

# 妙だな…をLLMに気付かせる

MCPサーバで始めるデータと対話可能なAIの作り方

# 自己紹介



Competitions Expert  
1,808 of 202,437

tonic (@tonic3561)

- 所属：合同会社AlgoSphere 代表
- Kaggle：万年Expert (5x 🏆)
- 専門：時系列予測



# もくじ

# もくじ

## 1. コンペ x 生成AIの現在地

# もくじ

1. コンペ x 生成AIの現在地
2. LLMが直面するデータ分析の壁

# もくじ

1. コンペ x 生成AIの現在地
2. LLMが直面するデータ分析の壁
3. LLMにデータを「触らせる」挑戦

# もくじ

1. コンペ x 生成AIの現在地
2. LLMが直面するデータ分析の壁
3. LLMにデータを「触らせる」挑戦
4. デモ：エージェントによる自律的データ探索

# もくじ

1. コンペ x 生成AIの現在地
2. LLMが直面するデータ分析の壁
3. LLMにデータを「触らせる」挑戦
4. デモ：エージェントによる自律的データ探索
5. まとめと展望

# 1. コンペ × 生成AIの現在地

# LLMは既に優秀なアシスタント

データ分析コンペにおいて、LLMは様々な場面で活躍



# LLMは既に優秀なアシスタント

データ分析コンペにおいて、LLMは様々な場面で活躍

- 問題設定やデータの説明



# LLMは既に優秀なアシスタント

データ分析コンペにおいて、LLMは様々な場面で活躍

- 問題設定やデータの説明
- 仮説・アイディアの壁打ち



# LLMは既に優秀なアシスタント

データ分析コンペにおいて、LLMは様々な場面で活躍

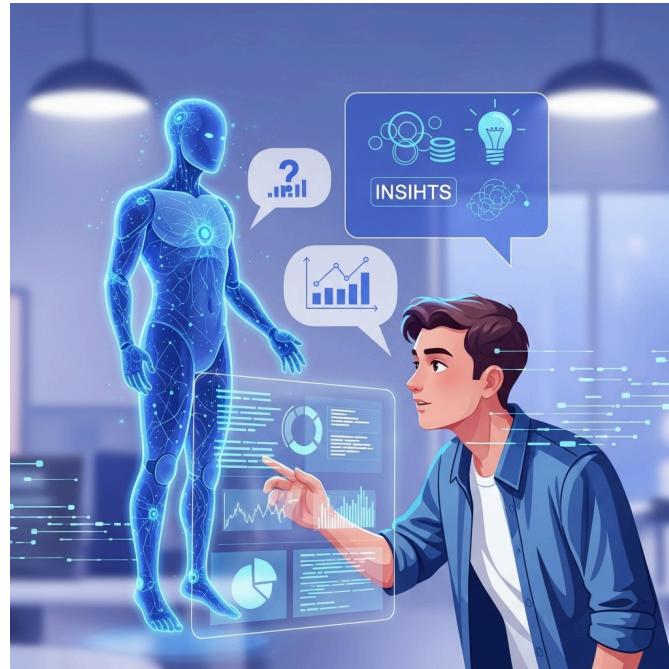
- 問題設定やデータの説明
- 仮説・アイディアの壁打ち
- 関連論文の調査



# LLMは既に優秀なアシスタント

データ分析コンペにおいて、LLMは様々な場面で活躍

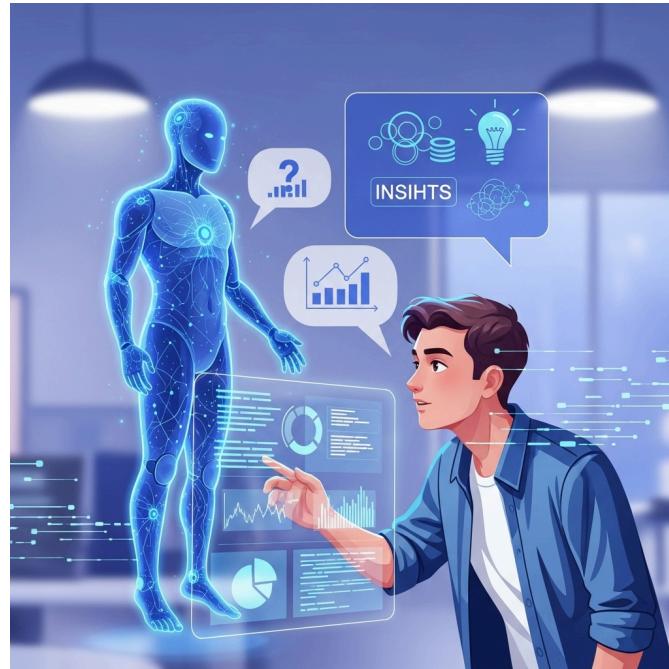
- 問題設定やデータの説明
- 仮説・アイディアの壁打ち
- 関連論文の調査
- アイディアの実装



# LLMは既に優秀なアシスタント

データ分析コンペにおいて、LLMは様々な場面で活躍

- 問題設定やデータの説明
- 仮説・アイディアの壁打ち
- 関連論文の調査
- アイディアの実装
- バグの原因特定



# LLMは既に優秀なアシスタント

データ分析コンペにおいて、LLMは様々な場面で活躍

- 問題設定やデータの説明
- 仮説・アイディアの壁打ち
- 関連論文の調査
- アイディアの実装
- バグの原因特定
- ベースラインの構築



# だが…あと一歩届かない！

- 仮説・アイディアの壁打ち ← ココ

# だが…あと一歩届かない！

- 仮説・アイディアの壁打ち ← ココ

仮説を立てるためにはデータをよく見るのが一番大切（？）

チャットベースのLLMではデータを「触る」ことが難しく、壁打ちくらいが限界

## 2. LLMが直面する データ分析の壁

# (人間が) 仮説・アイディアを考えるプロセス

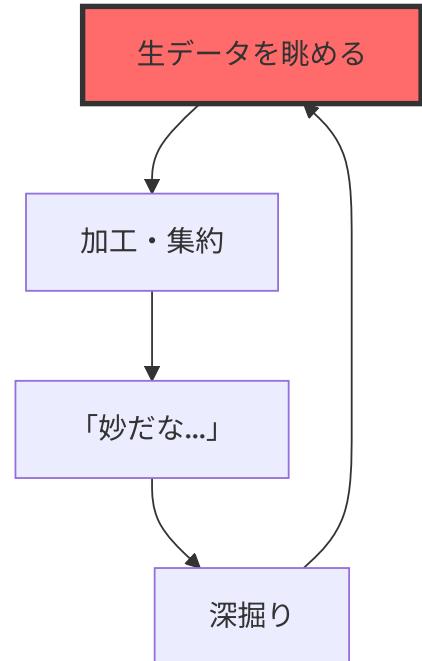
# (人間が) 仮説・アイディアを考えるプロセス

(1) EDA型

(2) 仮説駆動型

# (人間が) 仮説・アイディアを考えるプロセス

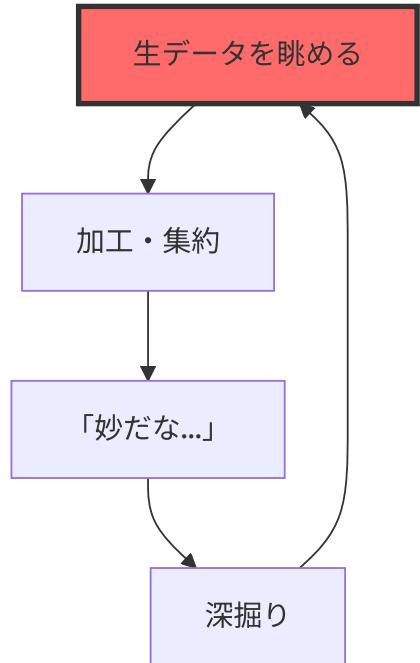
(1) EDA型



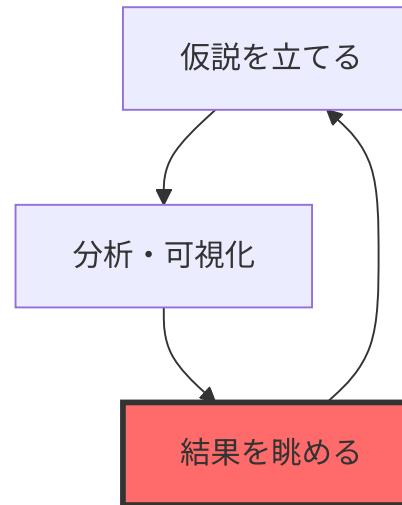
(2) 仮説駆動型

# (人間が) 仮説・アイディアを考えるプロセス

(1) EDA型

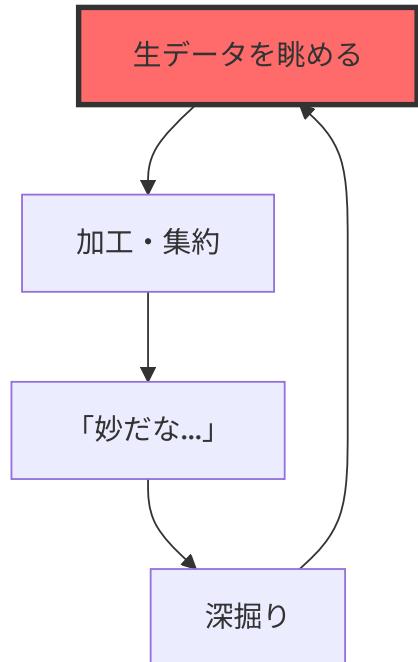


(2) 仮説駆動型

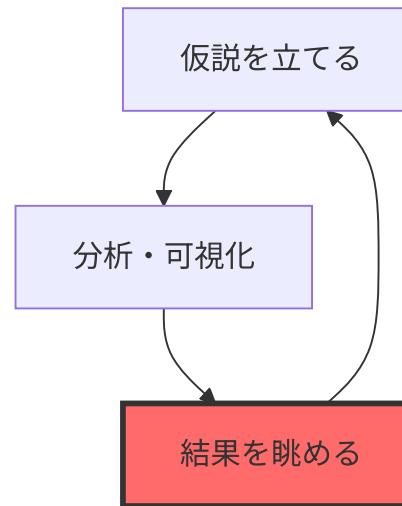


# (人間が) 仮説・アイディアを考えるプロセス

## (1) EDA型



## (2) 仮説駆動型



⇒ 生データや可視化結果を眺めながら思考を回す

# チャットLLMが直面するデータ分析の3つの壁

# チャットLLMが直面するデータ分析の3つの壁

## (1) コンテキストの壁

- 巨大なデータはそもそも読み  
込んでもらえない
- コンテキストを維持するのも  
大変



# チャットLLMが直面するデータ分析の3つの壁

## (1) コンテキストの壁

- 巨大なデータはそもそも読み込んでもらえない
- コンテキストを維持するのも大変

## (2) 作業の壁

- LLMが生成したコードを、人間がコピペして実行
- 実行結果のファイル（CSVや画像）を、人間が確認
- その結果やファイルを、人間がLLMに再入力



# チャットLLMが直面するデータ分析の3つの壁

## (1) コンテキストの壁

- 巨大なデータはそもそも読み込んでもらえない
- コンテキストを維持するのも大変



## (2) 作業の壁

- LLMが生成したコードを、人間がコピペして実行
- 実行結果のファイル（CSVや画像）を、人間が確認
- その結果やファイルを、人間がLLMに再入力



## (3) 思考の壁

- 分析の「気づき」は、データを多角的に眺める中で生まれる
- LLMは人間が与えた断片的な結果しか見ることができない？



### 3. LLMにデータを 「触らせる」挑戦

# LLMにデータを触ってもらうには？

LLMが自律的に分析を進めるために必要なタスクを分解してみる

# LLMにデータを触ってもらうには？

LLMが自律的に分析を進めるために必要なタスクを分解してみる

- 問題設計・データ構造を理解する

# LLMにデータを触ってもらうには？

LLMが自律的に分析を進めるために必要なタスクを分解してみる

- 問題設計・データ構造を理解する
- 集約・可視化コードを書く

# LLMにデータを触ってもらうには？

LLMが自律的に分析を進めるために必要なタスクを分解してみる

- 問題設計・データ構造を理解する
- 集約・可視化コードを書く
- 結果をファイル出力する

# LLMにデータを触ってもらうには？

LLMが自律的に分析を進めるために必要なタスクを分解してみる

- 問題設計・データ構造を理解する
- 集約・可視化コードを書く
- 結果をファイル出力する
- ファイル（csv, png）を読み込む

# LLMにデータを触ってもらうには？

LLMが自律的に分析を進めるために必要なタスクを分解してみる

- 問題設計・データ構造を理解する
- 集約・可視化コードを書く
- 結果をファイル出力する
- ファイル（csv, png）を読み込む
- 読み込んだテキストや画像を解釈する

# LLMにデータを触ってもらうには？

LLMが自律的に分析を進めるために必要なタスクを分解してみる

- 問題設計・データ構造を理解する
- 集約・可視化コードを書く
- 結果をファイル出力する
- ファイル（csv, png）を読み込む
- 読み込んだテキストや画像を解釈する
- 上記を繰り返す

# LLMにデータを触ってもらうには？

LLMが自律的に分析を進めるために必要なタスクを分解してみる

- 問題設計・データ構造を理解する →  できる
- 集約・可視化コードを書く
- 結果をファイル出力する
- ファイル（csv, png）を読み込む
- 読み込んだテキストや画像を解釈する
- 上記を繰り返す

# LLMにデータを触ってもらうには？

LLMが自律的に分析を進めるために必要なタスクを分解してみる

- 問題設計・データ構造を理解する → できる
- 集約・可視化コードを書く → できる
- 結果をファイル出力する
- ファイル（csv, png）を読み込む
- 読み込んだテキストや画像を解釈する
- 上記を繰り返す

# LLMにデータを触ってもらうには？

LLMが自律的に分析を進めるために必要なタスクを分解してみる

- 問題設計・データ構造を理解する →  できる
- 集約・可視化コードを書く →  できる
- 結果をファイル出力する →  できる
- ファイル（csv, png）を読み込む
- 読み込んだテキストや画像を解釈する
- 上記を繰り返す

# LLMにデータを触ってもらうには？

LLMが自律的に分析を進めるために必要なタスクを分解してみる

- 問題設計・データ構造を理解する → できる
- 集約・可視化コードを書く → できる
- 結果をファイル出力する → できる
- ファイル（csv, png）を読み込む → できる
- 読み込んだテキストや画像を解釈する
- 上記を繰り返す

# LLMにデータを触ってもらうには？

LLMが自律的に分析を進めるために必要なタスクを分解してみる

- 問題設計・データ構造を理解する →  できる
- 集約・可視化コードを書く →  できる
- 結果をファイル出力する →  できる
- ファイル（csv, png）を読み込む →  できる
- 読み込んだテキストや画像を解釈する →  できる
- 上記を繰り返す

# LLMにデータを触ってもらうには？

LLMが自律的に分析を進めるために必要なタスクを分解してみる

- 問題設計・データ構造を理解する → できる
- 集約・可視化コードを書く → できる
- 結果をファイル出力する → できる
- ファイル（csv, png）を読み込む → できる
- 読み込んだテキストや画像を解釈する → できる
- 上記を繰り返す → できる

# LLMにデータを触ってもらうには？

LLMが自律的に分析を進めるために必要なタスクを分解してみる

- 問題設計・データ構造を理解する → できる
- 集約・可視化コードを書く → できる
- 結果をファイル出力する → できる
- ファイル（csv, png）を読み込む → できる
- 読み込んだテキストや画像を解釈する → できる
- 上記を繰り返す → できる

⇒ 個別のタスクはすべて実現可能。

# LLMにデータを触ってもらうには？

LLMが自律的に分析を進めるために必要なタスクを分解してみる

- 問題設計・データ構造を理解する →  できる
- 集約・可視化コードを書く →  できる
- 結果をファイル出力する →  できる
- ファイル（csv, png）を読み込む →  できる
- 読み込んだテキストや画像を解釈する →  できる
- 上記を繰り返す →  できる

⇒ 個別のタスクはすべて実現可能。

あれれ～？

# 作ってみました

claude codeとMCPサーバを活用して、自律的にEDAを行うエージェントを作りました

- 題材：Predict Droughts using Weather & Soil Data
- Gitリポジトリ：[https://github.com/jintonic3561/comp\\_with\\_agent](https://github.com/jintonic3561/comp_with_agent)

# 題材紹介：U.S. Drought Prediction

気象・土壌データから、専門家が作成する「干ばつマップ」の自動化を目指す予測タスク

## 入力データ

### 時系列データ（気象）

- 日々の気温、降水量、風速など
- 時系列方法に train / valid / test を分割

### 静的データ（地理・土壌）

- 郡ごとの標高、傾斜、土地の種類など
- fips（郡ID）で時系列データに左結合

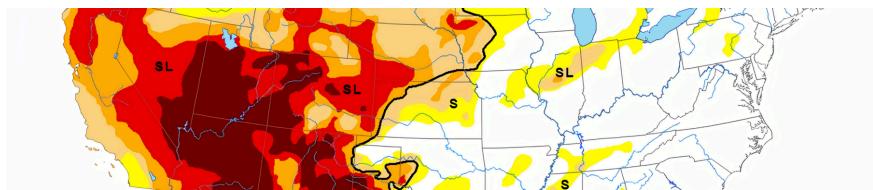
## 予測タスク

### 目的変数

- 週間の干ばつ深刻度を表す 0~5 の6段階カテゴリ

### 評価指標

- Macro F1 Score
- MAE



# 補足…MCPサーバとは？

ざっくり： 簡単に渡せる電卓みたいなもん

LLM（と人間）は暗算が苦手

→ 人間と同じように、LLMにも電卓とか渡してあげればいいんじゃない？

→ ほかにもいろんなツールを使ってもらおう！



- 電卓
- ファイルアクセス
- Web検索
- ...

# 構築したMCPサーバ一覧

LLMにデータを触ってもらうために、3つのMCPサーバを作りました

MCPサーバ名	役割	説明
Data Information	設計情報DB	問題設計や利用可能なデータの説明、列定義などの取得
Analysis Executor	分析実行環境	LLMから渡されたコードを実行し、その結果を返す
Notebook Writer	レポート作成	実行したコードと結果、考察をnotebookにまとめる

※ 各MCPサーバの中に、複数の「ツール」が実装されている

# ツール① DI.get\_data\_description

データの列定義等の情報に立ち返るためのツール

```
def get_data_description(
    data_type: Literal["timeseries", "soil_data"]
) -> str:
    if data_type == "timeseries":
        return """
# データ概要
このデータは、米国の干ばつ...

# 列定義
| 列名 | データ型 | 説明 |
| --- | --- | --- |
| fips | int | 米国郡のFIPSコード |
...
"""

    elif data_type == "soil_data":
        ...
...
```

# ツール① DI.get\_data\_description

データの列定義等の情報に立ち返るためのツール

```
def get_data_description(
    data_type: Literal["timeseries", "soil_data"]
) → str:
    if data_type == "timeseries":
        return """
# データ概要
このデータは、米国の干ばつ...

# 列定義
| 列名 | データ型 | 説明 |
| --- | --- | --- |
| fips | int | 米国郡のFIPSコード |
"""

    elif data_type == "soil_data":
        ...
    ...

```

## ツール① DI.get\_data\_description

データの列定義等の情報に立ち返るためのツール

```
def get_data_description():
    data_type: Literal["timeseries", "soil_data"]
) → str:
    if data_type == "timeseries":
        return """
# データ概要
このデータは、米国の干ばつ...

# 列定義
| 列名 | データ型 | 説明 |
| --- | --- | --- |
| fips | int | 米国郡のFIPSコード |
...
"""

    elif data_type == "soil_data":
        ...
...
```

# ツール① DI.get\_data\_description

データの列定義等の情報に立ち返るためのツール

```
def get_data_description(
    data_type: Literal["timeseries", "soil_data"]
) → str:
    if data_type == "timeseries":
        return """
# データ概要
このデータは、米国の干ばつ...

# 列定義
| 列名 | データ型 | 説明 |
| --- | --- | --- |
| fips | int | 米国郡のFIPSコード |
"""

    elif data_type == "soil_data":
        ...
    ...

```

# ツール① DI.get\_data\_description

データの列定義等の情報に立ち返るためのツール

```
def get_data_description(
    data_type: Literal["timeseries", "soil_data"]
) → str:
    if data_type == "timeseries":
        return """
# データ概要
このデータは、米国の干ばつ...

# 列定義
| 列名 | データ型 | 説明 |
| --- | --- | --- |
| fips | int | 米国郡のFIPSコード |

...
"""

    elif data_type == "soil_data":
        ...
    ...
```

# ツール① DI.get\_data\_description

データの列定義等の情報に立ち返るためのツール

```
def get_data_description(
    data_type: Literal["timeseries", "soil_data"]
) -> str:
    if data_type == "timeseries":
        return """
# データ概要
このデータは、米国の干ばつ...

# 列定義
| 列名 | データ型 | 説明 |
| --- | --- | --- |
| fips | int | 米国郡のFIPSコード |
...
"""

    elif data_type == "soil_data":
        ...
...
```

# ツール② AE.execute\_timeseries\_analysis

LLMが生成した関数を実行し、結果を返すためのツール

```
def execute_timeseries_analysis(  
    func_string: str,  
    data_type: Literal["train", "validation", "test"]  
) -> List[str]:  
    """
```

関数定義は以下の要件を満たすように実装すること：

- 引数として `df: pd.DataFrame` のみを取ること
- dfには `'{data\_type}\_timeseries.csv` の内容が格納される
- csvまたはpng形式の分析結果を関数内でファイル保存すること
- 関数の返り値は保存した分析結果のpath、またはそのリストとすること

このツールを呼び出した後は、分析結果を必ず読み込んで考察を行うこと

```
"""
```

```
# 実行する関数に渡すデータを読み込む
```

```
df = load_data(data_type)
```

```
# LLMが生成した関数を実行し、結果を返却
```

```
return _execute_function(func_string, df)
```

# ツール② AE.execute\_timeseries\_analysis

LLMが生成した関数を実行し、結果を返すためのツール

```
def execute_timeseries_analysis(  
    func_string: str,  
    data_type: Literal["train", "validation", "test"]  
) -> List[str]:  
    """
```

関数定義は以下の要件を満たすように実装すること：

- 引数として `df: pd.DataFrame` のみを取ること
- dfには `'{data\_type}\_timeseries.csv` の内容が格納される
- csvまたはpng形式の分析結果を関数内でファイル保存すること
- 関数の返り値は保存した分析結果のpath、またはそのリストとすること

このツールを呼び出した後は、分析結果を必ず読み込んで考察を行うこと

```
"""
```

```
# 実行する関数に渡すデータを読み込む
```

```
df = load_data(data_type)
```

```
# LLMが生成した関数を実行し、結果を返却
```

```
return _execute_function(func_string, df)
```

# ツール② AE.execute\_timeseries\_analysis

LLMが生成した関数を実行し、結果を返すためのツール

```
def execute_timeseries_analysis(  
    func_string: str,  
    data_type: Literal["train", "validation", "test"]  
) -> List[str]:  
    """
```

関数定義は以下の要件を満たすように実装すること：

- 引数として `df: pd.DataFrame` のみを取ること
- dfには `'{data\_type}\_timeseries.csv` の内容が格納される
- csvまたはpng形式の分析結果を関数内でファイル保存すること
- 関数の返り値は保存した分析結果のpath、またはそのリストとすること

このツールを呼び出した後は、分析結果を必ず読み込んで考察を行うこと

```
"""
```

```
# 実行する関数に渡すデータを読み込む  
df = load_data(data_type)  
  
# LLMが生成した関数を実行し、結果を返却  
return _execute_function(func_string, df)
```

# ツール② AE.execute\_timeseries\_analysis

LLMが生成した関数を実行し、結果を返すためのツール

```
def execute_timeseries_analysis(
    func_string: str,
    data_type: Literal["train", "validation", "test"]
) -> List[str]:
    """
    関数定義は以下の要件を満たすように実装すること：
    - 引数として `df: pd.DataFrame` のみを取ること
    - dfには `'{data_type}_timeseries.csv` の内容が格納される
    - csvまたはpng形式の分析結果を関数内でファイル保存すること
    - 関数の返り値は保存した分析結果のpath、またはそのリストとすること
    このツールを呼び出した後は、分析結果を必ず読み込んで考察を行うこと
    """

    # 実行する関数に渡すデータを読み込む
    df = load_data(data_type)

    # LLMが生成した関数を実行し、結果を返却
    return _execute_function(func_string, df)
```

# ツール② AE.execute\_timeseries\_analysis

LLMが生成した関数を実行し、結果を返すためのツール

```
def execute_timeseries_analysis(  
    func_string: str,  
    data_type: Literal["train", "validation", "test"]  
) -> List[str]:  
    """
```

関数定義は以下の要件を満たすように実装すること：

- 引数として `df: pd.DataFrame` のみを取ること
- dfには `'{data\_type}\_timeseries.csv` の内容が格納される
- csvまたはpng形式の分析結果を関数内でファイル保存すること
- 関数の返り値は保存した分析結果のpath、またはそのリストとすること

このツールを呼び出した後は、分析結果を必ず読み込んで考察を行うこと

```
"""
```

```
# 実行する関数に渡すデータを読み込む
```

```
df = load_data(data_type)
```

```
# LLMが生成した関数を実行し、結果を返却
```

```
return _execute_function(func_string, df)
```

# ツール② AE.execute\_timeseries\_analysis

LLMが生成した関数を実行し、結果を返すためのツール

```
def execute_timeseries_analysis(  
    func_string: str,  
    data_type: Literal["train", "validation", "test"]  
) -> List[str]:  
    """
```

関数定義は以下の要件を満たすように実装すること：

- 引数として `df: pd.DataFrame` のみを取ること
- dfには `{data\_type}\_timeseries.csv` の内容が格納される
- csvまたはpng形式の分析結果を関数内でファイル保存すること
- 関数の返り値は保存した分析結果のpath、またはそのリストとすること

このツールを呼び出した後は、分析結果を必ず読み込んで考察を行うこと

```
"""
```

```
# 実行する関数に渡すデータを読み込む
```

```
df = load_data(data_type)
```

```
# LLMが生成した関数を実行し、結果を返却
```

```
return _execute_function(func_string, df)
```

# ツール② AE.execute\_timeseries\_analysis

LLMが生成した関数を実行し、結果を返すためのツール

```
def execute_timeseries_analysis(  
    func_string: str,  
    data_type: Literal["train", "validation", "test"]  
) -> List[str]:  
    """
```

関数定義は以下の要件を満たすように実装すること：

- 引数として `df: pd.DataFrame` のみを取ること
- dfには `{data\_type}\_timeseries.csv` の内容が格納される
- csvまたはpng形式の分析結果を関数内でファイル保存すること
- 関数の返り値は保存した分析結果のpath、またはそのリストとすること

このツールを呼び出した後は、分析結果を必ず読み込んで考察を行うこと

```
"""
```

```
# 実行する関数に渡すデータを読み込む
```

```
df = load_data(data_type)
```

```
# LLMが生成した関数を実行し、結果を返却
```

```
return _execute_function(func_string, df)
```

# ツール② AE.execute\_timeseries\_analysis

LLMが生成した関数を実行し、結果を返すためのツール

```
def execute_timeseries_analysis(  
    func_string: str,  
    data_type: Literal["train", "validation", "test"]  
) -> List[str]:  
    """
```

関数定義は以下の要件を満たすように実装すること：

- 引数として `df: pd.DataFrame` のみを取ること
- dfには `'{data\_type}\_timeseries.csv` の内容が格納される
- csvまたはpng形式の分析結果を関数内でファイル保存すること
- 関数の返り値は保存した分析結果のpath、またはそのリストとすること

このツールを呼び出した後は、分析結果を必ず読み込んで考察を行うこと

```
"""
```

```
# 実行する関数に渡すデータを読み込む
```

```
df = load_data(data_type)
```

```
# LLMが生成した関数を実行し、結果を返却
```

```
return _execute_function(func_string, df)
```

# ツール② AE.execute\_timeseries\_analysis

LLMが生成した関数を実行し、結果を返すためのツール

```
def execute_timeseries_analysis(  
    func_string: str,  
    data_type: Literal["train", "validation", "test"]  
) -> List[str]:  
    """
```

関数定義は以下の要件を満たすように実装すること：

- 引数として `df: pd.DataFrame` のみを取ること
- dfには `'{data\_type}\_timeseries.csv` の内容が格納される
- csvまたはpng形式の分析結果を関数内でファイル保存すること
- 関数の返り値は保存した分析結果のpath、またはそのリストとすること

このツールを呼び出した後は、分析結果を必ず読み込んで考察を行うこと

```
    """  
  
    # 実行する関数に渡すデータを読み込む  
    df = load_data(data_type)  
  
    # LLMが生成した関数を実行し、結果を返却  
    return _execute_function(func_string, df)
```

# ツール② AE.execute\_timeseries\_analysis

LLMが生成した関数を実行し、結果を返すためのツール

```
def execute_timeseries_analysis(  
    func_string: str,  
    data_type: Literal["train", "validation", "test"]  
) -> List[str]:  
    """
```

関数定義は以下の要件を満たすように実装すること：

- 引数として `df: pd.DataFrame` のみを取ること
- dfには `'{data\_type}\_timeseries.csv` の内容が格納される
- csvまたはpng形式の分析結果を関数内でファイル保存すること
- 関数の返り値は保存した分析結果のpath、またはそのリストとすること

このツールを呼び出した後は、分析結果を必ず読み込んで考察を行うこと

```
"""
```

```
# 実行する関数に渡すデータを読み込む
```

```
df = load_data(data_type)
```

```
# LLMが生成した関数を実行し、結果を返却
```

```
return _execute_function(func_string, df)
```

# ツール② AE.execute\_timeseries\_analysis

LLMが生成した関数を実行し、結果を返すためのツール

```
def execute_timeseries_analysis(  
    func_string: str,  
    data_type: Literal["train", "validation", "test"]  
) -> List[str]:  
    """
```

関数定義は以下の要件を満たすように実装すること：

- 引数として `df: pd.DataFrame` のみを取ること
- dfには `'{data\_type}\_timeseries.csv` の内容が格納される
- csvまたはpng形式の分析結果を関数内でファイル保存すること
- 関数の返り値は保存した分析結果のpath、またはそのリストとすること

このツールを呼び出した後は、分析結果を必ず読み込んで考察を行うこと

```
"""
```

```
# 実行する関数に渡すデータを読み込む
```

```
df = load_data(data_type)
```

```
# LLMが生成した関数を実行し、結果を返却
```

```
return _execute_function(func_string, df)
```

# ツール② AE.execute\_timeseries\_analysis

LLMが生成した関数を実行し、結果を返すためのツール

```
def execute_timeseries_analysis(  
    func_string: str,  
    data_type: Literal["train", "validation", "test"]  
) -> List[str]:  
    """
```

関数定義は以下の要件を満たすように実装すること：

- 引数として `df: pd.DataFrame` のみを取ること
- dfには `'{data\_type}\_timeseries.csv` の内容が格納される
- csvまたはpng形式の分析結果を関数内でファイル保存すること
- 関数の返り値は保存した分析結果のpath、またはそのリストとすること

このツールを呼び出した後は、分析結果を必ず読み込んで考察を行うこと

```
"""
```

```
# 実行する関数に渡すデータを読み込む  
df = load_data(data_type)
```

```
# LLMが生成した関数を実行し、結果を返却  
return _execute_function(func_string, df)
```

# ツール② AE.execute\_timeseries\_analysis

LLMが生成した関数を実行し、結果を返すためのツール

```
def execute_timeseries_analysis(  
    func_string: str,  
    data_type: Literal["train", "validation", "test"]  
) -> List[str]:  
    """
```

関数定義は以下の要件を満たすように実装すること：

- 引数として `df: pd.DataFrame` のみを取ること
- dfには `'{data\_type}\_timeseries.csv` の内容が格納される
- csvまたはpng形式の分析結果を関数内でファイル保存すること
- 関数の返り値は保存した分析結果のpath、またはそのリストとすること

このツールを呼び出した後は、分析結果を必ず読み込んで考察を行うこと

```
"""
```

```
# 実行する関数に渡すデータを読み込む
```

```
df = load_data(data_type)
```

```
# LLMが生成した関数を実行し、結果を返却
```

```
return _execute_function(func_string, df)
```

# ツール③ NW.add\_cell\_to\_notebook

分析のコードや結果をJupyter Notebookに逐次記録するツール

```
def add_cell_to_notebook(
    content: str,
    cell_type: Literal["code", "markdown"],
    artifact_paths: List[str]
):
    # セルを追加する
    nb = load_or_create_notebook()
    cell = create_cell(content, cell_type)
    nb.cells.append(cell)

    # 出力ファイルを表示する
    for path in artifact_paths:
        # テキストを表示
        if path.endswith(".csv"):
            display_csv(path)
        # 画像を表示
        elif path.endswith(".png"):
            display_image(path)
```

# ツール③ NW.add\_cell\_to\_notebook

分析のコードや結果をJupyter Notebookに逐次記録するツール

```
def add_cell_to_notebook(
    content: str,
    cell_type: Literal["code", "markdown"],
    artifact_paths: List[str]
):
    # セルを追加する
    nb = load_or_create_notebook()
    cell = create_cell(content, cell_type)
    nb.cells.append(cell)

    # 出力ファイルを表示する
    for path in artifact_paths:
        # テキストを表示
        if path.endswith(".csv"):
            display_csv(path)
        # 画像を表示
        elif path.endswith(".png"):
            display_image(path)
```

# ツール③ NW.add\_cell\_to\_notebook

分析のコードや結果をJupyter Notebookに逐次記録するツール

```
def add_cell_to_notebook(
    content: str,
    cell_type: Literal["code", "markdown"],
    artifact_paths: List[str]
):
    # セルを追加する
    nb = load_or_create_notebook()
    cell = create_cell(content, cell_type)
    nb.cells.append(cell)

    # 出力ファイルを表示する
    for path in artifact_paths:
        # テキストを表示
        if path.endswith(".csv"):
            display_csv(path)
        # 画像を表示
        elif path.endswith(".png"):
            display_image(path)
```

# ツール③ NW.add\_cell\_to\_notebook

分析のコードや結果をJupyter Notebookに逐次記録するツール

```
def add_cell_to_notebook(
    content: str,
    cell_type: Literal["code", "markdown"],
    artifact_paths: List[str]
):
    # セルを追加する
    nb = load_or_create_notebook()
    cell = create_cell(content, cell_type)
    nb.cells.append(cell)

    # 出力ファイルを表示する
    for path in artifact_paths:
        # テキストを表示
        if path.endswith(".csv"):
            display_csv(path)
        # 画像を表示
        elif path.endswith(".png"):
            display_image(path)
```

# ツール③ NW.add\_cell\_to\_notebook

分析のコードや結果をJupyter Notebookに逐次記録するツール

```
def add_cell_to_notebook(
    content: str,
    cell_type: Literal["code", "markdown"],
    artifact_paths: List[str]
):
    # セルを追加する
    nb = load_or_create_notebook()
    cell = create_cell(content, cell_type)
    nb.cells.append(cell)

    # 出力ファイルを表示する
    for path in artifact_paths:
        # テキストを表示
        if path.endswith(".csv"):
            display_csv(path)
        # 画像を表示
        elif path.endswith(".png"):
            display_image(path)
```

# ツール③ NW.add\_cell\_to\_notebook

分析のコードや結果をJupyter Notebookに逐次記録するツール

```
def add_cell_to_notebook(
    content: str,
    cell_type: Literal["code", "markdown"],
    artifact_paths: List[str]
):
    # セルを追加する
    nb = load_or_create_notebook()
    cell = create_cell(content, cell_type)
    nb.cells.append(cell)

    # 出力ファイルを表示する
    for path in artifact_paths:
        # テキストを表示
        if path.endswith(".csv"):
            display_csv(path)
        # 画像を表示
        elif path.endswith(".png"):
            display_image(path)
```

# ツール③ NW.add\_cell\_to\_notebook

分析のコードや結果をJupyter Notebookに逐次記録するツール

```
def add_cell_to_notebook(
    content: str,
    cell_type: Literal["code", "markdown"],
    artifact_paths: List[str]
):
    # セルを追加する
    nb = load_or_create_notebook()
    cell = create_cell(content, cell_type)
    nb.cells.append(cell)

    # 出力ファイルを表示する
    for path in artifact_paths:
        # テキストを表示
        if path.endswith(".csv"):
            display_csv(path)
        # 画像を表示
        elif path.endswith(".png"):
            display_image(path)
```

# ツール③ NW.add\_cell\_to\_notebook

分析のコードや結果をJupyter Notebookに逐次記録するツール

```
def add_cell_to_notebook(
    content: str,
    cell_type: Literal["code", "markdown"],
    artifact_paths: List[str]
):
    # セルを追加する
    nb = load_or_create_notebook()
    cell = create_cell(content, cell_type)
    nb.cells.append(cell)

    # 出力ファイルを表示する
    for path in artifact_paths:
        # テキストを表示
        if path.endswith(".csv"):
            display_csv(path)
        # 画像を表示
        elif path.endswith(".png"):
            display_image(path)
```

# ツール③ NW.add\_cell\_to\_notebook

分析のコードや結果をJupyter Notebookに逐次記録するツール

```
def add_cell_to_notebook(
    content: str,
    cell_type: Literal["code", "markdown"],
    artifact_paths: List[str]
):
    # セルを追加する
    nb = load_or_create_notebook()
    cell = create_cell(content, cell_type)
    nb.cells.append(cell)

    # 出力ファイルを表示する
    for path in artifact_paths:
        # テキストを表示
        if path.endswith(".csv"):
            display_csv(path)
        # 画像を表示
        elif path.endswith(".png"):
            display_image(path)
```

# 4. デモ：エージェントによる 自律的データ探索

# 動かしてみる

実際にclaude codeが頑張った様子を見てみましょう

出力レポート例

## 5. まとめと展望

# まとめと展望

LLMがデータを触りながら自律的に分析を行えるようになった（？）

- 「データを眺めながら考える」という思考プロセスを（少しほは）模倣できるようになった
- ツール（MCPサーバ）を整備してあげれば、それなりに自律的に動いてくれる
  - 問題設計・データ情報を確認するツール
  - 関数を実行して出力を保存するツール
  - 結果を記録するツール
- ツールを適宜修正すれば、画像・自然言語などにも応用できそう？
- 今回はEDAしかやってないけど、実験サイクルも回せるようにしたい
- 今のところやっぱりClaude Codeが優秀
  - Gemini CLIは関数実装で無限時間かかるコードを書きがちで、完遂できないことが多い
    - → サブエージェントに移譲するようにすれば解決できそう？

# Enjoy Kaggle & atmaCup!

X (Twitter) : [@tonic3561](#)

GitHub : <https://github.com/jintonic3561>

Thanks for listening!