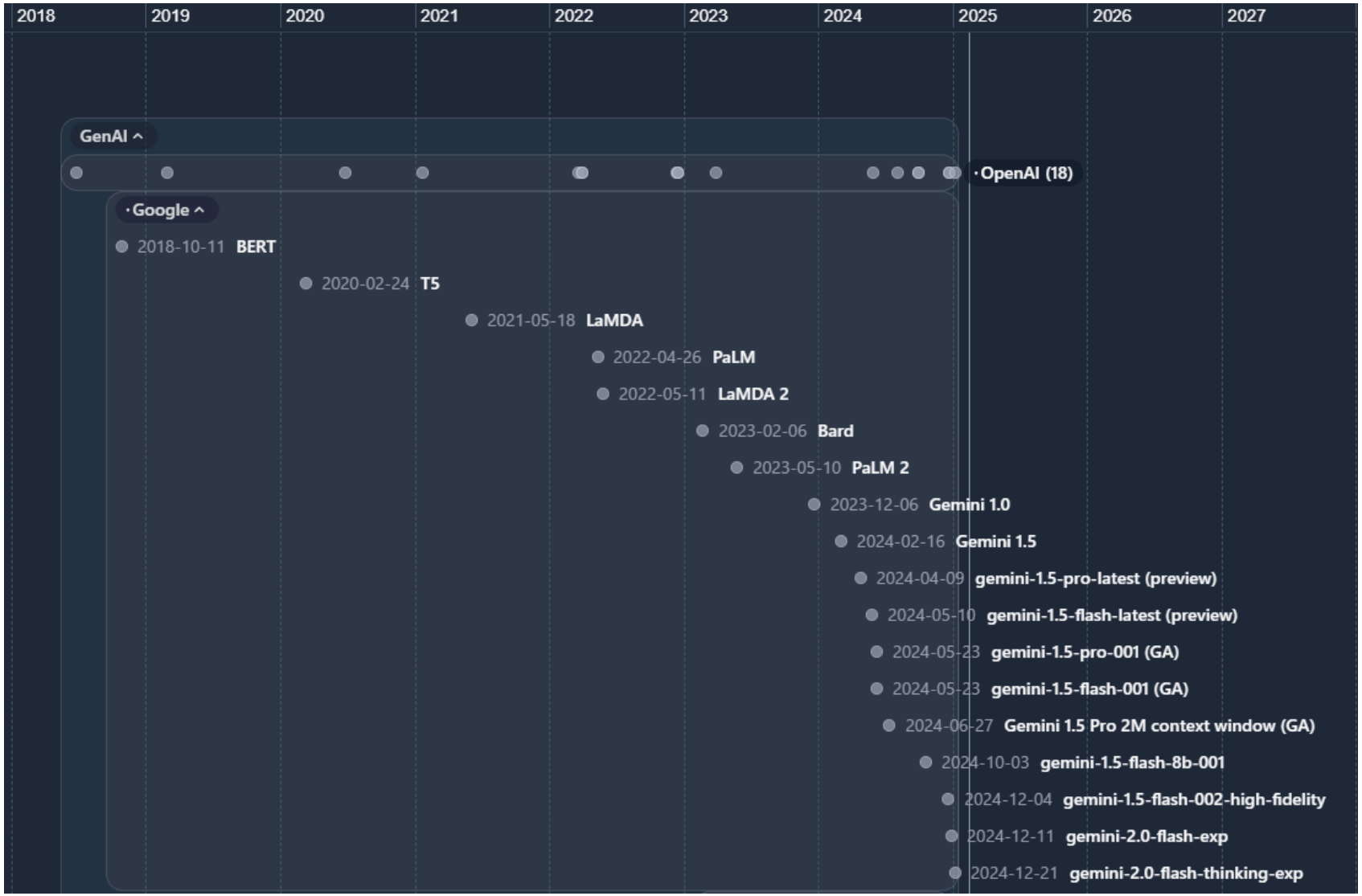
- 人工言語が、計算機やデータセットと、人間の自然言語とを橋渡ししてくれる

研究紹介

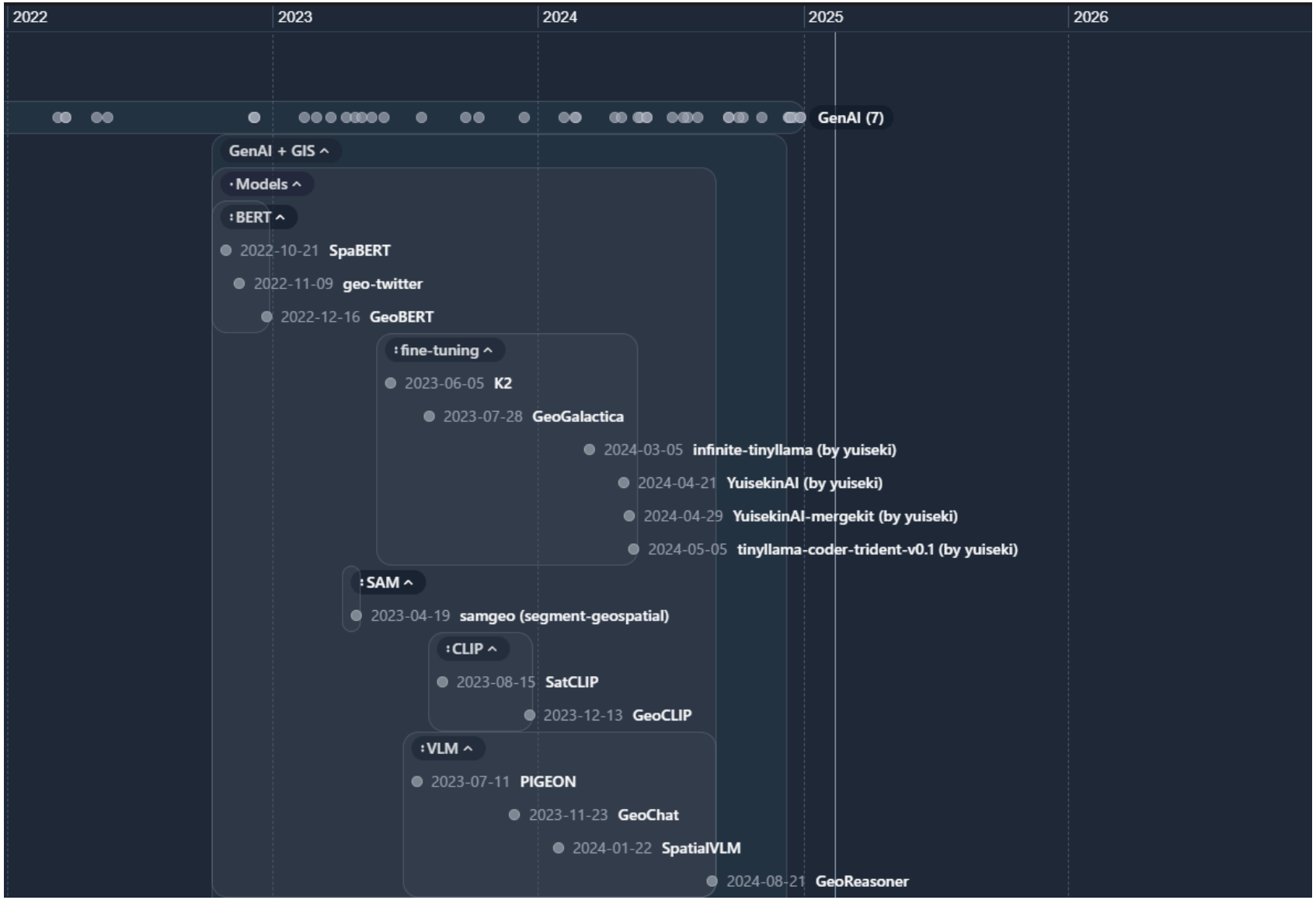
研究紹介 (1)


2018
2019
2020
2021
2022
2023
2024
2025
2026
2027
GenAI
OpenAI
2018-06-11 GPT-1
2019-02-14 GPT-2
2020-06-11 GPT-3
2021-01-05 CLIP
2022-03-04 InstructGPT
2022-03-15 GPT-3.5
2022-03-15 text-davinci-002
2022-11-28 text-davinci-003
2022-11-30 gpt-3.5-turbo
2022-11-30 ChatGPT
2023-03-14 GPT-4
2024-05-13 GPT-4o
2024-07-18 GPT-4o mini
2024-09-12 o1-preview
2024-09-12 o1-mini
2024-12-05 o1 (GA)
2024-12-05 o1 pro mode
2024-12-21 o3 (PR)

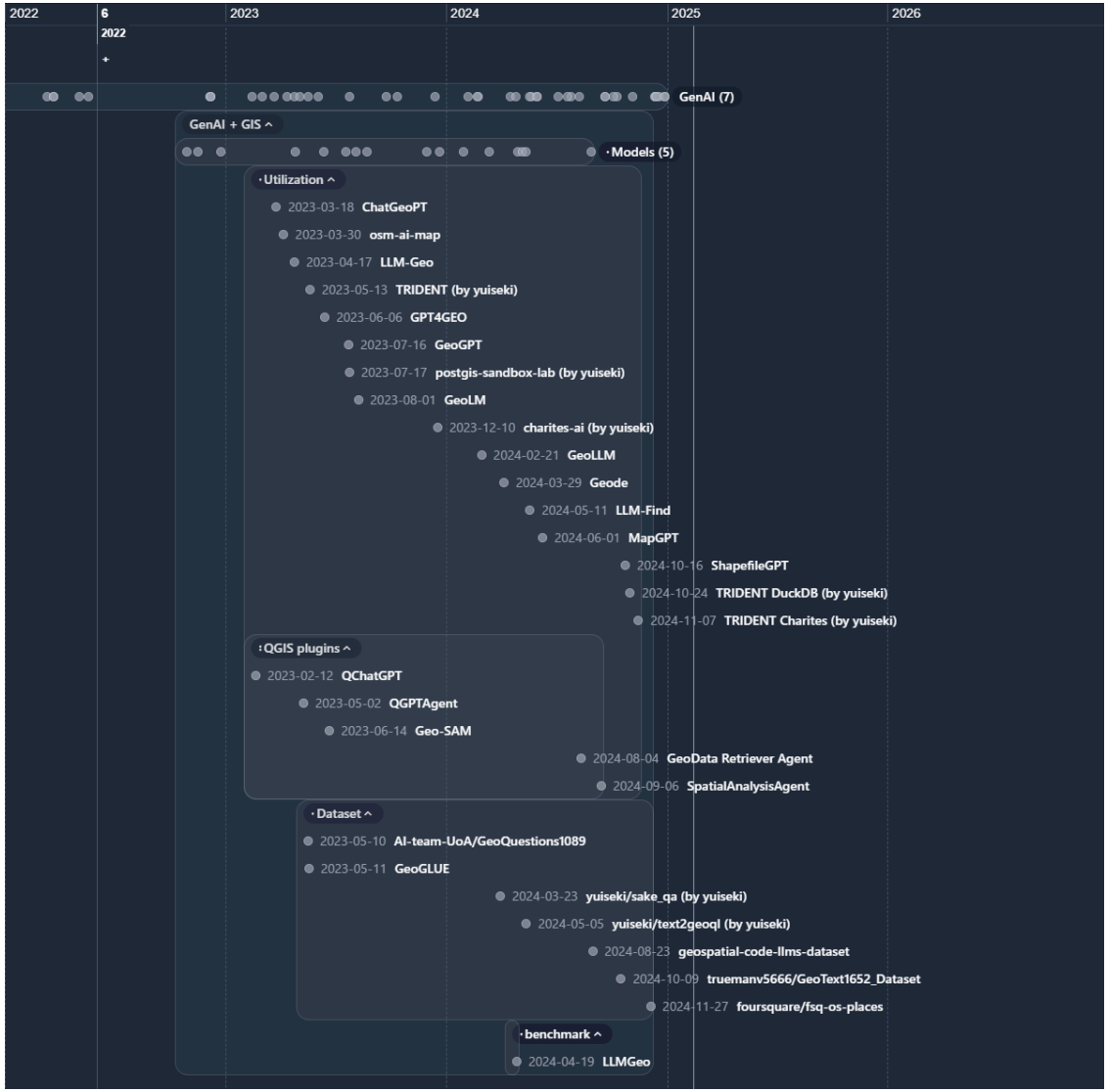
研究紹介 (2)


2018
2019
2020
2021
2022
2023
2024
2025
2026
2027
GenAI
·OpenAI (18)
·Google
2018-10-11 BERT
2020-02-24 T5
2021-05-18 LaMDA
2022-04-26 PaLM
2022-05-11 LaMDA 2
2023-02-06 Bard
2023-05-10 PaLM 2
2023-12-06 Gemini 1.0
2024-02-16 Gemini 1.5
2024-04-09 gemini-1.5-pro-latest (preview)
2024-05-10 gemini-1.5-flash-latest (preview)
2024-05-23 gemini-1.5-pro-001 (GA)
2024-05-23 gemini-1.5-flash-001 (GA)
2024-06-27 Gemini 1.5 Pro 2M context window (GA)
2024-10-03 gemini-1.5-flash-8b-001
2024-12-04 gemini-1.5-flash-002-high-fidelity
2024-12-11 gemini-2.0-flash-exp
2024-12-21 gemini-2.0-flash-thinking-exp

研究紹介 (3)


2022
2023
2024
2025
2026
GenAI (7)
GenAI + GIS
Models
BERT
2022-10-21 SpaBERT
2022-11-09 geo-twitter
2022-12-16 GeoBERT
fine-tuning
2023-06-05 K2
2023-07-28 GeoGalactica
2024-03-05 infinite-tinyllama (by yuiseki)
2024-04-21 YuisekinAI (by yuiseki)
2024-04-29 YuisekinAI-mergekit (by yuiseki)
2024-05-05 tinyllama-coder-trident-v0.1 (by yuiseki)
SAM
2023-04-19 samgeo (segment-geospatial)
CLIP
2023-08-15 SatCLIP
2023-12-13 GeoCLIP
VLM
2023-07-11 PIGEON
2023-11-23 GeoChat
2024-01-22 SpatialVLM
2024-08-21 GeoReasoner

研究紹介 (4)


2022
6
2023
2024
2025
2026
2022
GenAI (7)
GenAI + GIS
Models (5)
Utilization
2023-03-18 ChatGeoPT
2023-03-30 osm-ai-map
2023-04-17 LLM-Geo
2023-05-13 TRIDENT (by yuiseki)
2023-06-06 GPT4GEO
2023-07-16 GeoGPT
2023-07-17 postgis-sandbox-lab (by yuiseki)
2023-08-01 GeoLM
2023-12-10 charites-ai (by yuiseki)
2024-02-21 GeoLLM
2024-03-29 Geode
2024-05-11 LLM-Find
2024-06-01 MapGPT
2024-10-16 ShapefileGPT
2024-10-24 TRIDENT DuckDB (by yuiseki)
2024-11-07 TRIDENT Charites (by yuiseki)
QGIS plugins
2023-02-12 QChatGPT
2023-05-02 QGPTAgent
2023-06-14 Geo-SAM
2024-08-04 GeoData Retriever Agent
2024-09-06 SpatialAnalysisAgent
Dataset
2023-05-10 AI-team-UoA/GeoQuestions1089
2023-05-11 GeoGLUE
2024-03-23 yuiseki/sake_qa (by yuiseki)
2024-05-05 yuiseki/text2geoql (by yuiseki)
2024-08-23 geospatial-code-llms-dataset
2024-10-09 truemanv5666/GeoText1652_Dataset
2024-11-27 foursquare/fsq-os-places
benchmark
2024-04-19 LLMGeo

研究動向

研究動向 (1)

生成 AI (transformer) 以前

- GeoNLP (2011)
- GeoSPARQL (2014)
- LOD (Linked Open Data)
- 単語の埋め込み
  - Word2Vec (2013)
- NER (Named Entity Recognition, 固有表現抽出)
  - spaCy (2015)
  - GiNZA (2019)

研究動向 (2)

生成 AI (transformer) 以降

2022

- 各自が独自のアーキテクチャで pre-training
  - ERNIE-GeoL
- BERT の GIS への応用
  - SpaBERT, GeoBERT

研究動向 (3)

生成 AI (transformer) 以降

2023

- LLM のファインチューニング
  - K2
- VLM
- 生成 AI プラットフォーム + OpenStreetMap
- 生成 AI プラットフォーム + QGIS
  - Autonomous GIS

研究動向 (4)

生成 AI (transformer) 以降

2024

- 既存の基盤モデルからの地理空間情報抽出
- VLM によって GeoGuesser で人間のプレイヤーに匹敵するモデルの登場
- 自律エージェント研究のさらなる進展
  - データの取得まで自動化
- 生成 AI による GIS 開発タスクの遂行と評価
- MapEval: 実用性重視のベンチマークの登場

概観

- 学術研究が先行することもあれば、オープンソースプロジェクトが先行することもあるし、ビジネスのほうが先行することもある
    - これら 3 つは相互に影響を与え合っている

課題と展望

- QGIS をプラットフォームとした自律エージェントの開発が活性化している
  - Docker や Web で誰もが現実的な課題解決のために使える形での自律エージェントのコモディティ化を目指したい
- 基盤モデルや自律エージェントの客観的な性能評価基準がもっと必要
  - GeoGuesser などのゲームを利用した評価はできているが，実際の課題解決や GIS 開発タスクにおける評価は部分的
  - ベンチマークのためのタスク定義や評価用データセットのさらなる充実が必要
- 完全なオープンソースを目指したい

まとめ

まとめ

- 地理情報技術のオープンデータや API やライブラリの使い方を紹介しました
- 生成 AI をソフトウェアシステムのパーツとして扱うテクニックを紹介しました
- この 2 つの組み合わせを紹介しました

質疑応答