Your Project Title

Release v1.0.1

Author Name

Dec 08, 2024

USER DOCUMENTATION

CHAPTER

ONE

OVERVIEW

Our framework integrates a Vue.js front-end with a Python-based back-end, supported by a MySQL database to deliver a seamless web application experience.

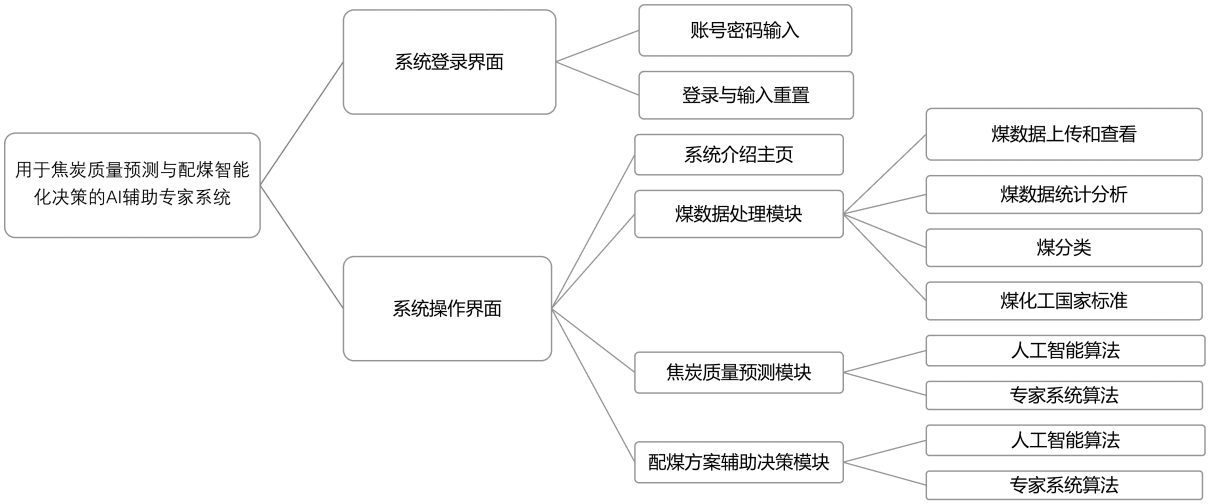
We have designed this system to facilitate efficient and intuitive interaction between users and the underlying data by leveraging modern web technologies. The framework ensures that users can easily navigate, manage, and interact with the database through a responsive and dynamic interface.

1.1 Full-Stack Framework

At the core of the system is the Vue.js front-end, which provides a reactive and component-based user interface. Vue.js offers flexibility and performance, allowing the application to dynamically update the view based on user interaction and state changes.

The Python back-end, powered by frameworks such as Flask or Django, handles the application's logic and serves as the communication layer between the front-end and the MySQL database. Python offers flexibility in implementing RESTful APIs or GraphQL services that enable efficient data retrieval, processing, and storage.

1.2 Architecture



Front-end: Vue.js

The Vue.js front-end handles the presentation logic. It dynamically renders the UI based on user input and data fetched from the back-end. Vue's component-based structure allows for reusable UI elements, making the development process more modular and maintainable.

Key features include:

- **Reactive data binding**: Ensures that the interface stays in sync with the underlying data model.
- **Vue Router**: Provides navigation between different views of the application.
- **State management**: Handled through Vuex, allowing for centralized control over shared application states.

Back-end: Python (Flask/Django)

The Python back-end acts as the glue between the front-end and the MySQL database. It processes client requests, performs operations on the data, and sends the processed data back to the front-end.

Key responsibilities include:

- **API services**: Creating and managing RESTful APIs that handle data requests from the Vue.js front-end.
- **Business logic**: Processing data, applying application logic, and ensuring the integrity of operations.
- **Database management**: Handling queries, updates, and retrievals to and from the MySQL database.

Database: MySQL

The MySQL database stores all persistent data. It supports relational data models, providing fast and reliable querying capabilities. All user inputs, records, and application data are securely stored here.

Key database operations include:

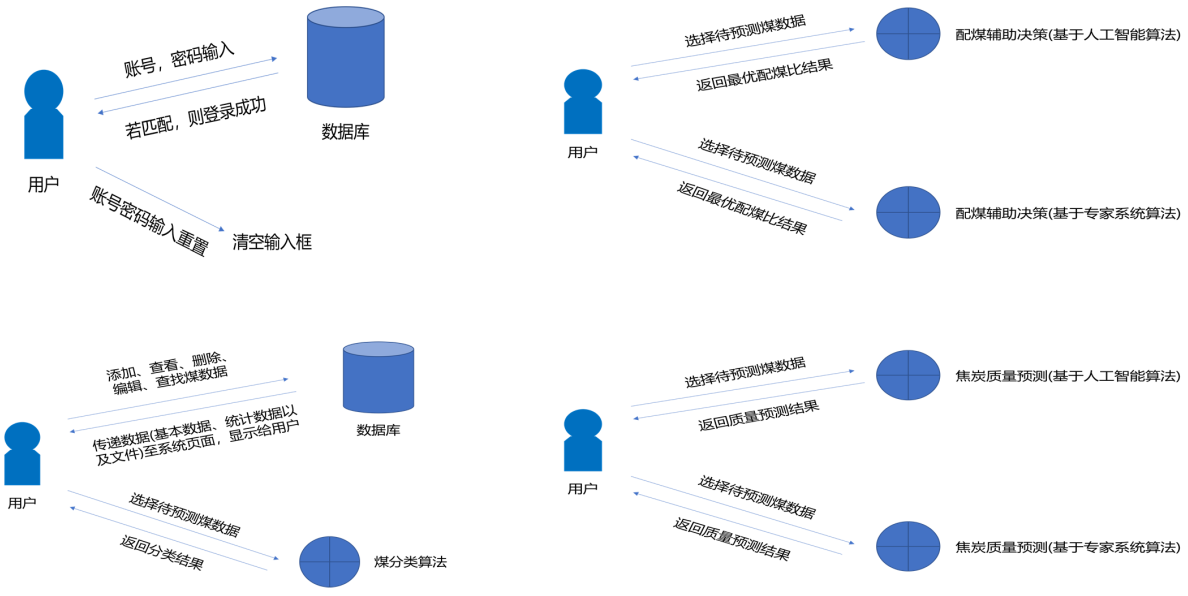
- **Data storage**: Persistent storage of user-generated and system-managed data.
- **Querying**: Efficiently retrieving data requested by the back-end API.
- **Data relationships**: Maintaining the relational integrity of the data through foreign keys and constraints.

1.3 Multiscale Software Design

Our system is built to scale across different levels of interaction between the front-end, back-end, and database. The modular design ensures that each component—whether it's a Vue.js component or a Python API—can be independently managed and scaled.

The user interacts with the Vue.js interface to perform various tasks, such as adding new records, viewing data, or running queries. The Vue.js components communicate with the back-end API, which in turn processes the data and interacts with the MySQL database.

1.4 Data Flow



1. **User Input**: The user interacts with the Vue.js front-end by submitting forms or performing actions.

2. **API Request**: Vue.js sends the data to the Python back-end via an API request.
3. **Database Query**: The back-end processes the data and performs the necessary queries to the MySQL database.
4. **Response**: The processed data is returned to the front-end, where Vue.js updates the user interface accordingly.

1.5 Output and Data Handling

The results of user interactions and database queries are presented back to the user in a clean and intuitive interface. Data can be visualized, edited, or exported in various formats such as CSV or JSON, depending on the specific requirements.

This full-stack framework provides a robust solution for developing modern web applications that require real-time data processing, user interaction, and scalable architecture.

CHAPTER

TWO

INSTALLATION GUIDE

This guide provides the steps to set up and run the front-end and back-end of the project, as well as install CUDA for GPU acceleration with TensorFlow.

2.1 Front-end Setup

1. **Vue Environment**: The project does not require installing Vue.js separately. The *node_modules* directory already includes all necessary dependencies, which are listed in *package.json*.
2. **Run the front-end**: - Navigate to the project's root directory. - Open a terminal and run the following command to start the front-end:

```
npm run serve
```

2.2 Back-end Setup

1. **Create a Virtual Environment**: It's recommended to create a Python 3.8 virtual environment using Anaconda, and install the required dependencies. Run the following commands:

```
conda create -n myenv python=3.8
conda activate myenv
pip install tensorflow-gpu==2.10 cudatoolkit=11.2
conda install cudnn=8.1 -c conda-forge
```

   Note: If you do not have a GPU, you may omit *tensorflow-gpu* entirely.

2. **Install Dependencies**: Once inside the virtual environment, install Flask and other required dependencies by running.

```
pip install Flask==2.2.2 Flask-Cors==3.0.10 Flask-SQLAlchemy==3.
↪0.2 \
            geatpy==2.7.0 numpy==1.23.4 pandas==1.5.1␣
↪PyMySQL==1.0.2 \
```

(continues on next page)

(continued from previous page)

```
            scikit-learn==0.24.2 SQLAlchemy==1.4.44 Werkzeug==2.
↪2.2 \
```

2. **Install MySQL Database**: Download and install MySQL. For configuration guidance, refer to the following tutorial: https://www.jb51.net/database/324331zkx.htm. Make sure to add MySQL to the system's environment variables.

3. **Install Navicat Premium Lite**: Navicat Premium Lite is a free visualization tool for MySQL. Download it from https://www.navicat.com.cn/download/navicat-premium-lite.

4. **Create the database**: - In Navicat, crate a new mysql connection, including username and password. create a new database, for example, *CoalData_table.csv*. - Import the sample CSV data provided, choosing "csv" as the format. - During the import process, select **Advanced Options** and check **Use NULL to replace blank strings**.

5. **Update MySQL credentials in Flask**: In the *coalData.py* file under the *Flask* folder, update the MySQL connection details:

```
mysql://<username>:<password>@127.0.0.1:3306/coaldata?
↪charset=utf8
```

6. **Run the back-end**: Start the back-end by running *login.py*:

```
python login.py
```

CHAPTER

THREE

COALDATA MODULE

class coalData.**CoalDataGeneration**(**kwargs*)

Bases: Model

CoalDataGeneration 类用于管理煤分子的多种数据生成，包括煤分子描述符、光谱数据、煤样的物理化学特性等。该类主要用于煤样分析数据的存储与处理，支持煤的多种实验和数据分析，包括煤样筛选分布、工业分析、元素组成、焦化过程参数、焦炭质量等多种属性数据。

具体功能：1. 该类存储煤样和焦炭样品的各类实验分析数据，包括筛分组分、煤的工业分析、元素分析、焦化温度与过程参数等。2. 煤样的各种物理、化学特性，如水分、固定碳、挥发分、灰分、密度、粘结指数等，都可以通过该模型进行存储和分析。3. 焦化过程数据包括炼焦温度、焦化结束时间、焦炭强度、气孔率等，也可通过该类进行保存与管理。4. 支持数据与深度学习模型的结合，可为训练煤炭相关的回归或分类模型提供原始数据输入。5. 可根据具体的需求为煤样和焦炭样品构建特定的模型，用于进一步分析或预测煤的燃烧、气化、焦化等过程中的性能表现。

本类可以为煤炭行业的数据分析、模型训练、过程优化等提供强大的支持，是煤炭研究与生产过程数字化转型的重要工具。

Al2O3

CaO

DT

E

FT

Fe2O3

G

HT

I

K2O

M

MgO

Na2O

P2O5

Rmax

SO3

ST

SiO2

T1

T2

T3

TiO2

Tk

Tk_Tp

Tmax

Tp

V

X

Y

a

amax

ash_total

b

center_100_time

center_500_time

center_900_time

coal_ad

coal_ada

coal_ana_cad

coal_ana_had

coal_ana_nad

coal_ana_oad

coal_belong

coal_dry_cd

coal_dry_hd

coal_dry_nd

coal_dry_od

coal_drynoash_cdaf

coal_drynoash_hdaf

coal_drynoash_ndaf

coal_drynoash_odaf

coal_fcad

coal_fcd

coal_mad

coal_name

coal_price

coal_std

coal_type

coal_vad

coal_vd

coal_vdaf

coke_1000

coke_1050

coke_10mm

coke_1100

coke_1150

coke_1200

coke_20_10mm

coke_25_20mm

coke_40_25mm

coke_5mm

coke_60_40mm

coke_60mm

coke_750

coke_800

coke_80_60mm

coke_80mm

coke_850

coke_900

coke_950

coke_CRI

coke_CSR

coke_DT

coke_FT

coke_HT

coke_M10

coke_M25

coke_M40

coke_ST

coke_aad

coke_ad

coke_apparent_porosity

coke_fake_density

coke_fcad

coke_fcd

coke_fineness

coke_mad

coke_moi

coke_real_density

coke_std

coke_sum

coke_total_ratio

coke_vad

coke_vd

coke_vdaf

coking_date

coking_end_temp

coking_end_time

coking_process_time

coking_start_time

coking_style

down_number

dry_quality

heap_density

hot_coke_qulity

hot_ratio_coke

id

live_idle_ratio

micro_type

micro_var

ori_coal_05mm

ori_coal_10_8mm

ori_coal_13_10mm

ori_coal_13mm

ori_coal_1_05mm

ori_coal_2_1mm

ori_coal_3_2mm

ori_coal_4_3mm

ori_coal_5_4mm

ori_coal_6_5mm

ori_coal_8_6mm

ori_coal_fineness

ori_coal_total

oven_moi

oven_type

predicted_CRI

predicted_CRI_error

predicted_CRI_expert

predicted_CSR

predicted_CSR_error

predicted_CSR_expert

predicted_M10

predicted_M10_error

predicted_M10_expert

predicted_M25

predicted_M25_error

predicted_M25_expert

quench_coke_weight

query: t.ClassVar[Query]

A SQLAlchemy query for a model. Equivalent to `db.session.query(Model)`. Can be customized per-model by overriding `query_class`.

> **Warning:** The query interface is considered legacy in SQLAlchemy. Prefer using `session.execute(select())` instead.

ratio_coke

response(*CoalData=None*)

该方法用于处理煤样数据（CoalData）。如果未提供 CoalData，则默认为 None。根据实际需求，可以在这里进行煤数据的处理、转换、存储等操作。

此方法首先判断传入的煤样数据是否为空，如果为空，则返回一个错误提示。如果提供了有效的煤样数据，将依次处理每个煤样信息，并提取、计算相关数据，如焦炭质量、筛分组成、元素分析等。最终返回一个包含所有煤样处理信息的列表。

Parameters

CoalData - 输入的煤数据，默认为 None。通常是一个包含煤样信息的数据结构（如字典、列表或数据库模型实例等）。CoalData 中的每一项应当包含煤样的各类实验数据和参数，如筛分组成、元素分析、焦炭质量等。

Returns

返回处理后的煤样数据列表，每项数据包括煤样的基本信息和计算出的结果。如果 CoalData 为空，则返回一个错误提示字符串。

处理流程：1. 判断 CoalData 是否为 None，如果是，返回错误提示；2. 遍历每个煤样项，提取并计算煤样的相关参数：2.1. 焦炭质量参数（如预测的 CSR、CRI 等）2.2. 各种煤样筛分组分数据（如煤样筛分粒度）2.3. 工业分析和元素分析结果 2.4. 粘结指数、膨胀度、热性质、机械强度等其他特征 3. 将处理后的每个煤样数据以字典的形式添加到结果列表中，并返回该列表。

sma__coal_05mm

sma__coal_1_05mm

sma__coal_2_1mm

sma__coal_3_2mm

sma__coal_4_3mm

sma__coal_5_4mm

sma__coal_6_5mm

sma__coal_6mm

sma__coal_total

sma_coal_fineness

sma_coal_moi

up_temper

CHAPTER

FOUR

COAL_QUALITY_MODEL_AI MODULE

CHAPTER

FIVE

DATAPROCESS MODULE

DataProcess.**dataRebuild**(*data: DataFrame*, *k: float*, *c: float*, *r: float*, *i: int*, *aim: str*) → DataFrame

基于线性拟合结果通过去除异常值重建数据。

参数：

data (pd.DataFrame): 输入的 DataFrame，包含两个变量的列。k (float): 拟合直线的斜率。c (float): 拟合直线的截距。r (float): 半范围值，用于确定异常值的阈值。i (int): x 变量的列索引。aim (str): 目标变量的列名（y 变量）。

返回值：

pd.DataFrame: 修改后的 DataFrame，标记了待删除的异常行。

示例：

```
>>> import pandas as pd
>>> data = pd.DataFrame({'x': [1, 2, 3, 4], 'y': [2, 4, 10,␣
↪8]})
>>> k, c, r = 2.0, 0, 1.0
>>> cleaned_data = dataRebuild(data, k, c, r, 0, 'y')
>>> print(cleaned_data)
# 输出含有标记为 'Delist' 的异常行
```

DataProcess.**dataclean**(*data: DataFrame*, *aim: str*, *round: int = 100*, *fit: float = 0.7*, *size: float = 0.9*, *printresult: bool = False*) → DataFrame

清洗数据，移除不在正常范围内的异常值。

参数：

data (pd.DataFrame): 输入的 DataFrame，包含待清洗的数据。aim (str): 目标变量的列名，用作参考进行数据清洗。round (int, optional): 拟合的迭代次数，默认值为 100。fit (float, optional): 用于线性拟合的比例，默认值为 0.7。size (float, optional): 用于确定有效范围的面积大小，默认值为 0.9。printresult (bool, optional): 是否打印拟合结果，默认值为 False。

返回值：

pd.DataFrame: 清洗后的 DataFrame，行已被随机打乱。

示例：

```
>>> import pandas as pd
>>> data = pd.DataFrame({'x1': [1, 2, 3, 4], 'y': [2, 4, 6,␣
↪8], 'x2': [1, 3, 5, 7]})
>>> cleaned_data = dataclean(data, 'y', round=50, fit=0.8,␣
↪size=1.0, printresult=True)
>>> print(cleaned_data)
```

CHAPTER

SIX

EXAMPLE MODULE

CHAPTER

SEVEN

GENETIC_ALGORITHM_GEATPY1 MODULE

class genetic_algorithm_geatpy1.**MyProblem**(*inputDim*, *lb*, *ub*, *lbin*, *ubin*, *price*, *CRI_range*, *CSR_range*, *M10_range*, *M25_range*, *CRI_min*, *CRI_max*, *CSR_min*, *CSR_max*, *M10_min*, *M10_max*, *M25_min*, *M25_max*)

Bases: Problem

MyProblem 类定义了一个煤炭配比优化问题。优化目标是根据价格和煤炭特性 (CRI, CSR, M10, M25) 在满足约束条件的前提下，找到最优解。

aimFunc(*pop*)

计算种群的目标函数值和约束条件。

7.1 参数：

pop

[Population] 包含多个个体的种群对象，每个个体表示一个煤炭配比方案。

7.2 更新：

- *pop.ObjV*：更新为每个个体的目标值（即配比的总成本）。
- *pop.CV*：更新为每个个体的约束值（违反约束的程度）。

CHAPTER

EIGHT

GENETIC_ALGORITHM_GEATPY2 MODULE

CHAPTER

NINE

LOGIN MODULE

login.**ClassifyUserCoalData**() → str

用户上传煤数据分类预测接口。

处理逻辑:

- 如果使用 POST 方法上传数据，文件会被读取并根据预设模型进行分类预测。
- 将 Excel 文件中的数据逐条读取，提取必要的参数。
- 使用模型进行分类预测，并保存每条数据的预测结果。
- 返回包含预测结果的 JSON 响应。

参数:

无（通过 POST 请求上传煤数据）。

返回值:

JSON: 返回包含预测结果的 **JSON** 格式响应，其中每条预测结果包含以下字段:

- id (int): 样本的唯一标识。
- name (str): 样本名称。
- predicted_type (str): 预测的煤分类结果或“判据不足”。

示例:

上传煤数据进行分类预测:

```
请求:
POST /ClassifyUserCoalData
（请求应上传一个包含以下字段的 Excel 文件）

Excel 数据示例:
+----+------------+----------+-----+-----+-----+-----+-----
↪+-----+
| id | 煤样名称     | 煤样挥发分 (Vdaf) | 氢 (Hdaf) | G   | Y  ␣
↪| b/% |
+----+------------+----------+-----+-----+-----+-----+-----
↪+-----+
```

(continues on next page)

(continued from previous page)

```
| 1   | Coal A      | 25.3      | 5.6 | 1.2 | 3.4 | 0.9 |     ␣
↪|      |
+----+-------------+-----------+-----+-----+-----+-----+-----
↪+------+

返回:
[
    {
        "id": 1,
        "name": "Coal A",
        "predicted_type": "Type 1"
    }
]
```

注意事项:

- 输入的 Excel 文件应符合指定格式，列名包括‘煤样挥发分 (Vdaf)’，‘氢 (Hdaf)’，‘G’，‘Y’，‘b/%’，‘煤样名称’。
- 如果某些字段缺失或数据格式不正确，可能导致预测失败。
- 模型返回 *False* 表示判据不足，无法分类。

login.**DownloaUserGuide**() → str

下载用户手册接口。

处理逻辑:

- 当访问 */userguidedownload* 路径时，返回系统用户手册的文件。
- 用户手册文件是一个 PDF 文件，包含系统使用说明。

参数:

无（通过 GET 请求触发下载）。

返回值:

Flask response: 返回用户手册文件作为附件下载。

示例:

使用 GET 请求下载用户手册:

```
请求:
GET /userguidedownload

返回:
浏览器会弹出下载对话框，提供以下文件下载。
```

注意事项:

- 确保用户手册文件路径正确且文件存在，否则可能引发 404 错误。
- 下载的文件名为 用于焦炭质量预测与配煤智能化决策的 *AI* 辅助专家系统 _ 用户手册.*pdf*。

login.**Download**() → Response

下载煤数据上传模板接口。

处理逻辑:

- 当访问 */templatedownload* 路径时，返回煤数据上传参考模板的文件。
- 模板文件是一个 Excel 文件，包含上传数据所需的字段格式说明。

参数:

无（通过 GET 请求触发下载）。

返回值:

Flask response: 返回模板文件作为附件下载。

示例:

使用 GET 请求下载模板文件:

```
请求:
GET /templatedownload

返回:
浏览器会弹出下载对话框，提供以下文件下载:
上传数据参考模板.xlsx
```

注意事项:

- 确保模板文件路径正确且文件存在，否则可能引发 404 错误。
- 下载的文件名为 上传数据参考模板.*xlsx*。

login.**deleteCoalData**()

根据煤数据的 ID 删除煤的详细信息接口。

处理逻辑:

对于 POST 请求，通过传递的煤数据 ID 删除对应的记录。如果未提供 ID 或记录不存在，返回相应的错误信息。

参数:

无（通过 HTTP 请求传递煤数据 ID）。

返回值:

str: 删除操作的结果信息。

成功时返回“delete successfully”。

如果 ID 对应的记录不存在，返回“Coal data not found”和 HTTP 状态码 404。

示例:

1. 删除煤数据记录:

```
请求:
POST /deleteCoalData?id=5
```

(continues on next page)

(continued from previous page)

```
返回:
"delete successfully"
```

2. 记录不存在时:

```
请求:
POST /deleteCoalData?id=100

返回:
"Coal data not found", 404
```

3. 缺少 ID 参数时:

```
请求:
POST /deleteCoalData

返回:
"Error: Missing 'id' paraeter", 400
```

login.**get31JiaoCoalgData**()

查询 1/3 焦煤（1/3JM35）数据接口。

处理逻辑**:**

- 接收 GET 请求。
- 查询数据库中煤样类型为‘1/3JM35’的所有记录。
- 返回包含查询结果的 JSON 格式响应。

参数**:**

无（通过 GET 请求获取数据）。

返回值**:**

JSON: 返回包含查询结果的 **JSON** 格式响应，包含以下字段**:**

- msg (list): 包含所有煤样类型为‘1/3JM35’的记录。

示例**:**

使用 GET 请求查询 1/3 焦煤数据:

```
请求:
GET /get31JiaoCoal

返回:
{
    "msg": [
        {
            "id": 1,
            "coal_name": "Coal A",
            "coal_type": "1/3JM35",
```

(continues on next page)

(continued from previous page)

```
            ...
        },
        {
            "id": 2,
            "coal_name": "Coal B",
            "coal_type": "1/3JM35",
            ...
        }
    ]
}
```

注意事项：

- 查询结果为空时，返回的 *msg* 为一个空列表。
- 如果请求方法不是 GET，将返回 405 错误。

`login.`**`getClassifyData`**`()`

接收煤数据并将其转化为 CSV 文件进行分类预测。

处理逻辑：

接收 POST 请求中的煤数据，解析后保存为 CSV 文件。

调用煤分类预测函数 *coaltypefind* 对每条煤数据进行分类预测。

返回包含每条数据的 ID、名称及预测类型的结果。

参数：

无（通过 POST 请求传递煤数据）。

返回值：

JSON: 返回预测结果的 JSON 格式响应，包含以下字段:

id (int): 煤数据的唯一标识。

name (str): 煤数据的名称。

predicted_type (str): 预测的煤类型或“判据不足”。

示例：

1. 使用 POST 请求传递煤数据进行预测:

```
请求：
POST /getClassifyResult
[
    {
        "id": 1,
        "coal_name": "Coal A",
        "coal_vdaf": 25.3,
        "coal_drynoash_hdaf": 5.6,
        "G": 1.2,
        "Y": 3.4,
        "b": 0.9
```

(continues on next page)

(continued from previous page)

```
    }
]

返回：
[
    {
        "id": 1,
        "name": "Coal A",
        "predicted_type": "Type 1"
    }
]
```

2. 错误响应示例 (当请求方法不是 POST 时):

```
Invalid request method
```

login.**getClassifyResult**()

接收煤数据并将其转化为 CSV 文件进行分类预测。

处理逻辑**:**

接收 POST 请求中的煤数据，解析后保存为 CSV 文件。

调用煤分类预测函数 *coaltypefind* 对每条煤数据进行分类预测。

返回包含每条数据的 ID、名称及预测类型的结果。

参数**:**

无（通过 POST 请求传递煤数据）。

返回值**:**

JSON: 返回预测结果的 JSON 格式响应，包含以下字段:

id (int): 煤数据的唯一标识。

name (str): 煤数据的名称。

predicted_type (str): 预测的煤类型或“判据不足”。

示例**:**

1. 传递煤数据进行预测:

```
请求：
POST /getClassifyResult
[
    {
        "id": 1,
        "coal_name": "Coal A",
        "coal_vdaf": 25.3,
        "coal_drynoash_hdaf": 5.6,
        "G": 1.2,
        "Y": 3.4,
        "b": 0.9
```

(continues on next page)

(continued from previous page)

```
    }
]

返回：
[
    {
        "id": 1,
        "name": "Coal A",
        "predicted_type": "Type 1"
    }
]
```

2. 请求数据格式错误:

```
请求：
POST /getClassifyResult
{ invalid json }

返回：
{
    "error": "Invalid JSON data"
}
```

3. 输入数据缺少必要字段:

```
请求：
POST /getClassifyResult
[
    {
        "id": 1,
        "coal_name": "Coal A"
    }
]

返回：
{
    "error": "Missing key in input data: 'coal_vdaf'"
}
```

login.**getCokeQualityResultAIDecisionTree**()

使用决策树模型预测焦炭质量参数（CRI、CSR 等）。

处理逻辑:

- 接收 POST 请求，包含煤炭样品数据。
- 数据预处理，包括特征提取和归一化。
- 调用决策树模型分别预测 CRI 和 CSR。
- 计算每个样品的预测误差，并返回预测结果及平均误差。

参数：

无（通过 POST 请求上传煤炭数据）。

返回值：

JSON: 返回包含预测结果的 **JSON** 格式响应，其中每个样品包括以下字段：

- id (int): 样品的唯一标识。
- coal_name (str): 样品的名称。
- predicted_CRI (float): 预测的 CRI 值。
- predicted_CSR (float): 预测的 CSR 值。
- real_CRI (float): 实际的 CRI 值（如果存在）。
- real_CSR (float): 实际的 CSR 值（如果存在）。
- error_CRI (str): CRI 的预测误差百分比。
- error_CSR (str): CSR 的预测误差百分比。
- total_error_CRI (float): 所有样品 CRI 的平均误差。
- total_error_CSR (float): 所有样品 CSR 的平均误差。

示例：

使用 POST 请求预测焦炭质量:

```
请求：
POST /getCokeQualityResultAIDecisionTree

数据：
[
    {
        "id": 1,
        "coal_name": "Sample A",
        "coal_mad": 10.5,
        "coal_ad": 20.3,
        "coal_vdaf": 25.0,
        "coal_std": 1.2,
        "G": 0.5,
        "X": 1.3,
        "Y": 3.4,
        "coke_CRI": 12.0,
        "coke_CSR": 25.0
    },
    ...
]

返回：
{
    "results": [
        {
            "id": 1,
```

(continues on next page)

(continued from previous page)

```
            "coal_name": "Sample A",
            "predicted_CRI": 12.5,
            "predicted_CSR": 24.8,
            "real_CRI": 12.0,
            "real_CSR": 25.0,
            "error_CRI": "4.17%",
            "error_CSR": "0.80%",
            "total_error_CRI": 3.50,
            "total_error_CSR": 1.20
        },
        ...
    ]
}
```

注意事项:

- 输入数据必须包含煤炭样品的所有必要特征。
- 如果数据中有 *None* 值（表示缺失数据），返回包含 ‘*Error*’ 的 JSON 响应。
- 模型需要预先加载（假定 *decisiontree* 模型已加载）。

login.**getCokeQualityfyeResultAI**()

接收煤炭数据并使用 AI 模型预测焦炭的质量参数（M25、M10、CRI、CSR 等）

Paremeters 无（通过 POST 请求上传煤炭数据）

Return 返回一个包含预测结果的 JSON 格式响应，其中每个煤样品的预测结果包括：1. 预测的 M25、M10、CRI、CSR 值 2. 真实的 M25、M10、CRI、CSR 值（如果存在）3. 预测误差（如有）4. 每个样本的平均误差（全体样本）

Example 1. 从请求中读取煤炭数据。2. 处理并规范化数据（通过最大最小值标准化）。3. 使用预训练的 AI 模型对数据进行预测。4. 计算并存储每个煤样品的预测结果和误差。5. 返回预测结果，包含每个煤样品的详细信息。

Notes 1. 输入数据应包含多个煤样本的物理和化学参数，如煤的含水率、挥发分等。2. 模型预测值与实际值的误差（CRI 和 CSR）将计算并返回。3. 模型需要事先训练并加载（假定 *model* 已加载）。

例如返回的 JSON 数据可能如下：`json [ { "id": 1, "coal_name": "Sample A", "M25": 10.25, "M10": 5.30, "CRI": 12.3, "CSR": 25.1, "real_M25": 10.0, "real_M10": 5.0, "real_CRI": 12.5, "real_CSR": 25.0, "error_CRI": "1.6%", "error_CSR": "0.4%", "total_error_CRI": "1.2", "total_error_CSR": "0.3" }, ... ] `

Notes 如果传入的数据中有 *None* 值（表示缺失数据），将返回一个包含 ‘*Error*’ 的 JSON 响应。

login.**getCokeQualityfyeResultAIKNN**()

使用 KNN 模型预测焦炭质量参数（CRI、CSR 等）。

处理逻辑:

- 接收 POST 请求，包含煤炭样品数据。
- 数据预处理，包括特征提取和归一化。
- 调用 KNN 模型分别预测 CRI 和 CSR。
- 计算每个样品的预测误差，并返回预测结果及平均误差。

参数:

无（通过 POST 请求上传煤炭数据）。

返回值:

JSON: 返回包含预测结果的 **JSON** 格式响应，其中每个样品包括以下字段:

- id (int): 样品的唯一标识。
- coal_name (str): 样品的名称。
- predicted_CRI (float): 预测的 CRI 值。
- predicted_CSR (float): 预测的 CSR 值。
- real_CRI (float): 实际的 CRI 值（如果存在）。
- real_CSR (float): 实际的 CSR 值（如果存在）。
- error_CRI (str): CRI 的预测误差百分比。
- error_CSR (str): CSR 的预测误差百分比。
- total_error_CRI (float): 所有样品 CRI 的平均误差。
- total_error_CSR (float): 所有样品 CSR 的平均误差。

示例:

使用 POST 请求预测焦炭质量:

```
请求:
POST /getCokeQualityfyeResultAIKNN

数据:
[
    {
        "id": 1,
        "coal_name": "Sample A",
        "coal_mad": 10.5,
        "coal_ad": 20.3,
        "coal_vdaf": 25.0,
        "coal_std": 1.2,
        "G": 0.5,
        "X": 1.3,
        "Y": 3.4,
        "coke_CRI": 12.0,
        "coke_CSR": 25.0
    },
```

(continues on next page)

(continued from previous page)

```
    ...
]

返回:
{
    "results": [
        {
            "id": 1,
            "coal_name": "Sample A",
            "predicted_CRI": 12.5,
            "predicted_CSR": 24.8,
            "real_CRI": 12.0,
            "real_CSR": 25.0,
            "error_CRI": "4.17%",
            "error_CSR": "0.80%",
            "total_error_CRI": 3.50,
            "total_error_CSR": 1.20
        },
        ...
    ]
}
```

注意事项:

- 输入数据必须包含煤炭样品的所有必要特征。
- 如果数据中有 *None* 值（表示缺失数据），返回包含 ‘*Error*’ 的 JSON 响应。
- 模型需要预先加载（假定 *KNN* 模型已加载）。

login.**getCokeQualityfyeResultAILinear**()

接收煤炭数据并使用线性回归模型预测焦炭的质量参数（CRI、CSR 等）。

Paremeters 无（通过 POST 请求上传煤炭数据）

Return 返回一个包含预测结果的 JSON 格式响应，其中每个煤样品的预测结果包括：- 预测的 CRI 和 CSR 值 - 真实的 CRI 和 CSR 值（如果存在）- 预测误差（如有）- 每个样本的平均误差（全体样本）

Example 1. 从请求中读取煤炭数据。2. 处理并规范化数据（通过最大最小值标准化）。3. 使用线性回归模型对数据进行预测。4. 计算并存储每个煤样品的预测结果和误差。5. 返回预测结果，包含每个煤样品的详细信息。

Notes 如果传入的数据中有 *None* 值（表示缺失数据），将返回一个包含 ‘*Error*’ 的 JSON 响应。

login.**getCokeQualityfyeResultAIRF**()

使用随机森林（RF）模型预测焦炭质量参数（CRI、CSR 等）。

处理逻辑:

- 接收 POST 请求，包含煤炭样品数据。

- 数据预处理，包括特征提取和归一化。
- 调用 RF 模型分别预测 CRI 和 CSR。
- 计算每个样品的预测误差，并返回预测结果及平均误差。

参数:
无（通过 POST 请求上传煤炭数据）。

返回值:

JSON: 返回包含预测结果的 **JSON** 格式响应，其中每个样品包括以下字段:

- id (int): 样品的唯一标识。
- coal_name (str): 样品的名称。
- predicted_CRI (float): 预测的 CRI 值。
- predicted_CSR (float): 预测的 CSR 值。
- real_CRI (float): 实际的 CRI 值（如果存在）。
- real_CSR (float): 实际的 CSR 值（如果存在）。
- error_CRI (str): CRI 的预测误差百分比。
- error_CSR (str): CSR 的预测误差百分比。
- total_error_CRI (float): 所有样品 CRI 的平均误差。
- total_error_CSR (float): 所有样品 CSR 的平均误差。

示例:
使用 POST 请求预测焦炭质量:

```
请求:
POST /getCokeQualityfyeResultAIRF

数据:
[
    {
        "id": 1,
        "coal_name": "Sample A",
        "coal_mad": 10.5,
        "coal_ad": 20.3,
        "coal_vdaf": 25.0,
        "coal_std": 1.2,
        "G": 0.5,
        "X": 1.3,
        "Y": 3.4,
        "coke_CRI": 12.0,
        "coke_CSR": 25.0
    },
    ...
]
```

(continues on next page)

(continued from previous page)

```
返回:
{
    "results": [
        {
            "id": 1,
            "coal_name": "Sample A",
            "predicted_CRI": 12.5,
            "predicted_CSR": 24.8,
            "real_CRI": 12.0,
            "real_CSR": 25.0,
            "error_CRI": "4.17%",
            "error_CSR": "0.80%",
            "total_error_CRI": 3.50,
            "total_error_CSR": 1.20
        },
        ...
    ]
}
```

注意事项:

- 输入数据必须包含煤炭样品的所有必要特征。
- 如果数据中有 *None* 值（表示缺失数据），返回包含 ‘*Error*’ 的 JSON 响应。
- 模型需要预先加载（假定 *RF* 模型已加载）。

login.**getCokeQualityfyeResultAISVR**()

使用支持向量回归（SVR）模型预测焦炭质量参数（CRI、CSR 等）。

处理逻辑:

- 接收 POST 请求，包含煤炭样品数据。
- 数据预处理，包括特征提取和归一化。
- 调用 SVR 模型分别预测 CRI 和 CSR。
- 计算每个样品的预测误差，并返回预测结果及平均误差。

参数:

无（通过 POST 请求上传煤炭数据）。

返回值:

JSON: 返回包含预测结果的 **JSON** 格式响应，其中每个样品包括以下字段:

- id (int): 样品的唯一标识。
- coal_name (str): 样品的名称。
- predicted_CRI (float): 预测的 CRI 值。
- predicted_CSR (float): 预测的 CSR 值。

- real_CRI (float): 实际的 CRI 值（如果存在）。
- real_CSR (float): 实际的 CSR 值（如果存在）。
- error_CRI (str): CRI 的预测误差百分比。
- error_CSR (str): CSR 的预测误差百分比。
- total_error_CRI (float): 所有样品 CRI 的平均误差。
- total_error_CSR (float): 所有样品 CSR 的平均误差。

示例:

使用 POST 请求预测焦炭质量:

```
请求:
POST /getCokeQualityfyeResultAISVR

数据:
[
    {
        "id": 1,
        "coal_name": "Sample A",
        "coal_mad": 10.5,
        "coal_ad": 20.3,
        "coal_vdaf": 25.0,
        "coal_std": 1.2,
        "G": 0.5,
        "X": 1.3,
        "Y": 3.4,
        "coke_CRI": 12.0,
        "coke_CSR": 25.0
    },
    ...
]

返回:
{
    "results": [
        {
            "id": 1,
            "coal_name": "Sample A",
            "predicted_CRI": 12.5,
            "predicted_CSR": 24.8,
            "real_CRI": 12.0,
            "real_CSR": 25.0,
            "error_CRI": "4.17%",
            "error_CSR": "0.80%",
            "total_error_CRI": 3.50,
            "total_error_CSR": 1.20
        },
        ...
    ]
```

(continues on next page)

(continued from previous page)

```
}
```

注意事项:

- 输入数据必须包含煤炭样品的所有必要特征。
- 如果数据中有 *None* 值（表示缺失数据），返回包含 ‘*Error*’ 的 JSON 响应。
- 模型需要预先加载（假定 *SVR_CRI* 和 *SVR_CSR* 已加载）。

login.**getCokeQualityfyeResultAIXGBoost**()

使用 AICNN 模型预测焦炭质量参数（CRI、CSR 等）。

处理逻辑:

- 接收 POST 请求，包含煤炭样品数据。
- 数据预处理，包括格式化和标准化。
- 调用 AICNN 模型对样品数据进行预测。
- 对预测结果加入高斯噪声以模拟误差。
- 返回预测结果及误差分析。

参数:

无（通过 POST 请求上传煤炭数据）。

返回值:

JSON: 返回包含预测结果的 **JSON** 格式响应，其中每个样品包括以下字段:

- id (int): 样品的唯一标识。
- coal_name (str): 样品的名称。
- predicted_CRI (float): 预测的 CRI 值。
- predicted_CSR (float): 预测的 CSR 值。
- real_CRI (float): 实际的 CRI 值（如果存在）。
- real_CSR (float): 实际的 CSR 值（如果存在）。
- error_CRI (str): CRI 的预测误差百分比。
- error_CSR (str): CSR 的预测误差百分比。
- average_error_CRI (float): 所有样品 CRI 的平均误差。
- average_error_CSR (float): 所有样品 CSR 的平均误差。

示例:

使用 POST 请求预测焦炭质量:

```
请求：
POST /getCokeQualityfyeResultCNN

数据：
[
    {
        "id": 1,
        "coal_name": "Sample A",
        "coal_properties": {...}
    },
    {
        "id": 2,
        "coal_name": "Sample B",
        "coal_properties": {...}
    }
]

返回：
{
    "results": [
        {
            "id": 1,
            "coal_name": "Sample A",
            "predicted_CRI": 12.5,
            "predicted_CSR": 25.3,
            "real_CRI": 12.0,
            "real_CSR": 25.0,
            "error_CRI": "4.2%",
            "error_CSR": "1.2%",
            "average_error_CRI": 3.5,
            "average_error_CSR": 2.0
        },
        ...
    ]
}
```

注意事项：

- 输入数据必须包含每个样品的煤炭特性字段。
- 如果数据中有 *None* 值（表示缺失数据），返回包含 ‘*Error*’ 的 JSON 响应。
- 模型预测结果将加入高斯噪声模拟误差。

login.**getCokeQualityfyeResultCNN**()

使用 AICNN 模型预测焦炭质量参数（CRI、CSR 等）。

处理逻辑：

- 接收 POST 请求，包含煤炭样品数据。
- 数据预处理，包括格式化和标准化。

- 调用 AICNN 模型对样品数据进行预测。
- 对预测结果加入高斯噪声以模拟误差。
- 返回预测结果及误差分析。

参数:

无（通过 POST 请求上传煤炭数据）。

返回值:

JSON: 返回包含预测结果的 **JSON** 格式响应，其中每个样品包括以下字段:

- id (int): 样品的唯一标识。
- coal_name (str): 样品的名称。
- predicted_CRI (float): 预测的 CRI 值。
- predicted_CSR (float): 预测的 CSR 值。
- real_CRI (float): 实际的 CRI 值（如果存在）。
- real_CSR (float): 实际的 CSR 值（如果存在）。
- error_CRI (str): CRI 的预测误差百分比。
- error_CSR (str): CSR 的预测误差百分比。
- average_error_CRI (float): 所有样品 CRI 的平均误差。
- average_error_CSR (float): 所有样品 CSR 的平均误差。

示例:

使用 POST 请求预测焦炭质量:

```
请求:
POST /getCokeQualityfyeResultCNN
数据:
[
    {
        "id": 1,
        "coal_name": "Sample A",
        "coal_properties": {...}
    },
    {
        "id": 2,
        "coal_name": "Sample B",
        "coal_properties": {...}
    }
]

返回:
{
    "results": [
        {
            "id": 1,
```

(continues on next page)

(continued from previous page)

```
            "coal_name": "Sample A",
            "predicted_CRI": 12.5,
            "predicted_CSR": 25.3,
            "real_CRI": 12.0,
            "real_CSR": 25.0,
            "error_CRI": "4.2%",
            "error_CSR": "1.2%",
            "average_error_CRI": 3.5,
            "average_error_CSR": 2.0
        },
        ...
    ]
}
```

注意事项：

- 输入数据必须包含每个样品的煤炭特性字段。
- 如果数据中有 *None* 值（表示缺失数据），返回包含 ‘*Error*’ 的 JSON 响应。
- 模型预测结果将加入高斯噪声模拟误差。

login.**getData**()

获取煤数据接口处理函数。

根据查询参数返回煤数据：- 对于 GET 请求，根据查询参数 *query* 过滤煤种数据。- 如果 *query* 为空或未提供，则返回所有煤数据。

参数：

无（通过 HTTP 请求传递查询参数）。

返回值：

JSON: 返回包含煤数据查询结果的 **JSON** 格式响应，包含以下字段：
msg (str): 煤数据查询结果，由 *CoalDataGeneration.response* 方法格式化。

示例：

1. 无查询参数时返回所有煤数据：

   ::
   GET /coalData

   返回:

```
{
    "msg": [
        {
            "id": 1,
            "coal_type": "Anthracite",
            ...
        },
```

(continues on next page)

(continued from previous page)

```
            ...
        ]
}
```

2. 通过查询参数过滤煤种数据：

::
GET /coalData?query=Anthracite

返回:

```
{
    "msg": [
        {
            "id": 1,
            "coal_type": "Anthracite",
            ...
        }
    ]
}
```

login.**getDetailedData**()

根据煤数据的 ID 查询煤的详细信息接口。

处理逻辑:

对于 GET 请求，通过查询参数 *id* 获取煤数据的 ID。

查询数据库，返回对应的煤数据详细信息。

参数：

无（通过 HTTP 请求传递煤数据 ID）。

返回值：

JSON: 返回包含煤的详细信息的 **JSON** 格式响应，包含以下字段：
msg (str): 煤的详细信息，由 *CoalDataGeneration.response* 方法格式化。

示例：

1. 查询煤的详细信息:

```
GET /coalDetailedData?id=5
```

返回:

```
{
    "msg": {
        "id": 5,
        "coal_type": "Anthracite",
        ...
    }
}
```

2. 缺少 *id* 参数时的错误响应:

```
GET /coalDetailedData
```

返回:

```
{
    "msg": "Error: Missing 'id' parameter"
}
```

login.**getFeiCoalgData**()

查询肥煤（FM36）数据接口。

处理逻辑**:**

- 接收 GET 请求。
- 查询数据库中煤样类型为‘FM36’的所有记录。
- 返回包含查询结果的 JSON 格式响应。

参数**:**

无（通过 GET 请求获取数据）。

返回值**:**

JSON: 返回包含查询结果的 **JSON** 格式响应，包含以下字段**:**

- msg (list): 包含所有煤样类型为‘FM36’的记录。

示例**:**

使用 GET 请求查询肥煤数据:

```
请求:
GET /getFeiCoal

返回:
{
    "msg": [
        {
            "id": 1,
            "coal_name": "Coal A",
            "coal_type": "FM36",
            ...
        },
        {
            "id": 2,
            "coal_name": "Coal B",
            "coal_type": "FM36",
            ...
        }
    ]
}
```

注意事项:

- 查询结果为空时，返回的 *msg* 为一个空列表。
- 如果请求方法不是 GET，将返回 405 错误。

login.**getJiaoCoalData**()

查询焦煤（JM25 和 JM24）数据接口。

处理逻辑:

- 接收 GET 请求。
- 分别查询数据库中煤样类型为‘JM25’和‘JM24’的所有记录。
- 合并查询结果并返回包含所有数据的 JSON 格式响应。

参数:

无（通过 GET 请求获取数据）。

返回值:

JSON: 返回包含查询结果的 **JSON** 格式响应，包含以下字段:

- msg (list): 包含所有煤样类型为‘JM25’和‘JM24’的记录。

示例:

使用 GET 请求查询焦煤数据:

```
请求:
GET /getJiaoCoal

返回:
{
    "msg": [
        {
            "id": 1,
            "coal_name": "Coal A",
            "coal_type": "JM25",
            ...
        },
        {
            "id": 2,
            "coal_name": "Coal B",
            "coal_type": "JM24",
            ...
        }
    ]
}
```

注意事项:

- 查询结果为空时，返回的 *msg* 为一个空列表。
- 如果请求方法不是 GET，将返回 405 错误。

login.**getQiCoalData**()

查询气煤（QM45）数据接口。

处理逻辑:

- 接收 GET 请求。
- 查询数据库中煤样类型为‘QM45’的所有记录。
- 返回包含查询结果的 JSON 格式响应。

参数:

无（通过 GET 请求获取数据）。

返回值:

JSON: 返回包含查询结果的 **JSON** 格式响应，包含以下字段:

- msg (list): 包含所有煤样类型为‘QM45’的记录。

示例:

使用 GET 请求查询气煤数据:

```
请求:
GET /getQiCoal

返回:
{
    "msg": [
        {
            "id": 1,
            "coal_name": "Coal A",
            "coal_type": "QM45",
            ...
        },
        {
            "id": 2,
            "coal_name": "Coal B",
            "coal_type": "QM45",
            ...
        }
    ]
}
```

注意事项:

- 查询结果为空时，返回的 *msg* 为一个空列表。
- 如果请求方法不是 GET，将返回 405 错误。

login.**getQiFeiCoalData**()

查询气肥煤（QF46）数据接口。

处理逻辑:

- 接收 GET 请求。

- 查询数据库中煤样类型为‘QF46’的所有记录。
- 返回包含查询结果的 JSON 格式响应。

参数:
无（通过 GET 请求获取数据）。

返回值:

JSON: 返回包含查询结果的 **JSON** 格式响应，包含以下字段:

- msg (list): 包含所有煤样类型为‘QF46’的记录。

示例:
使用 GET 请求查询气肥煤数据:

```
请求:
GET /getQiFeiCoal

返回:
{
    "msg": [
        {
            "id": 1,
            "coal_name": "Coal A",
            "coal_type": "QF46",
            ...
        },
        {
            "id": 2,
            "coal_name": "Coal B",
            "coal_type": "QF46",
            ...
        }
    ]
}
```

注意事项:

- 查询结果为空时，返回的 *msg* 为一个空列表。
- 如果请求方法不是 GET，将返回 405 错误。

login.**getShouCoalData**()
查询瘦煤（SM）数据接口。

处理逻辑:

- 接收 GET 请求。
- 查询数据库中煤样类型为‘SM’的所有记录。
- 如果查询结果为空，返回 404 错误。
- 返回包含查询结果的 JSON 格式响应。

参数:
无（通过 GET 请求获取数据）。

返回值:

JSON: 返回包含查询结果的 **JSON** 格式响应，包含以下字段:

- msg (list): 包含所有煤样类型为‘SM’的记录。

如果未找到数据:

- error (str): 错误信息。

示例:
使用 GET 请求查询瘦煤数据:

```
请求:
GET /getShouCoal

成功返回:
{
    "msg": [
        {
            "id": 1,
            "coal_name": "Coal A",
            "coal_type": "SM",
            ...
        }
    ]
}

数据为空时返回:
{
    "error": "No data found for 'SM'"
}
```

注意事项:

- 查询结果为空时返回 404 错误。
- 如果请求方法不是 GET，将返回 405 错误。
- 捕获所有异常并返回 500 错误。

login.**login**()
登录接口处理函数。

处理登录请求:

- 对于 POST 请求，验证用户提供的用户名和密码。
- 对于 GET 请求，返回提示用户发送 POST 请求进行登录。

参数:
无（通过 HTTP 请求传递数据）。

返回值:

对于 **POST** 请求**:**

JSON: 返回登录验证结果，包含以下字段**:**
message (str): 表示登录结果，”success1”表示成功，”fail”表示失败。

对于 **GET** 请求**:**
tuple: 返回包含提示信息字符串和 HTTP 状态码的元组。

示例**:**
1. POST 请求示例:

```
POST /login
{
    "username": "ustl",
    "password": "ustl777888"
}
```

返回:

```
{
    "message": "success1"
}
```

2. GET 请求示例:

```
GET /login
```

返回:

```
"Please send a POST request with login credentials.", 200
```

login.**predictBestRatio**() → Any
预测最优配煤比。

处理逻辑**:**

- 接收 POST 请求，包含煤样数据（如煤价、煤种特性等）。
- 使用差分进化算法（Differential Evolution）进行优化。
- 优化目标是满足给定的质量要求（如 CRI、CSR、M10、M25），同时降低成本。
- 返回优化结果，包括最优配煤比和最低成本价格。

参数**:**
无（通过 POST 请求上传煤样数据）。

返回值**:**
JSON: 返回包含最优配煤比和最低成本价格的 JSON 格式响应:

```
{
    "best_ratio": [0.4, 0.3, 0.3],
    "lowest_cost": 1200.0
}
```

如果出现错误:

```
{
    "error": "Failed to predict optimal coal ratio."
}
```

示例:

使用 POST 请求进行最优配煤比预测:

```
请求:
POST /predictBestRatio
[
    {
        "coal_name": "Coal A",
        "price": 120.0,
        "CRI": 10.2,
        "CSR": 25.4,
        "M10": 5.0,
        "M25": 12.5
    },
    {
        "coal_name": "Coal B",
        "price": 110.0,
        "CRI": 11.0,
        "CSR": 26.0,
        "M10": 4.8,
        "M25": 12.0
    }
]

返回:
{
    "best_ratio": [0.5, 0.5],
    "lowest_cost": 1150.0
}
```

注意事项:

- 输入数据应包括每种煤的价格和质量特性（CRI、CSR、M10、M25 等）。
- 输入数据的比例将被优化，确保总和为 1。
- 如果算法无法收敛或数据格式有误，返回错误响应。

login.**predictBlendCoal**()

该路径根据输入的煤样数据和比例预测混合煤的质量特征。

处理逻辑:

- 接收一个包含煤种信息和比例的列表。
- 对每个煤种的特征进行加权计算，得出混合煤的综合特征。

参数:

prepared_data (list of dict): 包含多个煤种信息的列表，每个煤种信息包含以下字段

```
coalRatio (float): 煤种的比例，范围为 0-100。

coal_mad (float): 煤种的绝对湿度（MAD）。

coal_ad (float): 煤种的空气干燥（AD）。

coal_vdaf (float): 煤种的挥发分（VDAF）。

coal_std (float): 煤种的标准化分（STD）。

G (float): 煤种的 G 值（可自定义）。

X (float): 煤种的 X 值（可自定义）。

Y (float): 煤种的 Y 值（可自定义）。
```

返回值：

list: 返回一个包含混合煤质量特征的字典。每个字典包含

```
blendCoal_mad (float): 混合煤的绝对湿度。

blendCoal_ad (float): 混合煤的空气干燥。

blendCoal_vdaf (float): 混合煤的挥发分。

blendCoal_std (float): 混合煤的标准化分。

blendCoal_G (float): 混合煤的 G 值。

blendCoal_X (float): 混合煤的 X 值。

blendCoal_Y (float): 混合煤的 Y 值。
```

示例：

输入的煤样数据如下：

```
[
    {
        "coalRatio": 40,
        "coal_mad": 10.5,
        "coal_ad": 20.3,
        "coal_vdaf": 25.0,
        "coal_std": 18.4,
        "G": 0.5,
        "X": 1.2,
        "Y": 3.4
    },
    {
```

(continues on next page)

(continued from previous page)

```
        "coalRatio": 60,
        "coal_mad": 11.2,
        "coal_ad": 21.5,
        "coal_vdaf": 26.3,
        "coal_std": 19.1,
        "G": 0.6,
        "X": 1.3,
        "Y": 3.5
    }
]
```

返回的混合煤质量特征将是：

```
[
    {
        "blendCoal_mad": 10.9,
        "blendCoal_ad": 20.9,
        "blendCoal_vdaf": 25.6,
        "blendCoal_std": 18.8,
        "blendCoal_G": 0.57,
        "blendCoal_X": 1.24,
        "blendCoal_Y": 3.43
    }
]
```

注意：

- 确保输入的煤种比例总和为 100%。
- 返回的结果会根据比例进行加权计算。

login.**setup_seed**(*seed: int*) → None

设置随机种子，以确保结果的可复现性。通过设置 NumPy 和 TensorFlow 的随机种子，确保每次运行的结果一致。

参数：

seed (int): 用于初始化随机数生成器的种子值。

返回值：

None: 本函数没有返回值，仅修改全局状态以设置随机种子。

示例：

设置随机种子为 42:

::

    setup_seed(42)

注意：

1. 调用 *np.random.seed(seed)* 为 NumPy 设置随机种子。
2. 调用 *tf.random.set_seed(seed)* 为 TensorFlow 设置 CPU 随机种子。

3. 通过设置环境变量 *os.environ['TF_DETERMINISTIC_OPS'] = '1'* 为 TensorFlow 的 GPU 操作启用确定性行为。请确保安装了 *tensorflow-determinism* 库以支持 GPU 的完全确定性操作。

login.**uploadClassifyResult**()

将分类预测结果上传到数据库，并更新对应煤数据的类型。

处理逻辑**:**

检查全局变量 *upload_classify* 是否包含分类结果。

遍历 *upload_classify* 中的每条预测结果。

根据预测结果的 *id* 更新数据库中对应记录的 *coal_type* 字段。

提交更新到数据库。

参数**:**

无（通过 POST 请求上传分类预测结果）。

返回值**:**

str: 返回上传结果的响应字符串:

“upload successfully”：表示分类结果已成功上传。

示例**:**

上传分类预测结果至数据库:

```
全局变量 `upload_classify` 包含以下数据:
[
    {
        "id": 1,
        "predicted_type": "Type A"
    },
    {
        "id": 2,
        "predicted_type": "Type B"
    }
]

请求:
POST /uploadClassifyResult

返回:
"upload successfully"
```

注意事项**:**

1. *upload_classify* 必须是一个已存储分类结果的全局变量，数据格式应包含以下字段:

   id (int): 煤数据的唯一标识。

   predicted_type (str): 分类预测的煤类型。

2. 使用 SQLAlchemy 的 *update()* 方法批量更新数据库中的记录。

3. 必须确保数据库会话 (*db.session*) 的提交成功。

login.**uploadCoalData**() → str

上传原煤数据并存储到数据库。

处理逻辑:

- 接收 POST 请求，上传煤样数据文件。
- 读取上传的 Excel 文件内容，将数据逐条解析为标准化格式。
- 根据模板字段填充每条煤样数据，数据中缺失值用 None 替代。
- 将解析后的煤样数据存储到数据库。
- 返回上传成功的消息。

参数:

无（通过 POST 请求上传 Excel 文件）。

返回值:

str: 返回“upload successfully”表示上传成功。如果发生异常，将返回错误消息。

示例:

使用 POST 请求上传煤样数据:

```
请求:
POST /UploadCoalData
上传文件: upload_coal.xlsx

Excel 文件内容示例:
+------------+-------+--------+----------+----------+-------
↪---+----------+-----+-----+
| 煤样名称      | 煤种   | 价格 (吨) | 煤样水分 (Mad) | 煤样灰分
(Ad) | 煤样挥发分 (Vdaf) | 氢 (Hdaf) | ... |
+------------+-------+--------+----------+----------+-------
↪---+----------+-----+-----+
| Coal Sample A | JM24   | 1000    | 10.5      | 15.2      | 25.
↪3      | 1.5       | ... | ... |
+------------+-------+--------+----------+----------+-------
↪---+----------+-----+-----+

返回:
"upload successfully"
```

注意事项:

- 上传文件必须为符合模板的 Excel 文件。
- 数据中的 NaN 值将替换为 None。
- 数据存储前，将字段值与模板匹配。
- 数据表中的字段包括煤种类型、价格、挥发分等。

错误处理:

- 如果文件上传失败或解析出错，将返回详细的错误信息。
- 如果数据库存储失败，将回滚事务，确保数据一致性。

login.**uploadQualityResult**()

将预测结果上传到数据库（针对 AI 算法预测的 CRI 和 CSR 值）。

处理逻辑**:**

- 检查全局变量 *upload_quality* 是否包含预测结果。
- 遍历每条预测结果，根据 *id* 查询数据库中的对应记录。
- 更新数据库记录的 *predicted_CRI* 和 *predicted_CSR* 字段。
- 提交所有更新到数据库。

参数**:**

无（通过 POST 请求上传预测结果）。

返回值**:**

str: 返回上传结果的响应字符串**:**

- “upload successfully”：表示所有预测结果已成功上传。
- 错误响应包含失败的具体原因。

示例**:**

上传预测结果至数据库:

```
假设全局变量 `upload_quality` 包含以下数据:
[
    {
        "id": 1,
        "CRI": 12.5,
        "CSR": 25.3
    },
    {
        "id": 2,
        "CRI": 15.2,
        "CSR": 30.1
    }
]

请求:
POST /uploadQualityResult

返回:
"upload successfully"
```

注意事项**:**

- *upload_quality* 必须是一个包含预测结果的全局变量。
- 每条数据应包含以下字段**:**

  - id (int): 样本的唯一标识。
  - CRI (float): 预测的 CRI 值。
  - CSR (float): 预测的 CSR 值。
- 如果事务提交失败，将回滚所有更改并返回错误信息。

login.**uploadUserClassifyResult**() → str

用户上传煤样分类结果接口。

处理逻辑:

- 检查全局变量 *upload_classify_user* 是否包含分类结果。
- 从上传的 Excel 文件中读取煤样数据。
- 合并用户分类结果和煤样数据，根据模板填充每条记录。
- 数据格式化后插入到数据库。

参数:

无（通过 POST 请求上传数据）。

返回值:

str: 返回“upload successfully”表示上传成功。如果发生异常，返回错误消息。

示例:

使用 POST 请求上传煤样分类结果:

```
请求：
POST /uploadUserClassifyResult
文件路径: ../src/assets/Upload_files/upload_coal.xlsx

返回：
"upload successfully"
```

注意事项:

- *upload_classify_user* 必须包含每条煤样的分类结果。
- 上传的 Excel 文件路径为 *../src/assets/Upload_files/upload_coal.xlsx*。
- 数据中的 NaN 值将替换为 None。
- 确保数据库事务成功提交。

错误处理:

- 如果文件读取或数据库操作失败，将返回详细的错误信息。
- 未找到分类结果时返回错误提示。

login.**uploadUserCoalData**() → str

上传用户自选煤数据接口。

处理逻辑:

- 接收 POST 请求上传用户自选的煤数据文件。

- 将上传的文件保存到指定路径，供后续分类或处理使用。

参数：
无（通过 POST 请求上传 Excel 文件）。

返回值：
str: 返回“upload successfully”表示上传成功。如果发生异常，返回错误信息。

示例：
使用 POST 请求上传煤数据:

```
请求：
POST /UploadUserCoalData
上传文件: user_coal_data.xlsx

返回：
::
"upload successfully"
```

注意事项：

- 上传的文件必须为 Excel 格式（如 *.xlsx* 或 *.xls*）。
- 文件将被保存到 *../src/assets/Upload_files/upload_coal.xlsx* 路径。
- 确保目标目录存在且具有写入权限。

错误处理：

- 如果文件上传失败或保存出错，将返回详细的错误消息。

CHAPTER

TEN

MODELPREDICT MODULE

ModelPredict.**centerresult**(*parameter*, *edge*, *r*, *x*)

ModelPredict.**distancecal**(*point1*, *point2*)

ModelPredict.**modelpredict**(*input*, *model*)

ModelPredict.**singlepred**(*data*, *model*)

ModelPredict.**valuebycenter**(*center*, *edge*, *r*)

ModelPredict.**votePre**(*input*, *reverse=True*)

CHAPTER

ELEVEN

TYPECLASSFICATION MODULE

typeClassfication.**coalcal**(*data*, *ab*, *twob*)

typeClassfication.**coaltype**(*data*, *coaltype*)

typeClassfication.**coaltypefind**(*coal*)

typeClassfication.**nanlocation**(*data1*, *data2*, *data3*)

typeClassfication.**nullreplace**(*data*, *nullmap*)

PYTHON MODULE INDEX

c

coal_quality_model_AI, **??**
coalData, **??**

d

DataProcess, **??**

e

Example, **??**

g

genetic_algorithm_geatpy1, **??**
genetic_algorithm_geatpy2, **??**

l

login, **??**

m

ModelPredict, **??**

t

typeClassfication, **??**