DataAnalytics

Release 1.0

Team GUI

Mar 27, 2025

CONTENTS

CHAPTER

ONE

INTRODUCTION

This project is a **graphical user interface (GUI)** application built using **Tkinter**, which serves as the **frontend**, integrated with a **Django-based backend** that handles **data processing** and **computation**. The application enables users to **upload files** (**CSV** or **images**) and select specific **processing methods**, leveraging different backend **APIs** to perform **Data Filtering**, **Regression and Classification**, **AI Model Execution**, and **Image Processing**. By combining **Tkinter** for **user interaction** and **Django** for **backend processing**, the project creates an efficient, **user-friendly**, and **modular system** that can handle diverse **data processing tasks** seamlessly.

1.1 Overview of the software

This software is designed to convert Excel datasets into meaningful graphical representations, enabling users to analyze and visualize data interactively based on their requirements. It integrates multiple data processing techniques, including **Data Filtering & Smoothing, Regression & Classification, AI Model Integration, and Image Processing**, to enhance the quality, accuracy, and interpretability of data-driven insights.

Users can upload Excel files (.xls, .csv) and select specific processing modules to refine, predict, classify, or visualize their data. The **Data Filtering & Smoothing** module ensures clean datasets by removing noise and handling missing values. The **Regression & Classification** module applies predictive models to analyze trends and categorize data using algorithms such as linear regression, decision trees, and neural networks. The **AI Model Integration** module enhances analysis through machine learning techniques, supporting clustering, anomaly detection, and deep learning applications. The **Image Processing** module provides functionalities for image enhancement, filtering, and segmentation if the dataset contains image-related data.

The software features an **intuitive graphical user interface (GUI)** that allows users to select visualization types (bar charts, scatter plots, histograms, heatmaps) and customize them as per their needs. AI-powered recommendations suggest optimal visualization formats based on the dataset structure. Additionally, the software enables users to **export** processed data and graphical outputs in various formats for reporting and further analysis.

With a structured workflow and modular approach, this software is an efficient tool for data analysts, researchers, and professionals looking to derive insights from Excel data through dynamic and interactive visualizations.

1.2 Purpose of Integrating Tkinter with Django

The integration of **Tkinter**, a Python-based graphical user interface (GUI) framework, with **Django**, a robust web framework, serves the objective of developing a **hybrid application** that combines a **local desktop interface** with backend processing capabilities. This approach ensures a **structured, modular, and scalable** software solution that efficiently handles **data processing, machine learning tasks, and image analysis**.

The rationale behind this integration is outlined as follows:

1.2.1 Separation of Concerns

By leveraging **Tkinter** for the user interface and **Django** for backend operations, the application follows the principle of **separation of concerns**, ensuring that the GUI remains lightweight while the computational workload is handled by Django. This architectural choice enhances **code maintainability, modularity, and ease of debugging**.

1.2.2 API-Based Communication

The application utilizes **Django REST Framework (DRF)** to facilitate seamless **communication between the frontend (Tkinter) and backend (Django)**. This API-based approach enables **structured data exchange**, allowing Tkinter to send HTTP requests (e.g., file uploads, processing requests) and receive responses efficiently. Such a mechanism supports **flexibility in backend operations** while maintaining a **decoupled system architecture**.

1.2.3 Enhanced Performance

Integrating Django as the backend significantly improves application performance by **offloading resource-intensive computations** to the server-side. Instead of processing large datasets or running complex AI models within the Tkinter frontend, these tasks are executed by Django, ensuring that the GUI remains **responsive, user-friendly, and efficient**.

1.2.4 Scalability and Extensibility

A key advantage of this integration is its **scalability**. The Django backend can be extended to support additional functionalities such as:

- **Database Integration** (e.g., PostgreSQL, MySQL) for persistent data storage.
- **Cloud-based Processing** to enhance computational efficiency.
- **Web-based Access** via Django's native capabilities, allowing future web application deployment with minimal modifications to the backend.

1.2.5 Versatile File Processing

The system supports both **structured data processing (CSV files) and image-based operations**, making it a **versatile solution for data scientists, analysts, and AI practitioners**. The ability to handle multiple data formats enhances its applicability in real-world scenarios, particularly in **machine learning pipelines and data preprocessing workflows**.

1.3 Frontend: Tkinter-Based Graphical User Interface (GUI)

The **Tkinter frontend** serves as an intuitive desktop interface, allowing users to interact with the application efficiently. Key features include:

1.3.1 File Upload Interface

- Users can upload files in different formats, including **CSV for data processing** and **images for image analysis**.

1.3.2 Automated Navigation System

- The system dynamically directs users based on the uploaded file type, ensuring a **streamlined workflow**.

1.3.3 Process Selection Module

- Users can select from multiple data processing functionalities, such as:
  - **Data Filtering & Preprocessing**
  - **Regression & Classification**

  - **AI Model Execution**
  - **Image Processing**

1.3.4 Real-Time Processing Updates

- The interface displays **real-time status updates**, including:
  - **Progress indicators**
  - **Result previews**
- This enhances **user experience** by providing immediate feedback on ongoing processes.

1.4 Backend: Django-Based API and Processing Engine

The **Django backend** serves as the core processing unit of the application, facilitating **structured data exchange, request handling, and processing operations** through a **RESTful API architecture**. The system leverages a **data object** to store user inputs before serialization into **JSON format** for transmission to the backend. The backend processes the received data and stores it in the **data object**, which is then deserialized back as a **JSON response** in the frontend.

1.4.1 Data Object and API-Based Data Exchange

The application utilizes a **data object** as an **intermediary structure** to store and manage user inputs before transmitting them to the backend. The workflow is as follows:

- **User Input Handling:** The Tkinter frontend collects input data from the user and stores it in a structured **data object**.
- **Serialization:** The data object is converted into a **JSON-formatted payload** to ensure compatibility with API communication.
- **API Transmission:** The JSON payload is sent to the **Django backend** via a **RESTful API request**.
- **Backend Processing:** The Django backend **parses the JSON data**, extracts relevant information, and executes necessary processing operations.
- **Response Generation:** The processed data is stored in the **data object**, converted back to **JSON**, and transmitted as a response.

1.4.2 REST API for Request Handling

The backend employs **Django REST Framework (DRF)** to facilitate structured communication. Key functionalities include:

- **Standardized HTTP Methods:**
  - **POST:** Receives user input in JSON format and processes the data.
  - **GET:** Returns processed results stored in the response data object.
- **Data Serialization & Deserialization:** Ensures that the **data object remains structured and accessible** throughout transmission.
- **File Handling via API Endpoints:** If file-based data is involved, the backend can **extract, process, and return file-related responses efficiently**.

1.4.3 Modular and Scalable API Architecture

The backend follows a **modular API design**, ensuring **flexibility and scalability**:

- **Modular Endpoint Design:** Each API endpoint is designed to handle a **specific function** (e.g., data validation, transformation, computation) to enhance maintainability.
- **Error Handling & Logging:** The system incorporates **robust exception handling mechanisms** to manage invalid data, request failures, and debugging logs for monitoring API performance.

CHAPTER

TWO

INSTALLATION PREREQUISITES

This project is a **Data Analytics Application** built using a **Tkinter GUI frontend** and a **Django backend**, managed with **Conda** for environment setup and reproducibility.

Before running the application, make sure you have the following installed and configured on your system:

2.1 Requirements

- Python 3.10
- Anaconda or Miniconda (for Conda environment management)

2.2 Environment Setup

1. Open **Anaconda Prompt** (or any terminal with Conda initialized).
2. Confirm Conda is installed:

```
conda --version
```

3. Navigate to the directory containing the *env_setup.yml* file.
4. Create the environment using:

```
conda env create -f env_setup.yml
```

   The *env_setup.yml* includes the following libraries:

   pandas==2.2.3, numpy==1.26.4, matplotlib==3.9.2, scikit-learn==1.4.2, Django==5.1.4, djangorestframework, requests, catboost==1.2.7, xgboost==2.1.2, tensorflow==2.16.1, statsmodels==0.14.4, customtkinter, seaborn

5. Activate the environment:

```
conda activate oopEnv
```

2.3 Running the Application

Backend

1. Navigate to the *scripts* folder:

```
cd "C:\Users\YOUR_USERNAME\Path\To\data-analytics\scripts"
```

2. Start the Django backend server:

   ```
   run_backend.bat
   ```

   This will activate the environment and start the Django development server at: http://127.0.0.1:8000/

Frontend

1. From the same *scripts* folder, run the frontend GUI:

   ```
   run_frontend.bat
   ```

   This will launch the Tkinter desktop application.

CHAPTER

THREE

STEP-BY-STEP USAGE GUIDE

This guide provides a clear walkthrough of how to use the Data Analytics UI, from uploading files and selecting modules to processing data, interpreting results, and refining your analysis. Follow each step to make the most of the platform's features.

3.1 Uploading a File and Selecting a Module

1. **Upload a file** using the file uploader on the main page.

   Supported file types: - `.csv` for structured data - `.zip` for image datasets

2. **Conditional Navigation**:

   - If a **CSV file** is uploaded:
   - The UI automatically navigates to the **Data Filtering Page**.
   - If a **ZIP file** is uploaded:
   - The UI automatically navigates to the **Image Processing Page**.

3. **For CSV files**, the user can select one of the following: - **Filtering Process** - **Scaling and Encoding**

3.2 Processing the Data and Generating Outputs

For CSV Files (Data Filtering Page):

1. **Filtering Process** (must be done step-by-step):

   a. Outlier Detection

   b. Interpolation

   c. Smoothing

   d. Scaling and Encoding

   After each step: - The user can **export the intermediate data**. - Results can be **compared** with previous steps.

2. **Direct Scaling and Encoding**: - The user can skip filtering and go straight to scaling/encoding.

3. **Send Processed Data**: - After completing either filtering or encoding, - Clicking the **Send** button forwards the data to: - **Regression & Classification Page**, or - **AI Model Page**, based on user selection.

For ZIP Files (Image Processing Page):

- The system navigates directly to the **Image Processing Module**.
- The user can: - Perform various **image processing operations** - **Set parameters** - View and **save outputs**

3.3 Interpreting Results and Refining Analysis

1. **Regression & Classification Page**: - Users set parameters - Train models - View prediction outputs and performance metrics
2. **AI Model Page**: - Select and configure AI models - Visualize and evaluate output results
3. **Image Processing Page**: - Tune processing parameters - Validate and refine image outputs
4. **Iteration and Export**: - At any point, users can: - Export current or intermediate results - Revisit and refine previous steps - Reprocess data/images for improved analysis

CHAPTER

FOUR

FRONT END: TKINTER GUI

The front end provides an intuitive graphical user interface with a multi-page layout. Each page corresponds to a specific task or process and dynamically updates based on user interaction. Key functionalities include:

- **File Uploading**:
  - Users can upload *.csv* or *.zip* files through the GUI.
  - *.csv* files are routed to data analysis processes.
  - *.zip* files are routed to image processing workflows.
- **Process Selection**:

  For *.csv* files, users choose between:
  - Data Filtering & Preprocessing
  - Regression & Classification
  - AI Model-based Analysis
- **Dynamic UI Rendering**:

  UI elements (e.g., sliders, dropdowns, radio buttons) change depending on the selected process or model type.
- **Result Display**:

  Graphs (e.g., regression plots, residuals, confusion matrices), metrics, and predictions are rendered directly within the GUI using embedded plotting libraries such as `matplotlib`.

`class main_sphinx.`**`App`**

Bases: `CTk`

Main application class for the Tkinter-based GUI.

This class initializes the GUI, handles page navigation, and manages assets.

`current_page`

Stores the current active page in the application.

Type

ctk.CTkFrame

`configure_grid()`

Configures the layout grid of the application.

This method sets up the grid structure for proper UI arrangement.

create_sidebar()

Creates the sidebar navigation panel.

This method initializes navigation buttons for different pages.

load_assets()

Loads all image assets used in the application.

This method loads required images for the UI elements.

load_header_image()

Loads and resizes the header image.

This method retrieves the header image, resizes it, and assigns it to the header label.

resize_header_image(*event*)

Resizes the header image dynamically when the window is resized.

Parameters

event (`tk.Event`) – The resize event that triggers this function.

show_page(*page_name*, **args*, ***kwargs*)

Handles navigation between different pages.

Parameters

- **page_name** (`str`) – The name of the page to display.
- ***args** (`tuple`) – Additional arguments passed to the page class.
- ****kwargs** (`dict`) – Keyword arguments passed to the page class.

toggle_mode()

Toggles between Light and Dark mode.

class pages.aimodel_page.**AImodelPage**(*parent*, *file_data=None*, *file_name=None*, **args*, ***kwargs*)

Bases: `CTkFrame`

A custom Tkinter frame for configuring and submitting AI models.

This class provides a GUI interface for selecting and configuring machine learning models (RandomForest, CatBoost, Artificial Neural Network, XGBoost), adjusting their hyperparameters, and submitting them to a backend for training or evaluation.

Parameters

- **parent** – The parent tkinter container.
- **file_data** – The processed data to be used by the models.
- **file_name** – The name of the file associated with the data.

change_segment(*segment_name*)

Switch between the different AI model configuration segments.

Parameters

segment_name – The selected model name from the segmented button.

create_ann_frame()

Create the UI frame for configuring Artificial Neural Network (ANN) hyperparameters.

Returns

A CTkFrame widget containing sliders and dropdowns for ANN.

`create_cb_frame()`

Create the UI frame for configuring CatBoost model hyperparameters.

Returns
A CTkFrame widget containing sliders for CatBoost.

`create_combobox_frame`(*parent*, *label_text*, *options*, *default*, *row*)

Create a dropdown combobox for selecting categorical parameters.

Parameters

- **parent** – The parent widget to attach this frame.
- **label_text** – The label describing the dropdown.
- **options** – List of available options.
- **default** – Default selected option.
- **row** – Row position inside the parent grid.

`create_info_button`(*parent*, *text*)

Create a small button with an info icon that triggers a popup.

Parameters

- **parent** – The parent widget to attach this button.
- **text** – The info message to display when clicked.

`create_rf_frame()`

Create the UI frame for configuring RandomForest model hyperparameters.

Returns
A CTkFrame widget containing sliders for RandomForest.

`create_slider_frame`(*parent*, *model_name*, *label_text*, *from_*, *to*, *default*, *row*)

Create a slider UI element for numeric hyperparameters.

Parameters

- **parent** – The parent widget to attach this frame.
- **model_name** – Name of the model this parameter belongs to.
- **label_text** – The label describing the slider parameter.
- **from** – Minimum value for the slider.
- **to** – Maximum value for the slider.
- **default** – Default value of the slider.
- **row** – Row position inside the parent grid.

`create_xgb_frame()`

Create the UI frame for configuring XGBoost model hyperparameters.

Returns
A CTkFrame widget containing sliders for XGBoost.

`get_combobox_value`(*combobox_name*)

Helper to get the value of a specific combobox.

Parameters
combobox_name – The name of the combobox.

Returns
The selected value if found, else None.

`get_slider_value`(*slider_name*)

Helper to get the value of a specific slider.

Parameters
`slider_name` – The name of the slider.

Returns
Slider value if found, else 0.

`messagebox = <module 'tkinter.messagebox' from 'C:\\Users\\USER\\.conda\\envs\\oopDevEnv\\lib\\tkinter\\messagebox.py'>`

`preview_data`()

Open a popup window displaying the first 50 rows of the processed dataset. Provides a scrollable view using ttk.Treeview.

`send_request`(*json_data*)

Send a POST request with the selected model data to the Django backend.

Parameters
`json_data` – The full dictionary containing model configuration and dataset.

`show_info_dialog`(*text*)

Show an information popup with a provided text.

Parameters
`text` – The info message to display in the dialog.

`submit_action`()

Handle the submit button click by collecting all model configurations, packaging them into a JSON structure, and sending a request to the backend.

`class pages.data_filtering_page.DataFilteringPage`(*parent*, *file_path*, *file_name=''*)

Bases: `CTkFrame`

A Tkinter-based GUI page for performing various data filtering processes including Outlier Detection, Interpolation, Smoothing, and Scaling & Encoding.

Parameters

- `parent` – The main application container.
- `file_path` – Path to the uploaded CSV file.
- `file_name` – Name of the file (optional).

`add_export_send_buttons`()

Adds export, compare, and send buttons at the bottom of the UI.

`change_segment`(*segment_name*)

Changes the active segment based on user navigation.

Parameters
`segment_name` – Name of the segment to activate.

`create_interpolation_frame`()

Creates the Interpolation segment with radio button method selection.

Returns
CTkFrame with interpolation method options.

`create_process_selection_frame()`

Creates the initial selection frame for choosing filtering vs scaling.

Returns
CTkFrame with radio button options.

`create_scaling_encoding_frame()`

Creates the Scaling & Encoding segment with sliders for test size and random state.

Returns
CTkFrame containing configuration widgets.

`create_segment_frame()`

Creates the Outlier Detection frame with method options and slider.

Returns
CTkFrame containing radio buttons, slider, and column checkboxes.

`create_smoothing_frame()`

Creates the Smoothing segment, including SMA and TES options.

Returns
CTkFrame with smoothing controls.

`export_data()`

Exports processed data to CSV depending on the most completed step.

`get_data_by_name(`*name*`)`

Fetches a DataFrame based on the dataset name string.

Parameters
`name` – Dataset name like “Raw Data”, “Interpolated Data” etc.

Returns
Corresponding Pandas DataFrame.

`load_csv_columns(`*file_path*`)`

Loads column names from CSV and sets up dropdowns and checkboxes.

Parameters
`file_path` – Path to the CSV file.

`lock_segment(`*frame*`)`

Locks a segment UI to prevent changes after completion.

Parameters
`frame` – The segment frame to disable.

`move_to_next_segment()`

Moves the user to the next logical segment based on progress. Updates visibility of segment buttons.

`open_comparison_popup()`

Opens a popup window to compare different stages of data processing.

`open_send_popup()`

Opens a destination selection popup after scaling & encoding.

`plot_boxplot`(*column_name*, *cleaned=False*)

Generates a boxplot for a selected column with optional cleaned data.

Parameters

- **`column_name`** – Name of the column to visualize.
- **`cleaned`** – Whether to use cleaned data or not.

`plot_comparison_graph`(*left_data*, *right_data*, *column_name*)

Plots graph comparison of two data stages using boxplot or line chart.

Parameters

- **`left_data`** – Name of the left dataset.
- **`right_data`** – Name of the right dataset.
- **`column_name`** – Column to plot.

`plot_line_graph`(*column_name*, *original_data*, *processed_data*, *title*)

Plots line graphs comparing original and processed data.

Parameters

- **`column_name`** – Name of the data column.
- **`original_data`** – The original data series.
- **`processed_data`** – The processed data series.
- **`title`** – Plot title.

`preview_csv`()

Opens a popup window showing the first 50 rows of the uploaded CSV.

`preview_scaled_encoded_data`()

Displays the scaled and encoded data in a table popup.

`run_interpolation`()

Executes interpolation on cleaned data and updates graph.

`run_outlier_detection`()

Triggers outlier detection and updates cleaned data based on user selections.

`run_scaling_and_encoding`()

Executes scaling and encoding on the latest available dataset (raw or processed).

`run_smoothing`()

Applies smoothing (SMA or TES) on interpolated data and updates visuals.

`send_request`(*process_name*, *json_data*)

Sends a request to the backend with specified process name and data.

Parameters

- **`process_name`** – The name of the backend API process.
- **`json_data`** – The data payload as dictionary.

Returns

Parsed JSON response from backend.

`show_comparison_data`(*left_data*, *right_data*, *column_name*)

Displays side-by-side table comparison of two datasets.

Parameters

- **`left_data`** – Name of the first dataset.
- **`right_data`** – Name of the second dataset.
- **`column_name`** – Column to compare.

`show_info_dialog`(*text*)

Displays an information popup with help text.

Parameters

`text` – The text to show in the popup.

`show_loading_message`()

Displays a loading message in the graph area.

`submit_action`()

Handles submission logic for each segment and moves to the next step. Validates inputs and triggers processing functions.

`toggle_slider`(*frame*, *show*)

Show or hide contamination slider based on outlier method.

Parameters

- **`frame`** – The parent frame.
- **`show`** – Boolean to show or hide.

`toggle_smoothing_options`(*frame*, *method*)

Toggles visibility of smoothing options based on selected method.

Parameters

- **`frame`** – The parent frame.
- **`method`** – Either 'SMA' or 'TES'.

`update_boxplot`(*column_name*)

Asynchronously updates the boxplot for a selected column.

Parameters

`column_name` – The name of the column to plot.

`update_buttons_visibility`()

Updates visibility of export, compare, and send buttons.

`update_comparison_view`()

Handles the action of switching between data and graph views in the compare popup.

`update_dropdown`(*selected_columns*)

Updates the dropdown values and behavior based on selected columns.

Parameters

`selected_columns` – List of column names to include in dropdown.

class pages.file_upload_page.**FileUploadPage**(*parent*)

Bases: `CTkFrame`

File upload interface for CSV and ZIP files. Routes uploaded files to the appropriate next page depending on file type.

Parameters
`parent` – The parent application container.

upload_file()

Handles file upload via a file dialog and routes to the correct page based on file type (.csv or .zip). Stores file path and name in the main application state.

class pages.image_processing_page.**ImageProcessingPage**(*parent*, *file_path=None*, *file_name=None*, *data=None*, ***page_state*)

Bases: `CTkFrame`

CustomTkinter page for training an image classification model. Allows user to configure activation, optimizer, and dataset splitting.

Parameters

- `parent` – Parent widget.
- `file_path` – Path to the dataset ZIP.
- `file_name` – Optional display name.
- `data` – Placeholder for future data passing.
- `page_state` – Page-specific settings (e.g. saved state).

cancel_file()

Handles file cancellation and resets only this page.

change_segment(*segment_name*)

Switches UI to show only the active segment.

Parameters
`segment_name` – Label of the selected segment.

create_image_train_frame()

Builds a small image upload and preview block.

Returns
CTkFrame containing upload and preview buttons.

create_segment_frame()

Creates UI segment to configure activation, epochs, optimizer, test size, and random seed.

Returns
CTkFrame with interactive settings.

plot_confusion_matrix(*cm_data*)

Renders a matplotlib confusion matrix inside a popup.

Parameters
`cm_data` – Dictionary with matrix values and labels.

preview_image()

Opens a new preview window with the uploaded image. Resizes image to fit window if necessary.

`show_info_dialog`(*text*)

Displays a popup window with explanatory text.

Parameters
`text` – Message content to show.

`submit_action`()

Sends selected image classification settings to the backend for training. Logs training info and receives metrics for display.

`upload_image`()

Opens a file picker dialog and stores uploaded image path. Enables preview button after selection.

class `pages.process_selection_page.ProcessSelectionPage`(*parent*, *file_path*, *filename*)

Bases: `CTkFrame`

This module implements the Process Selection Page using CustomTkinter. It allows users to choose between Data Filtering, Regression & Classification, and AI Model training.

A GUI page that allows users to select the next processing step for the uploaded file.

`parent`

The parent application window.

Type
ctk.CTk

`file_path`

Path of the uploaded file.

Type
str

`filename`

Name of the uploaded file.

Type
str

`__init__`(*parent*, *file_path*, *filename*)

Initializes the Process Selection Page UI.

`create_buttons`(*parent*, *file_path*, *filename*)

Creates buttons for different processing options.

Parameters

- **parent** (*ctk.CTk*) – The parent application window.
- **file_path** (*str*) – Path of the uploaded file.
- **filename** (*str*) – Name of the uploaded file.

class `pages.regression_classification.RegressionClassificationPage`(*parent*, *file_path=None*, *file_name=None*, *data=None*, ***page_state*)

Bases: `CTkFrame`

GUI page for configuring and submitting regression and classification models. Supports various regression (Linear, Polynomial, Ridge, Lasso) and classification (RandomForest, SVC, KNN) models.

Parameters

- `parent` – Main parent application.
- `file_path` – Path to the uploaded file (optional).
- `file_name` – Name of the uploaded file.
- `data` – Preprocessed dataset.
- `page_state` – State dictionary for restoring prior settings (if any).

`cancel_file()`

Cancels the current file and resets this page state to its default. Clears file paths, sidebar references, and page data.

`change_segment`(*segment_name*)

Handles switching between Regression and Classification segments.

Parameters
`segment_name` – Either 'Regression' or 'Classification'.

`clear_frame`(*frame*)

Clears the contents of a given frame.

Parameters
`frame` – The CTkFrame widget to be cleared.

`create_classification_frame()`

Creates the Classification segment frame with dynamic parameters.

Returns
A CTkFrame containing radio buttons and config for classification models.

`create_dropdown`(*parent*, *label_text*, *options*)

Creates a labeled dropdown menu.

Parameters

- `parent` – Container for the dropdown.
- `label_text` – Label to display above the dropdown.
- `options` – List of dropdown options.

`create_info_button`(*parent*, *text*)

Creates an info button to show contextual information.

Parameters

- `parent` – The frame to place the button in.
- `text` – Info text to show when clicked.

`create_regression_frame()`

Creates the Regression segment frame with dynamic parameters.

Returns
A CTkFrame containing radio buttons and config for regression models.

`create_slider`(*parent*, *label_text*, *min_val*, *max_val*, *default*)

Creates a labeled slider with value label and info button.

Parameters

- `parent` – The container frame.

- `label_text` – Text to describe the slider.
- `min_val` – Minimum value.
- `max_val` – Maximum value.
- `default` – Default slider position.

`create_textbox`(*parent*, *label_text*, *mode*)

Creates a labeled textbox for custom input parameters.

Parameters

- `parent` – The container frame.
- `label_text` – The textbox label.
- `mode` – Input mode ('polynomial' or 'alpha') for validation. - "polynomial": Allows 1-5 single-digit numbers (0-9), separated by commas. - "alpha": Allows 1-5 float values (0-1) with at least 4 decimal places, separated by commas.

`lasso_plot`(*results_lasso*, *best_params*)

Plots Lasso Regression performance with alpha values against R2 scores.

Parameters

- `results_lasso` – Dictionary containing cross-validation results.
- `best_params` – Dictionary with best_degree_lasso and best_alpha_lasso.

`polynomial_plot`(*x_scatter*, *y_scatter*, *y_poly*, *x_label*, *y_label*, *degree*)

Generates and displays a polynomial regression plot overlaying actual vs predicted values.

Parameters

- `x_scatter` – Input feature values.
- `y_scatter` – Actual target values.
- `y_poly` – Predicted values from polynomial regression.
- `x_label` – Label for x-axis.
- `y_label` – Label for y-axis.
- `degree` – Polynomial degree used in the model.

`preview_data`()

Opens a new popup window to display the scaled and encoded data.

`regression_plot`(*x*, *y*, *x_label*, *y_label*, *data=None*, *ax=None*)

Displays a linear regression plot with regression line and scatter points.

Parameters

- `x` – Feature values (can be list, dict, or array).
- `y` – Target values (can be list, dict, or array).
- `x_label` – Label for x-axis.
- `y_label` – Label for y-axis.
- `data` – Optional dataset (not used).
- `ax` – Matplotlib axis to draw on.

`residual_plot`(*x*, *y*, *ax=None*)

Creates a residual plot showing prediction errors.

Parameters

- `x` – Predicted values (can be list, dict, or array).
- `y` – Actual target values (can be list, dict, or array).
- `ax` – Optional Matplotlib axis.

`ridge_plot`(*results_ridge*, *best_params*)

Plots Ridge Regression performance with alpha values against R2 scores.

Parameters

- `results_ridge` – Dictionary containing cross-validation results.
- `best_params` – Dictionary with best_degree_ridge and best_alpha_ridge.

`send_request_classification`(*json_data*)

Sends a POST request with classification model data to the Django backend.

Parameters
`json_data` – JSON-serializable data object to send to the backend.

`send_request_regression`(*json_data*)

Sends a POST request with regression model data to the Django backend. Handles and visualizes results for Linear, Polynomial, Ridge, and Lasso regressions.

Parameters
`json_data` – JSON-serializable data object to send to the backend.

`show_info_dialog`(*text*)

Displays an information popup.

Parameters
`text` – The text to show in the info dialog.

`submit_action`()

Handles submission of configured model parameters. Builds DataObject and sends it to backend.

`toggle_classification_options`()

Updates the Classification options dynamically based on selection.

`toggle_regression_options`()

Updates the Regression options dynamically based on selection.

`class pages.help_page.HelpPage`(*parent*)

Bases: `CTkFrame`

This module implements a simple Help Page using CustomTkinter. A class representing the Help Page in the GUI.

This page provides user assistance and guidance.

`None`

`__init__`(*parent*)

Initializes the Help Page.

CHAPTER

FIVE

BACK END: DJANGO API

The Django backend serves as the processing engine and is structured around modular views and serializers. Key responsibilities include:

- **Process Handling**:

  Performs data transformations, model training, predictions, and visualizations.

- **Modular APIs**:

  Four dedicated API routes handle distinct functionality:

  1. **Data Filtering & Preprocessing**
  2. **Regression & Classification**
  3. **AI Models**
  4. **Image Processing**

`class backend.api.image_processing_engine.`**`ImageProcessingAPIView`**`(`*`*args`*`, `*`**kwargs`*`)`

Bases: `APIView`

API endpoint to process zipped image datasets for training a CNN-based image classifier.

The pipeline includes: 1. Loading and preprocessing zipped image data 2. Splitting dataset 3. Creating a CNN model 4. Training the model 5. Evaluating test accuracy/loss 6. Generating a confusion matrix

`post``(`*`request`*`)`

Handle POST request containing DataObject with zipped image dataset and model parameters.

Expects structure: - dataobject[image_processing][fileio, train_test_split, model_params, training_params]

Parameters
`request` – HTTP request from frontend.

Returns
Response containing test accuracy, loss, and confusion matrix.

`class backend.api.data_preprocessing_engine.`**`DataFilteringFileAPIView`**`(`*`*args`*`, `*`**kwargs`*`)`

Bases: `APIView`

`post``(`*`request`*`)`

`run_outlier_detection``(`*`dataset`*`, `*`data_object`*`)`

Runs the outlier detection based on the method defined in DataObject.

`class backend.api.data_preprocessing_engine.`**`InterpolationAPIView`**`(`*`*args`*`, `*`**kwargs`*`)`

Bases: `APIView`

`post`(*request*)

`run_interpolation`(*cleaned_outlier_data*)

Runs the cubic spline interpolation on the dataset after outlier detection.

`class backend.api.data_preprocessing_engine.`**`SmoothingAPIView`**(**args*, ***kwargs*)

Bases: `APIView`

`post`(*request*)

`run_smoothing`(*interpolated_data*, *data_object*)

Runs the smoothing process based on the method defined in DataObject.

`class backend.api.ai_models_engine.`**`AIModelAPIView`**(**args*, ***kwargs*)

Bases: `APIView`

API endpoint to handle training and evaluation of AI models.

Supports classification and regression models: - RandomForest - CatBoost - Artificial Neural Network (ANN) - XGBoost

Uses user-selected hyperparameters and preprocessed data to fit and evaluate the chosen model.

`extract_hyperparameters`(*data_object*, *model_name*)

Extract user-defined hyperparameters from the data_object or fallback to defaults.

Parameters

- `data_object` – Structured DataObject containing model info.
- `model_name` – Name of the selected model.

Returns

Dictionary of validated hyperparameters.

`post`(*request*)

Handle POST request from the frontend containing a *dataobject*.

Parameters

`request` – HTTP request object with model input and configuration.

Returns

Response with model evaluation results.

`run_selected_model`(*data_object*)

Orchestrates the training and evaluation pipeline for the selected model.

Converts JSON lists back into NumPy arrays, instantiates the model class, trains it, evaluates performance, and returns metrics.

Parameters

`data_object` – DataObject containing training data and parameters.

Returns

Dictionary of results such as accuracy, confusion matrix, or regression scores.

`class backend.api.scaling_encoding_engine.`**`ScalingEncodingAPIView`**(**args*, ***kwargs*)

Bases: `APIView`

API endpoint for scaling, encoding, and train-test splitting of preprocessed data.

This endpoint accepts smoothed data and configuration from a DataObject, applies encoding and scaling transformations, splits the dataset, and returns the processed splits ready for training.

`post`(*request*)

Handle POST request with a DataObject and smoothed dataset.

Performs encoding, scaling, and train-test split using provided parameters.

Parameters
`request` – HTTP request with JSON-encoded data and smoothed data.

Returns
Response containing split, encoded, and scaled data.

`run_encoding_scaling_train_test_split`(*data*, *params*)

Perform encoding, feature scaling, and split into training/testing sets.

Parameters

- `data` – The input pandas DataFrame.
- `params` – Parameters for preprocessing such as scalers, encoders, test size.

Returns
Dictionary containing split DataFrames for X_train, X_test, y_train, y_test.

`class backend.api.regression_engine.RegressionAPIView`(**args*, ***kwargs*)

Bases: `APIView`

API endpoint for training and evaluating regression models on preprocessed data.

Supports: - Linear Regression - Polynomial Regression - Ridge Regression - Lasso Regression

Takes preprocessed training/test sets and regression configuration from a frontend DataObject. Returns appropriate scores, predictions, and hyperparameters.

`post`(*request*)

Handle POST request with dataobject that includes: - Train/test split (X_train, y_train, etc.) - Selected model - Model-specific parameters

Automatically trains and evaluates the chosen model, and returns regression metrics.

Parameters
`request` – HTTP request containing a JSON-serialized DataObject.

Returns
JSON response with evaluation results including R2 score, predictions, and parameters.

`class backend.api.classification_engine.ClassificationAPIView`(**args*, ***kwargs*)

Bases: `APIView`

API endpoint to handle training and evaluation of classification models.

Supported Models:

- Random Forest
- Support Vector Classifier (SVC)
- K-Nearest Neighbors (KNN)

Accepts data and model parameters via POST request and returns evaluation metrics.

`post`(*request*)

Handle POST request to train and evaluate a classification model.

Expects a JSON request with model configuration and split training/testing data inside a *dataobject* structure.

Example

{“dataobject”: {“classification”: {“Model_Selection”: “RandomForest”,”RandomForest”: {“n_estimators”: 100,”max_depth”: 5}},”data_filtering”:”Train-Test Split”: {“split_data”: {“X_train”: […],”X_test”: […],”y_train”: […],”y_test”: […]}}}}}

Parameters
`request` (*Request*) – Django REST framework request containing the dataobject.

Returns

A JSON response with model evaluation results including:

- accuracy (float)
- confusion matrix (list)
- mean squared error (float)

Return type
Response

Raises
`ValueError` – If an invalid model name is provided.

CHAPTER

SIX

DATA FILTERING

class `data_filtering.Outlier_final.`**`OutlierDetection`**(*data*)

Bases: `object`

`detect_outliers_iqr`(*dataset*, *column_names*)

Replaces outliers with NaN using the IQR method.

`detect_outliers_isolation_forest`(*dataset*, *contamination*, *column_names*)

Replaces outliers with NaN using Isolation Forest.

`is_numeric_columns`(*dataset*, *column_names*)

Checks if the values in the given columns are numeric.

class `data_filtering.Smoothing_final.`**`SmoothingMethods`**(*data*)

Bases: `object`

`apply_tes`(*data*, *seasonal_periods*, *trend*, *seasonal*, *smoothing_level*, *smoothing_trend*, *smoothing_seasonal*)

Apply Triple Exponential Smoothing (TES) to numeric columns.

`calculate_sma`(*data*, *window*)

Apply Simple Moving Average (SMA) to numeric columns.

class `data_filtering.Spline_Interpolation_final.`**`SplineInterpolator`**(*data*)

Bases: `object`

`check_column_validity`(*column_name*)

Checks if a column can be interpolated (at least 2 known numeric values).

`fill_missing_values`()

Applies cubic spline interpolation to fill missing values in numeric columns.

class `data_filtering.Scaling_and_Encoding_final.`**`EncodeAndScaling`**(*data*)

Bases: `object`

`encode_categorical_features`(*feature_data*)

`preprocess`(*data_object*)

Runs encoding, scaling, and train-test splitting on the dataset.

`scale_numerical_features`(*encoded_data*) `train_test_split`(*processed_data*, *target_column*, *test_size*, *random_state*)

Splits the processed dataset into training and testing sets. :type processed_data: :param processed_data: The encoded and scaled dataset. :type test_size: :param test_size: Proportion of dataset to use as the test set (default 20%). :type random_state: :param random_state: Random seed for reproducibility. :return: Training and testing datasets.

CHAPTER SEVEN

REGRESSION

`class regression.regression_models.`**`RegressionModels`**

Bases: `object`

A class to train different regression models including Linear, Polynomial, Ridge, and Lasso regression.

`model`

The trained regression model (e.g., LinearRegression, Pipeline with PolynomialFeatures and regression).

`best_params`

The best hyperparameters found during grid search for the respective model (if applicable).

`train_linear_regression`(*dataobj*)

Trains a Linear Regression model.

`train_polynomial_regression`(*dataobj*, *param_grid=None*, *cv=5*)

Trains a Polynomial Regression model with grid search.

`train_ridge`(*dataobj*, *param_grid=None*, *cv=5*)

Trains a Ridge Regression model with polynomial features and grid search.

`train_lasso`(*dataobj*, *param_grid=None*, *cv=5*)

Trains a Lasso Regression model with polynomial features and grid search.

`train_lasso`(*dataobj*, *param_grid=None*, *cv=3*, *subsample_ratio=0.3*)

Train a Lasso Regression model with polynomial features and hyperparameter tuning using GridSearchCV, utilizing a subsample of the training data for efficiency.

Parameters

- **`dataobj`** (*`dict`*) – A dictionary containing split data with keys 'split_data' which further contains 'X_train' and 'y_train' for training features and target variables respectively (expected as pandas DataFrames or Series).
- **`param_grid`** (*`dict, optional`*) – Dictionary with parameter names (string) as keys and lists of parameters as values, e.g., for polynomial degree and Lasso alpha. Defaults to None.
- **`cv`** (*`int, optional`*) – Number of cross-validation folds. Defaults to 3.
- **`subsample_ratio`** (*`float, optional`*) – Fraction of training data to use for hyperparameter tuning. Must be between 0 and 1. Defaults to 0.3.

Returns

The trained Lasso Regression model (best estimator from grid search).

Return type

Pipeline

`self.model`

Stores the trained Lasso Regression model (best estimator).

`self.best_params_lasso`

Stores the best hyperparameters found during grid search.

`self.results_lasso`

Stores the cross-validation results from grid search.

`self.best_degree_lasso`

Stores the best polynomial degree from the best parameters.

Notes

- Subsampling is performed randomly without replacement to reduce computational load.
- The Lasso regression is configured with a maximum of 2000 iterations and a tolerance of 1e-2.

`train_linear_regression`(*dataobj*)

Train a Linear Regression model using the provided training data.

Parameters

dataobj (*dict*) – A dictionary containing split data with keys 'split_data' which further contains 'x_train' and 'y_train' for features and target variables respectively.

Returns

The trained Linear Regression model.

Return type

LinearRegression

`self.model`

Stores the trained Linear Regression model.

`train_polynomial_regression`(*dataobj*, *param_grid=None*, *cv=5*)

Train a Polynomial Regression model with optional hyperparameter tuning using GridSearchCV.

Parameters

- **dataobj** (*dict*) – A dictionary containing split data with keys 'split_data' which further contains 'x_train' and 'y_train' for features and target variables respectively.
- **param_grid** (*dict*) – Dictionary with parameters names (string) as keys and lists of parameter as values. Defaults to None.
- **cv** (*int, optional*) – Number of cross-validation folds. Defaults to 5.

Returns

The trained Polynomial Regression model (best estimator from grid search).

Return type

Pipeline

`self.model`

Stores the trained Polynomial Regression model (best estimator).

`self.best_params_poly`

Stores the best hyperparameters found during grid search.

`train_ridge`(*dataobj*, *param_grid=None*, *cv=3*, *subsample_ratio=0.3*)

Train a Ridge Regression model with polynomial features and hyperparameter tuning using GridSearchCV, utilizing a subsample of the training data for efficiency.

Parameters

- `dataobj` (*dict*) – A dictionary containing split data with keys 'split_data' which further contains 'X_train' and 'y_train' for training features and target variables respectively (expected as pandas DataFrames or Series).
- `param_grid` (*dict, optional*) – Dictionary with parameter names (string) as keys and lists of parameters as values, e.g., for polynomial degree and Ridge alpha. Defaults to None.
- `cv` (*int, optional*) – Number of cross-validation folds. Defaults to 3.
- `subsample_ratio` (*float, optional*) – Fraction of training data to use for hyperparameter tuning. Must be between 0 and 1. Defaults to 0.3.

Returns

The trained Ridge Regression model (best estimator from grid search).

Return type

Pipeline

`self.model`

Stores the trained Ridge Regression model (best estimator).

`self.best_params_ridge`

Stores the best hyperparameters found during grid search.

`self.results_ridge`

Stores the cross-validation results from grid search.

`self.best_degree_ridge`

Stores the best polynomial degree from the best parameters.

Notes

- Subsampling is performed randomly without replacement to reduce computational load.
- The Ridge regression is configured with a maximum of 2000 iterations and a tolerance of 1e-2.

CHAPTER EIGHT

CLASSIFICATION

Module: base_model.py Description: Provides a base class for evaluating classification models.

`class classification.base_model.ClassifierClass`(*data_train*, *data_test*, *target_train*, *target_test*, *target_labels*)

Bases: `object`

Base class for evaluating classification models.

`data_train`

Training feature data.

Type

array-like

`data_test`

Testing feature data.

Type

array-like

`target_train`

Training target labels.

Type

array-like

`target_test`

Testing target labels.

Type

array-like

`target_labels`

List of class labels.

Type

list

`display_confusion_matrix`(*cm*)

Display the confusion matrix as a percentage plot.

Parameters

cm (*np.array*) – The confusion matrix to display.

evaluate(*model*)

Fit the model on training data, predict on test data, and compute evaluation metrics.

Parameters
`model` – A machine learning model instance.

Returns
Contains accuracy, classification report, confusion matrix, and mean squared error.

Return type
tuple

set_model(*model*)

Set the model for evaluation.

Parameters
`model` – A machine learning model instance.

Module: knn_model.py Description: Implements the K-Nearest Neighbors (KNN) classification model with grid search. Best Parameters for KNN: {'n_neighbors': 3, 'p': 1, 'weights': 'uniform'}

class classification.knn_model.**KNNModel**(*data_train*, *data_test*, *target_train*, *target_test*, *target_labels*)

Bases: *ClassifierClass*

KNNModel uses K-Nearest Neighbors algorithm for classification and inherits from ClassifierClass.

train()

Train the KNN model using GridSearchCV to determine the best parameters, then prompt the user for final parameter selection and train the model.

Module: random_forest_model.py Description: Implements the Random Forest classification model with grid search parameter tuning. Best Parameters for RandomForest: {'max_depth': 20, 'n_estimators': 150}

class classification.random_forest_model.**RandomForestModel**(*data_train*, *data_test*, *target_train*, *target_test*, *target_labels*)

Bases: *ClassifierClass*

RandomForestModel uses a Random Forest algorithm for classification and inherits from ClassifierClass.

train()

Train the Random Forest model using GridSearchCV to determine the best parameters, then prompt the user for final parameter selection and train the model.

Module: svc_model.py Description: Implements the Support Vector Classifier (SVC) model with grid search parameter tuning. Best Parameters for SVC: {'C': 1, 'gamma': 'scale', 'kernel': 'rbf'}

class classification.svc_model.**SVCModel**(*data_train*, *data_test*, *target_train*, *target_test*, *target_labels*)

Bases: *ClassifierClass*

SVCModel uses the Support Vector Classifier for classification and inherits from ClassifierClass.

train()

Train the SVC model using GridSearchCV to determine the best parameters, then prompt the user for final parameter selection and train the model.

CHAPTER

NINE

AI MODULE

class ai_model.ann.**ArtificialNeuralNetwork**(*problem_type='classification'*, *options=None*)

Bases: `BaseModel`

Implements an Artificial Neural Network (ANN) model for classification.

model

The ANN model instance.

Type

Sequential

problem_type

Type of problem (only "classification" is supported).

Type

str

batch_size

Number of samples per batch during training.

Type

int

epochs

Number of training iterations.

Type

int

evaluate()

Evaluates the trained model on the test data.

Returns

A dictionary containing evaluation metrics: - `Accuracy` (float): Accuracy score of the model. - `Confusion Matrix` (list): Confusion matrix representing classification performance.

Return type

dict

Raises

ValueError – If test data is missing.

load_weights(*filepath='ann_weights.h5'*)

Loads pre-trained model weights from a file.

Parameters
`filepath` (*str, optional*) – Path from where the weights will be loaded (default is "ann_weights.h5").

Raises

- `FileNotFoundError` – If the specified file does not exist.
- `Exception` – If an error occurs while loading weights.

`save_weights`(*filepath='ann_weights.h5'*)

Saves the trained model's weights to a file.

Parameters
`filepath` (*str, optional*) – Path where the weights will be saved (default is "ann_weights.h5").

Raises
`Exception` – If an error occurs while saving weights.

`train`()

Trains the ANN model using the provided training data.

Raises
`ValueError` – If training data is missing.

class ai_model.base.`BaseModel`(*model*, *problem_type='classification'*)

Bases: `object`

Base class for all models that centralizes hyperparameter validation and common functionalities like data splitting, training, and evaluation.

`HYPERPARAMETER_RANGES`

Defines the valid range of hyperparameters for different model types.

Type
dict

`model`

The machine learning model instance.

Type
object

`problem_type`

Type of problem (either "classification" or "regression").

Type
str

`X_train`

Training feature dataset.

Type
pandas.DataFrame or None

`X_test`

Testing feature dataset.

Type
pandas.DataFrame or None

`y_train`

Training target dataset.

Type
pandas.Series or None

`y_test`

Testing target dataset.

Type
pandas.Series or None

`HYPERPARAMETER_RANGES = {'ArtificialNeuralNetwork': {'activation': {'allowed': ['relu', 'sigmoid', 'tanh', 'softmax'], 'default': ['relu', 'relu', 'softmax']}, 'batch_size': {'default': 30, 'max': 128, 'min': 16}, 'epochs': {'default': 100, 'max': 300, 'min': 10}, 'layer_number': {'default': 3, 'max': 6, 'min': 1}, 'optimizer': {'allowed': ['adam', 'sgd', 'rmsprop'], 'default': 'adam'}, 'units': {'default': [128, 64, 4], 'max': 256, 'min': 1}}, 'CatBoost': {'learning_rate': {'default': 0.03, 'max': 0.1, 'min': 0.01}, 'max_depth': {'default': 6, 'max': 10, 'min': 4}, 'n_estimators': {'default': 500, 'max': 1000, 'min': 100}, 'reg_lambda': {'default': 3, 'max': 10, 'min': 1}}, 'RandomForest': {'max_depth': {'default': 20, 'max': 50, 'min': 3}, 'min_samples_leaf': {'default': 1, 'max': 10, 'min': 1}, 'min_samples_split': {'default': 5, 'max': 10, 'min': 4}, 'n_estimators': {'default': 200, 'max': 500, 'min': 10}}, 'XGBoost': {'learning_rate': {'default': 0.3, 'max': 0.3, 'min': 0.01}, 'max_depth': {'default': 6, 'max': 10, 'min': 0}, 'min_split_loss': {'default': 10, 'max': 10, 'min': 3}, 'n_estimators': {'default': 200, 'max': 1000, 'min': 100}}}`

`static validate_options`(*options*, *model_type*)

Validates and ensures that the provided hyperparameters fall within allowed ranges.

Parameters

- `options` (*dict*) – The dictionary containing model hyperparameters.
- `model_type` (*str*) – The type of model being validated.

Returns
A dictionary of validated hyperparameters with out-of-range values replaced with defaults.

Return type
dict

Raises
`Exception` – If an unexpected error occurs during validation.

`class ai_model.catboost_model.Catboost`(*problem_type='classification'*, *options=None*)

Bases: `BaseModel`

Implements the CatBoost model for both classification and regression, ensuring hyperparameters are validated before model initialization.

`problem_type`

Specifies whether the model is for classification or regression.

Type
str

`options`
Contains hyperparameters such as *n_estimators*, *learning_rate*, *max_depth*, and *reg_lambda*.

Type
dict

`class ai_model.xgboost_model.`**`XGBoost`**`(`*problem_type='regression'*, *options=None*`)`
Bases: `BaseModel`

This module provides an implementation of the XGBoost model for regression. It extends the BaseModel class and ensures that hyperparameters are validated before model initialization.

A class to implement the XGBoost model for regression.

Parameters

- **`problem_type`** – Defines the type of problem being solved (only regression supported).
- **`options`** – Contains hyperparameters such as *n_estimators*, *learning_rate*, *min_split_loss*, and *max_depth*.

Raises

- **`ValueError`** – If *problem_type* is not “regression”.
- **`Exception`** – If an error occurs during model initialization.

`class ai_model.random_forest.`**`RandomForest`**`(`*problem_type='classification'*, *options=None*`)`
Bases: `BaseModel`

A class to implement the Random Forest model for both classification and regression.

9.1 Attributes:

problem_type
[str] Defines whether the model is for classification or regression.

options
[dict] Contains hyperparameters such as n_estimators, max_depth, min_samples_split, and min_samples_leaf.

CHAPTER

TEN

IMAGE PROCESSING

`class image_processing.dataloader.`**`DataLoadingAndPreprocessing`**(*image_size=(28, 28)*)

Bases: `object`

A class to handle data loading and preprocessing.

`image_size`

Size of images (default is (28, 28)).

Type
tuple

`data`

Loaded data as a Pandas DataFrame.

Type
DataFrame

`labels`

List of labels.

Type
list

`label_dict`

Dictionary mapping labels to integers.

Type
dict

`images`

List of processed images.

Type
list

`data_loader`(*dataset_name*, *is_zipped=True*)

Loads and preprocesses image dataset.

`get_label_dict`()

Returns label dictionary.

`encode_labels`()

Encodes labels into numeric values.

`create_labels`(*folder_name*, *is_zipped=True*)

Creates and encodes labels from directory structure.

`unzip_folder`(*current_path*, *folder_name*, *data_dir*)

Unzips dataset if necessary.

`normalize_dataset`()

Normalizes dataset to the range [0,1].

`split_dataset`(*test_size=0.2*, *random_state=42*)

Splits dataset into training and test sets.

`create_labels`(*data_dir*)

Creates and encodes labels from the dataset folder.

This method assumes that the dataset consists of subdirectories named after the class labels, each containing images of that class. It processes these subdirectories, encodes the labels numerically, and stores them in a DataFrame.

Parameters
`data_dir` (*str*) – The name of the folder containing the dataset.

Returns
None

`data_loader`(*dataObj*)

This method loads the data from the dataset folder (zipped or unzipped), creates labels, encodes them, loads the dataset and normalizes the dataset.

Parameters: dataObj: dict

A data object dictionary containing

dataset_name: str
The name of the folder where the images are stored

is_zipped: bool
If the dataset is zipped or not

`encode_labels`()

Encodes the labels into numerical values.

This method assigns a unique integer to each label and maps the dataset labels to their corresponding numerical values.

Returns
None

`get_label_dict`()

Retrieves the label dictionary.

Returns
Mapping of label names to integers.

Return type
dict

`normalize_dataset`()

Normalizes the dataset by converting images to numpy arrays and scaling pixel values.

This method ensures that pixel values are in the range [0,1] for better training efficiency in deep learning models.

Returns
None

`split_dataset`(*dataObj*)

Splits the dataset into training and testing sets.

This method partitions the preprocessed dataset into training and test sets, ensuring reproducibility with a fixed random state.

Parameters

`dataObj` (*dict*) – The data object dictionary consisting of: test_size (float): The proportion of the dataset to include in the test split. Defaults to 0.2. random_state (int): The seed used by the random number generator for reproducibility. Defaults to 42.

Returns

A tuple containing four numpy arrays:

- X_train (numpy.ndarray): Training images.
- y_train (numpy.ndarray): Training labels.
- X_test (numpy.ndarray): Test images.
- y_test (numpy.ndarray): Test labels.

Return type

tuple

`unzip_folder`(*current_path*, *zip_file*, *data_dir*)

Extracts a ZIP file if it is not already unzipped.

This method checks if the dataset directory exists. If not, it attempts to extract the ZIP file containing the dataset.

Parameters

- **`current_path`** (*str*) – The path where the script is executed.
- **`zip_file`** (*str*) – The name of the zipfile.
- **`data_dir`** (*str*) – The directory where the dataset should be extracted.

Returns

None

`class image_processing.evaluator.`**`Evaluation`**(*model*)

Bases: `object`

A class for evaluating a trained model and visualizing its performance using a confusion matrix.

`evaluate_model`(*dataObj*)

Evaluates the model on the test dataset.

10.1 Attributes:

X_test

[numpy.ndarray] The test dataset features.

y_test

[numpy.ndarray] The true labels for the test dataset.

10.2 Returns:

: - test_acc: Test accuracy. - test_loss: Test loss.

`get_confusion_matrix`(*dataObj*, *pred_tuple*)

Generates the confusion matrix.

10.3 Parameters:

pred_tuple
[tuple] A tuple containing the predicted labels and indicies for the test dataset.

10.4 Returns:

: dict: A dictionary containing six key value pairs:

- labels: Unique labels from the dataset
- values: The confusion matrix values in percentage
- xlabel: X-axis label to be used for visualization.
- ylabel: Y-axis label to be used for visualization.
- title: Graph title to be used for visualization.
- tick_marks: Tick marks to be used for visualization.

`class image_processing.nn.NeuralNetwork`

Bases: `object`

A class to create and manage a Convolutional Neural Network (CNN) model using TensorFlow and Keras.

`classmethod create_cnn_model`(*dataObj*)

Create a Convolutional Neural Network (CNN) model with configurable optimizer, loss function, and activation functions.
The model consists of: - Two convolutional layers with ReLU activation. - Two max-pooling layers. - A fully connected dense layer with 128 neurons and ReLU activation. - An output layer with 10 neurons and softmax activation for multi-class classification.

Parameters
dataObj (*dict*) – Data dictionary containing: - optimizer (str): Optimizer to compile the model (e.g., 'adam', 'RMSPROP' & 'adamax'). - activation_function (str): Activation function for hidden layers (e.g., 'relu', 'sigmoid').

Returns
A compiled CNN model.

Return type
model

`class image_processing.test.Testing`(*model*, *dataObj*)

Bases: `object`

A class for testing a trained model by making predictions and visualizing results.

`model`

The trained machine learning model.

Type
object

`X_test`

The test dataset features.

Type
numpy.ndarray

`y_test`

The test dataset labels.

Type
numpy.ndarray

`label_dict`

A dictionary mapping class indices to class labels.

Type
dict, optional

`set_label_dict`(*label_dict*)

Stores a reverse mapping of the label dictionary.

`make_predictions`()

Generates predictions using the trained model.

`plot_image`(*pred_tuple*, *index*)

Displays a sample test image along with its predicted and true labels.

`get_predicted_tuple`()

Converts model output to class labels and probabilities.

`get_predicted_tuple`()

Converts model predictions to a tuple of (index, predicted label, probability).

Returns
Each tuple contains (predicted class index, predicted class name, prediction probability).

Return type
list of tuples

`make_predictions`(*X_test_reshaped*)

Makes predictions using the trained model.

Returns
The predicted output from the model.

Return type
numpy.ndarray

`plot_image`(*pred_tuple*, *index*)

Visualizes a sample image and shows predictions.

Parameters

- **pred_tuple** (*list of tuples*) – A list of tuples containing (predicted index, predicted label, predicted probability).

- `index` (*int*) – The index of the image in the test dataset.

Raises
`ValueError` – If the label dictionary is not set.

`set_label_dict`(*label_dict*)

Stores the reverse mapping of a label dictionary.

Parameters
`label_dict` (*dict*) – A dictionary mapping class names to class indices.

`class image_processing.train.Training`(*model*)

Bases: `object`

A class for testing a trained model by making predictions and visualizing results.

10.5 Attributes:

model
[object] The trained machine learning model.

X_test
[numpy.ndarray] The test dataset features.

y_test
[numpy.ndarray] The test dataset labels.

10.6 Methods:

make_predictions(use_cnn=False):
Generates predictions using the trained model.

plot_image(index, use_cnn=False):
Displays a sample test image along with its predicted and true labels.

get_predicted_labels(y_predicted):
Converts model output to class labels.

`train_nn`(*dataObj*, *epochs=10*)

Trains the neural network model on the training data.

10.6.1 Parameters:

dataObj
[dict] Data object dictionary containing training and test datasets (X_train, y_train, X_test, y_test).

epochs
[int, optional] Number of training epochs (default is 10).

PYTHON MODULE INDEX

INDEX