Fraunhofer IPK Ticket-classification System

December 20, 2024

Introduction

This system leverages three years of historical data, comprising 20,000 tickets, to generate predictions for new tickets. It features a comprehensive Machine Learning pipeline that encompasses the entire lifecycle of model development and deployment. The pipeline includes components for data fetching, preprocessing, model training, and deployment. Additionally, it integrates an agent that performs actions based on model predictions. Also, monitoring scripts are included to oversee pipeline performance and trigger automated actions as necessary.

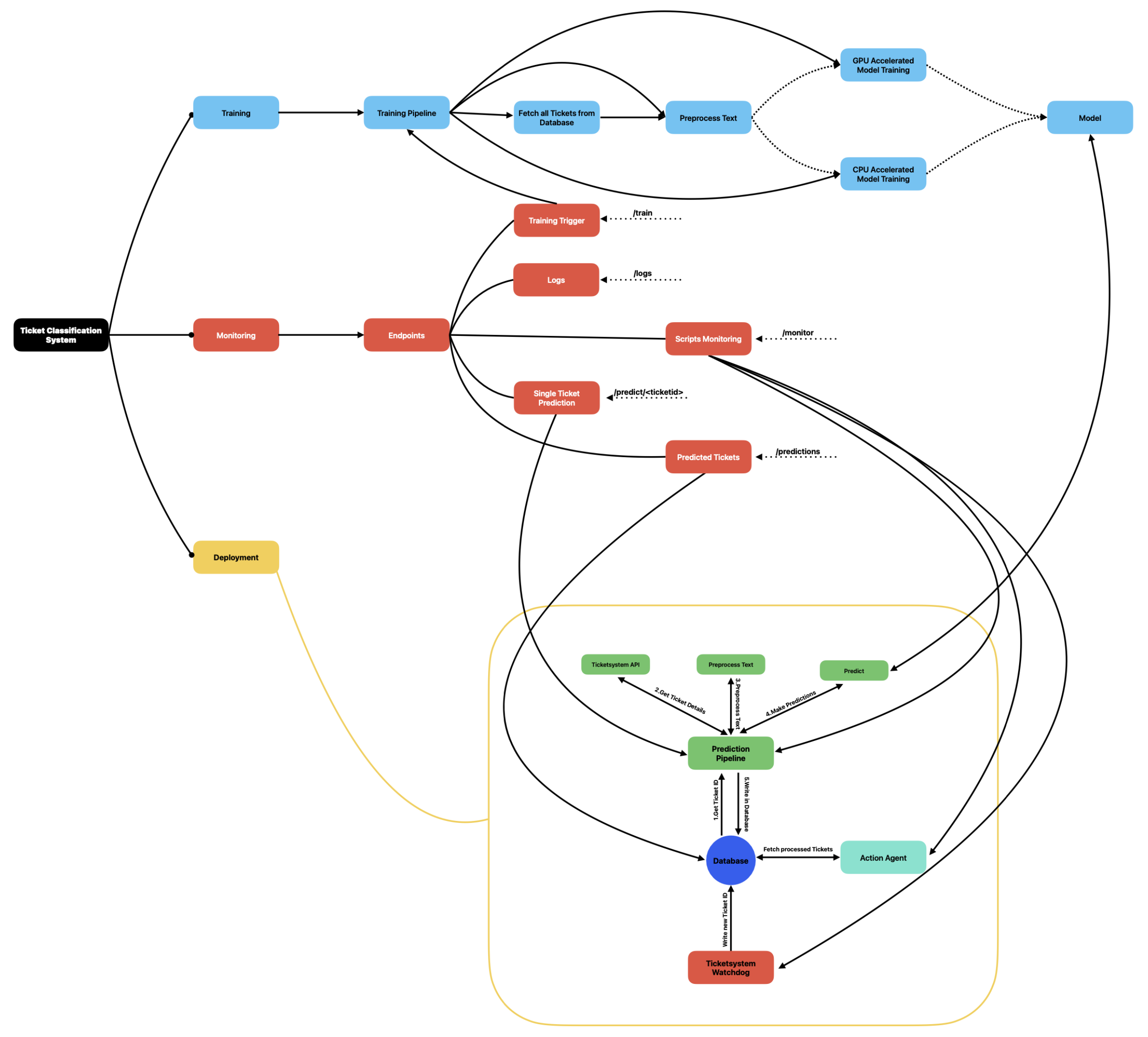


Figure 1: System Architecture

System Overview

The system is structured into three primary components, each further divided into detailed subcomponents.

Contents

Part I
Training

1 Training Pipeline

1.1 Data Fetching

1.2 Data Preprocessing

1.3 Model Training (CPU/GPU)

Part II
Model Deployment

2 Prediction Pipeline

2.1 Ticketsystem Messenger

2.2 Ticket Preprocessing

2.3 Ticket Prediction

3 Ticketsystem Watchdog

4 Action Agent

Part III
Monitoring

5 Mission Control

5.1 Training Cycle Trigger

5.2 Logs

5.3 Scripts Monitoring

5.4 Single Ticket Prediction

5.5 Database View

Part 1: Training

The training cycle begins with data retrieval from the REDACTED Ticket System database. The retrieved data undergoes preprocessing, during which features are extracted using a TF-IDF Vectorizer. Subsequently, Machine Learning algorithms such as Linear Regression, Support Vector Machine (SVM), Random Forest, and Gradient Boosting Machine are used to train the models either by utilizing CPU or GPU.

1: Training Pipeline

This script manages the entire training cycle. Users can opt to fetch data from the REDACTED Ticket System database and preprocess it, preprocess existing text without fetching new data, or skip data operations entirely. Additionally, the script facilitates triggering the training process, providing options to leverage CPU or GPU and choose the preferred algorithm for model training.

Code Documentation

Functions

1. `main`
    - **Description**: The main function coordinates the operations based on the command-line arguments passed when running the script.
    - **Command-Line Arguments**:
        - `sys.argv[1]`: Defines the data operation to perform:
            * `'df'` – Fetch and preprocess data.
            * `'pp'` – Only preprocess the data.
            * `'d'` – Do nothing (skip data operations).
        - `sys.argv[2]`: Defines the training type:
            * `'cpu'` – Perform training on CPU.
            * `'gpu'` – Perform training on GPU.
            * `'d'` – Do nothing (skip training).
        - `sys.argv[3]`: An additional argument passed to the training function, which represents the training algorithm to use:
            * `'lr'` – Run the Logistic Regression Algorithm.
            * `'svm'` – Run the Support Vector Machines Algorithm.
            * `'rf'` – Run the Random Forest Algorithm.
            * `'gbm'` – Run the Gradient Boosting Machine Algorithm.
            * `'train'` – Run all of the algorithms and choose the best one.
    - **Flow**:
        - If `sys.argv[1]` is `'df'`:
            * Calls `datafetch.main()` to fetch the data.
            * Followed by `preprocess.main()` to preprocess the data.
        - If `sys.argv[1]` is `'pp'`:
            * Skips data fetching and only preprocesses the data.
        - If `sys.argv[1]` is `'d'`:
            * Skips the data-related operations entirely.
        - If `sys.argv[2]` is `'cpu'`:
            * Invokes `cputraining.main(sys.argv[3])` to begin CPU-based training.
        - If `sys.argv[2]` is `'gpu'`:
            * Invokes `gputraining.main(sys.argv[3])` to begin GPU-based training.
        - If `sys.argv[2]` is `'d'`:
            * Skips the training process entirely.

1.1: Data Fetching

This script is responsible for fetching data from a Microsoft SQL Server database (REDACTED Ticket-system Database) and exporting the data into JSON files.

Code Documentation

DecimalEncoder Class

The `DecimalEncoder` class is a custom JSON encoder designed to handle the serialization of `Decimal` and `datetime` objects when exporting data to JSON:

- **Decimal**: Converts `Decimal` objects to `float`.
- **datetime**: Converts `datetime` objects to ISO format strings.

Functions

1. `fetch_and_export_data(cursor, query, output_file)`
   - **Description**: This function fetches data from the database using the provided cursor and SQL query, then exports the data to a JSON file.
   - **Arguments**:
     - `cursor` (`pyodbc.Cursor`): The cursor used to execute the SQL query.
     - `query` (`str`): The SQL query to fetch data.
     - `output_file` (`str`): The path to the output JSON file where data will be saved.
   - **Flow**:
     - Executes the query using the provided cursor.
     - Retrieves the result rows and columns.
     - Converts the rows into a list of dictionaries, with one dictionary per row.
     - Creates any necessary directories for saving the output file.
     - Exports the data to a JSON file, utilizing the `DecimalEncoder` class to handle special data types.
     - Logs a success message upon successful data export or an error message if any issue occurs.
2. `main()`
   - **Description**: This function connects to the SQL Server database and calls `fetch_and_export_data()` for each table to be exported.
   - **Flow**:
     - Constructs a connection string using environment variables to connect to the SQL Server database.
     - Establishes the connection and creates a cursor for database interaction.
     - Calls `fetch_and_export_data()` to fetch data from specific tables (`REDACTED`, `REDACTED`, and `REDACTED`) and export them to corresponding JSON files.
     - Logs success or error messages based on the connection and data-fetching process.

1.2: Data Preprocessing

This script processes raw ticket data, cleans and structures it. It uses various tools and techniques such as **spaCy** for natural language processing, **BeautifulSoup** for HTML tag removal, and **regular expressions** for pattern matching in ticket titles.

Code Documentation

Functions

1. `load_json_file(file_path: str) -> dict`
   - **Arguments**:
     - `file_path`: Path for the JSON file.
   - **Process**:
     - Loads and returns the content of a JSON file.
     - Logs errors if the file is not found or if there are issues decoding the JSON.
2. `process_resources(raw_resources_data: dict) -> dict`
   - **Arguments**:
     - `raw_resources_data`: The path for the raw resources table data in disk.
   - **Process**:
     - Processes raw resource data to create a mapping of resource IDs to their descriptions.
     - Constructs full resource descriptions by recursively traversing parent resources.
     - Logs errors for any issues during processing.
     - Updates resource descriptions for specific resource IDs (e.g., ServiceDesk resources).
3. `process_categories(raw_categories_data) -> dict`
   - **Arguments**:
     - `raw_categories_data`: The path for the raw categories table data in disk.
   - **Process**:
     - Processes raw category data to create a mapping of category IDs to their descriptions.
     - Constructs full category descriptions by recursively traversing parent categories.
     - Logs errors for any issues during processing.
     - Updates category descriptions for specific category IDs.
4. `process_ticket_title(title)`
   - **Arguments**:
     - `title`: The text in the ticket title
   - **Process**:
     - Removes specific patterns from ticket titles using regular expressions (e.g., `[warte]`, `[abholbereit]`).
     - Processes the cleaned title further by calling the process_text function.
5. `remove_html_tags(text)`
   - **Arguments**:
     - `text`: The text in the ticket body
   - **Process**:
     - Removes HTML tags from the provided text using BeautifulSoup's `lxml` parser.
     - Processes the cleaned text further by calling process_text function.
6. `normalize_text(token_text: str) -> str`
   - **Arguments**:
     - `token_text`: tokenized text
   - **Process**:
     - Normalizes German umlauts by replacing them with their respective letter combinations (e.g., ä $\rightarrow$ `ae`).

7. `process_text(text)`
   - **Arguments**:
     - `text`: text to be processed
   - **Process**:
     - Processes text by tokenizing it with spaCy's German language model.
     - Filters out punctuation, spaces, stop words, and very short tokens.
     - Replaces numerical tokens with a placeholder (`<NUM>`) and URL/email tokens with `<URL/EMAIL>`.
     - Lemmatizes tokens and converts them to lowercase.
     - Calls normalize_text for the normalization step
     - Returns the list of processed tokens.
8. `process_ticket(ticket, resources_dict, categories_dict) -> dict`
   - **Arguments**:
     - `ticket`: single ticket
     - `resources_dict`: Resources dictionary
     - `categories_dict`: Categories Dictionary
   - **Process**:
     - Processes a single ticket, extracting relevant information (Ticket ID, Title, Category, Resource, and Text).
     - The `Title` is processed with `process_ticket_title()`, and the `Text` is cleaned with `remove_html_tags()`.
     - The `Category` and `Resource` are looked up in their respective dictionaries.
9. `main()`
   - **Process**:
     - Coordinates the entire preprocessing pipeline.
     - Loads raw data (tickets, resources, and categories) from JSON files.
     - Processes the resource and category data to create dictionaries for quick lookup.
     - Processes the raw ticket data, cleaning the ticket title, description, and linking them to the appropriate resource and category.
     - Measures the time taken to preprocess the raw ticket data and logs the elapsed time.
     - Exports the processed ticket data to a new JSON file in the `Processed` directory.
     - Logs any errors during the export process.

1.3: Model Training

This script is used for training and evaluating different classification models on text data, specifically predicting categories and resources from a dataset of tickets. It utilizes various models such as Logistic Regression, SVM, Random Forest, and Gradient Boosting, and optimizes their hyperparameters using `RandomizedSearchCV`. The pipeline also integrates data preprocessing using TF-IDF vectorization and logging for monitoring.

Code Documentation

Functions

1. `load_json_file(file_path)`
   - **Arguments**:
     - `filepath`: JSON file to load
   - **Process**:
     - Loads a JSON file and returns its content as a dictionary.
     - Logs errors if the file is not found or if there are issues parsing the JSON.

2. `input_report(df, type)`
   - **Arguments**:
     - `df`: Pandas data frame object
     - `type`: type of the classification - resources or categories
   - **Process**:
     - Creates a bar plot of the distribution of categories or resources and saves it as a PNG file.
     - Used to visualize the count of target labels (e.g., `Category` or `Resource`).
3. `generate_report(model, classification_report, elapsed_time)`
   - **Arguments**:
     - `model`: the algorithm that was used to train the model
     - `classification_report`: the classification report generated by scikit-learn
     - `elapsed_time`: the time it took to train the model
   - **Process**:
     - Saves the classification report and the model's performance to a text file.
     - Logs the elapsed time for training.
4. `save_model(model, model_name, model_type)`
   - **Arguments**:
     - `model`: the trained model
     - `model_name`: the algorithm used to train the model
     - `model_type`: the type of classification the model was trained on - Categories or Resources
   - **Process**:
     - Saves the trained model to the `Models` directory using `joblib.dump`.
5. `save_preprocessors(vectorizer, label_encoder_category, label_encoder_resource)`
   - **Arguments**:
     - `vectorizer`: the TF-IDF vectorizer
     - `label_encoder_category`: The encoder to encode the labels for categories
     - `label_encoder_resource`: The encoder to encode the labels for resources
   - **Process**:
     - Saves the TF-IDF vectorizer and label encoders for categories and resources to disk.
6. `tfidf_vectorizer(train_set, test_set)`
   - **Arguments**:
     - `train_set`: the training set for vectorization
     - `test_set`: the test set for vectorization
   - **Process**:
     - Converts text data into numeric features using TF-IDF vectorization.
     - Custom stopwords are excluded from the text data to reduce noise.
7. **Model Functions** (`logistic_regression`, `svm`, `randomforest`, `gbm`)
   - Each of these functions handles training a specific model (Logistic Regression, SVM, Random Forest, or Gradient Boosting) using a randomized hyperparameter search (`RandomizedSearchCV`).
   - They evaluate the model using the `classification_report` and log the best hyperparameters, accuracy, and other metrics.
   - The model is saved after training using the `save_model` function.
8. `model_selection(X_train, X_test, y_train, y_test, label_encoder, model_type)`

- **Arguments**:
  - `X_train`: the training set data
  - `X_test`: the test set data
  - `y_train`: the training set labels
  - `y_test`: the test set labels
  - `label_encoder`: encoder for the labels
  - `model_type`: type of classification - categories/resources
- **Process**:
  - Trains all the models (Logistic Regression, SVM, Random Forest, Gradient Boosting) and selects the best-performing model based on accuracy.
  - It compares model performance on the test set and saves the best model.

9. `main(argument)`
   - **Command-Line Arguments**:
     - `sys.argv[1]`: the Argument passed to the training function, which represents the training algorithm to use:
       - `'lr'` – Run the Logistic Regression Algorithm.
       - `'svm'` – Run the Support Vector Machines Algorithm.
       - `'rf'` – Run the Random Forest Algorithm.
       - `'gbm'` – Run the Gradient Boosting Machine Algorithm.
       - `'train'` – Run all of the algorithms and choose the best one.
   - **Process**:
     - The entry point of the script.
     - Takes a command-line argument to decide which models or processes to run:
       - `'lr'`, `'svm'`, `'rf'`, `'gbm'`: Train a specific model.
       - `'train'`: Train all models and select the best one.
     - Loads the data, encodes labels, splits the dataset, applies TF-IDF, and trains and evaluates the selected models.

Detailed Flow

1. **Data Loading**
   - The JSON data is loaded from the `Data/Processed/tickets.json` file and converted to a pandas DataFrame.
2. **Feature Engineering**
   - Features are extracted from the `Title` and `Text` columns of the ticket dataset by concatenating the text fields.
   - TF-IDF vectorization is applied to convert the textual data into numerical features for the model.
3. **Model Training**
   - The data is split into training and test sets (80/20 split).
   - Each model (Logistic Regression, SVM, Random Forest, Gradient Boosting) is trained on the dataset using the appropriate function.
   - Hyperparameters are tuned using `RandomizedSearchCV` for each model.
4. **Model Evaluation**
   - The best models are evaluated on the test set.
   - Classification metrics (accuracy, precision, recall, F1-score) are logged.
   - The best-performing model is saved.

5. **Output**
    - Logs of the training process, including the best hyperparameters and evaluation metrics, are saved.
    - Model reports and training summaries are saved to disk.
    - Plots of category and resource distributions are generated and saved.

Part 2: Model Deployment

Model Deployment is comprised of several components that work together. The core component is the local SQLite database, which manages the ticket-saving, prediction, and action flow. New tickets are identified by the watchdog script, which then writes them to the database. The prediction pipeline script subsequently reads the ticket ID, retrieves the corresponding raw ticket information from the REDACTED Ticketsystem Web API, preprocesses the data, and calls the machine learning model to generate a prediction. This prediction is then written back into the database. Finally, the agent script retrieves the predictions from the database and uses the REDACTED Web API to perform actions based on the predictions.

2: Prediction Pipeline

This script is designed to monitor a SQLite database for unprocessed tickets. For each unprocessed ticket, it retrieves the ticket details, preprocesses the data, and uses a prediction model to generate a result.

Code Documentation

Functions

1. `process_ticket(ticket_id)`
   - **Description**: Processes a single ticket by fetching details, preprocessing the data, and generating a prediction using a machine learning model.
   - **Arguments**:
     - `ticket_id` (int): The unique identifier for the ticket to be processed.
2. `monitor_db(db_path, sleep_interval=10)`
   - **Description**: Monitors the SQLite database for unprocessed tickets and triggers the processing pipeline. It checks for tickets marked as unprocessed (`Processed = 0`), processes them, and updates the database with the predictions.
   - **Arguments**:
     - `db_path` (str): The full path to the SQLite database where the tickets are stored.
     - `sleep_interval` (int): The number of seconds to wait between checks for unprocessed tickets. Default is 10 seconds.
   - **Flow**:
     (a) Connects to the SQLite database.
     (b) Queries for unprocessed tickets.
     (c) Processes each ticket by calling `process_ticket()`.
     (d) Updates the ticket status and prediction in the database.

Main Flow

- **Database Path**: The script determines the path to the SQLite database relative to the script directory.
- **Threading**: The `monitor_db` function is run in a separate background thread to continuously monitor the database for new tickets.
- **Main Loop**: The script enters an infinite loop where it sleeps for 1 second. The monitoring thread runs concurrently, checking the database for unprocessed tickets and processing them when found.

2.1: Ticketsystem Messenger

This script is responsible for fetching ticket details from a REDACTED Ticketsystem using its API. It retrieves the title of a ticket and the text of a specific step in the ticket's workflow (identified by `actionID = 4`). The script also logs activities and errors during the process. The ticket data is then returned in a JSON format.

Functions

1. `make_api_request(url, headers)`
   - **Description**: A helper function that makes an HTTP `GET` request to the provided URL and handles potential errors by logging them.
   - **Arguments**:
     - `url` (str): The full URL for the API endpoint.
     - `headers` (dict): The headers to be included in the request (e.g., Authorization headers).
2. `get_ticket_details(ticket)`
   - **Description**: Retrieves the title and specific step text (identified by `actionID = 4`) of a ticket from the REDACTED Ticketsystem API.
   - **Arguments**:
     - `ticket` (int): The ticket ID to fetch details for.
   - **Flow**:
     (a) Makes an API request to fetch the ticket's title.
     (b) Makes another request to fetch the steps associated with the ticket.
     (c) Searches for the step with `actionID = 4`, which signifies the step containing the text we need.
     (d) If the step is found, fetches the step's text.
     (e) Returns the ticket title and step text as a JSON string.

2.2: Ticket Preprocessing

This script preprocesses ticket data, including the ticket's title and text. The preprocessing involves cleaning HTML tags, normalizing special characters (e.g., umlauts), tokenizing the text, and handling specific types of information such as numbers, URLs, and emails. The processed data is returned in JSON format. The script utilizes **spaCy** for tokenization and linguistic processing, and **BeautifulSoup** for cleaning HTML tags. The preprocessing approach aligns with the bulk processing of Ticketsystem Data during the Training phase.

Functions

1. `remove_html_tags(text)`
   - **Description**: Removes any HTML tags from the provided text using **BeautifulSoup**. Processes the cleaned text further by calling the `process_text()` function. Uses `lxml` as the parser for **BeautifulSoup**.
   - **Arguments**:
     - `text` (str): The input text that may contain HTML tags.
2. `normalize_text(token_text)`
   - **Description**: Normalizes German umlauts (e.g., `ä` to `ae`, `ö` to `oe`) in the provided text.
   - **Arguments**:
     - `token_text` (str): A token (word) that may contain umlauts.
3. `process_text(text)`
   - **Description**: Processes the input text by:
     (a) Tokenizing the text using **spaCy**.
     (b) Removing punctuation, spaces, stop words, and tokens shorter than 3 characters.
     (c) Replacing numbers, URLs, and emails with placeholders.
     (d) Normalizing the lemma of each token and converting it to lowercase.
   - **Arguments**:

    – `text` (str): The input text to process.

4. `process_ticket_title(title)`
    - **Description**: Processes the title of a ticket by removing specific patterns such as `[warte...]` or `[abholbereit...]` using regular expressions. Cleans the title and processes it using `process_text()`.
    - **Arguments**:
        – `title` (str): The ticket's title.

5. `preprocess_ticket(json_data)`
    - **Description**: The main entry point for processing a ticket. This function:
        (a) Loads the input JSON data.
        (b) Processes the ticket's title using `process_ticket_title()`.
        (c) Removes HTML tags from the ticket's text using `remove_html_tags()`.
        (d) Logs the received and processed data.
    - **Arguments**:
        – `json_data` (str): A JSON string containing ticket data with fields such as `Title` and `Text`.

2.3: Ticket Prediction

This script is designed for predicting the category and resource of a ticket using pre-trained machine learning models. The prediction process includes:

1. Loading pre-trained models for text vectorization, category prediction, and resource prediction.
2. Transforming ticket data using a TF-IDF vectorizer.
3. Making predictions on the ticket's category and resource.
4. Returning the predicted category and resource.

The script uses **joblib** for loading models, **pandas** for data handling, and **logging** for tracking the process.

Functions

1. `load_model(file_path)`
    - **Description**: Loads a machine learning model from a specified file path using **joblib**. Logs success and failure during the loading process.
    - **Arguments**:
        – `file_path` (str): The path to the model file to be loaded.

2. `predict(ticket_data)`
    - **Description**: Predicts a ticket's category and resource based on its title and text. The process involves:
        (a) Loading pre-trained models for text vectorization, category prediction, and resource prediction.
        (b) Preprocessing the ticket data.
        (c) Transforming the ticket text using the TF-IDF vectorizer.
        (d) Making predictions using the category and resource models.
        (e) Inversely transforming predictions back to the original labels.
        (f) Returning the predicted category and resource.
    - **Arguments**:
        – `ticket_data` (str): The ticket data in JSON format, including fields like `Title` and `Text`.

3: Watchdog

This script monitors a ticket system by periodically checking for new tickets, logging them into a SQLite database, and refreshing the page as needed. It uses **Selenium** to automate browser interactions, allowing login to the ticket system, navigation through the ticket list, and storing ticket information in the database.

Functions

1. `init_db()`
   - **Description**: Initializes the SQLite database and creates the `tickets` table if it does not already exist.
2. `save_ticket(cursor, conn, ticket_id)`
   - **Description**: Saves a new ticket into the database. If the ticket ID already exists, it logs a warning.
   - **Arguments**:
     - `cursor`: The database cursor used for executing SQL queries.
     - `conn`: The SQLite connection object.
     - `ticket_id` (int): The unique identifier for the ticket.
3. `login(driver, wait)`
   - **Description**: Logs into the ticket system using Selenium WebDriver. The process involves navigating to the ticket system URL and entering the username and password.
   - **Arguments**:
     - `driver`: The Selenium WebDriver instance.
     - `wait`: A `WebDriverWait` instance for waiting until elements are available.
4. `process_tickets(driver, wait, cursor, conn)`
   - **Description**: Continuously checks for new tickets on the ticket system and saves them to the database. The page is refreshed every hour, and the script waits for ticket elements to load before processing.
   - **Arguments**:
     - `driver`: The Selenium WebDriver instance.
     - `wait`: A `WebDriverWait` instance for waiting until elements are available.
     - `cursor`: The database cursor used for inserting tickets.
     - `conn`: The SQLite connection used to commit transactions.
5. `main()`
   - **Process**:
     (a) **Initialize Database**:
       - The script initializes the SQLite database and ensures the `tickets` table exists.

     (b) **Start WebDriver**:
       - A Firefox WebDriver instance is started using GeckoDriver, and Selenium waits for specific elements (such as login and ticket elements) to load.

     (c) **Login**:
       - Logs into the ticket system using provided credentials (username and password).

     (d) **Navigate to Ticket List**:
       - After logging in, the script navigates to the ticket list page and begins monitoring for new tickets.

     (e) **Monitor Tickets**:
       - The script continuously checks the ticket list for new tickets. If a new ticket is found (not previously logged), it is saved to the database.

(f) **Page Refresh**:
   - The page is refreshed every hour to ensure the script stays up-to-date with the ticket system.

(g) **Error Handling**:
   - Errors such as timeouts or WebDriver exceptions are logged, and the script retries after a short delay.

(h) **Shutdown and Restart**:
   - If a critical error occurs, the WebDriver is shut down, and the script restarts from the beginning.

4: Action Agent

Part 3: Monitoring

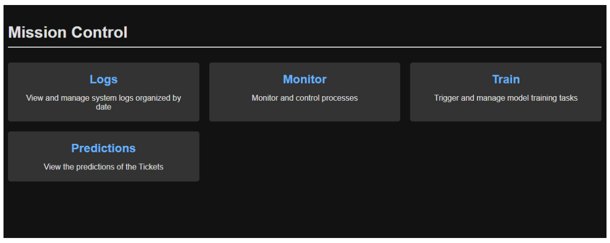
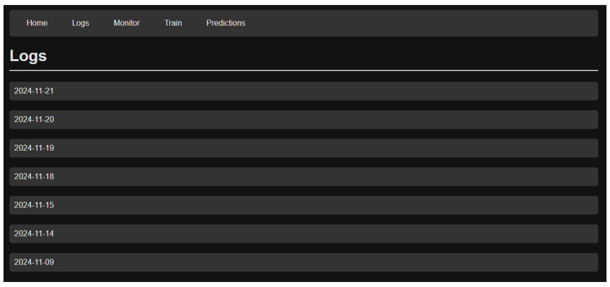
The Monitoring component features a Flask-powered web interface that enables remote management. It allows users to manage and monitor various scripts, view logs generated by these scripts, trigger training cycles, and access predicted ticket data. Additionally, it offers an endpoint for predicting individual tickets.

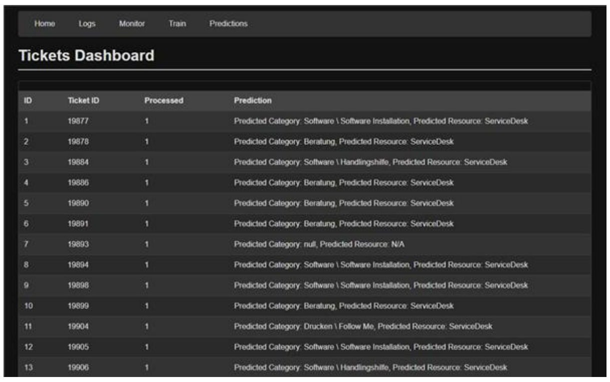
Database Integration

- Uses SQLite database for storing ticket information.
- Database connection is managed through `get_db_connection()`.
- Database path is configurable via the environment variable `DB_PATH`.

Routes and Endpoints

Main Pages

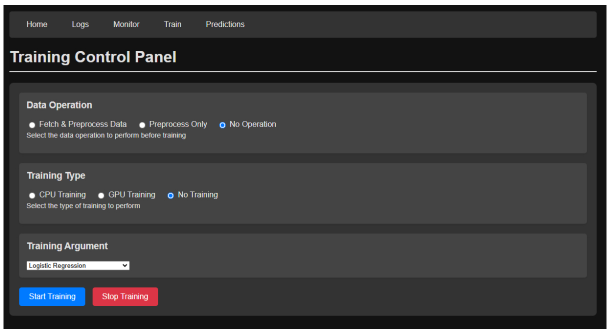
1. **Home Page (`/`)**
   - Renders the main interface through `index.html`.
   - 
     Mission Control
     Logs
     View and manage system logs organized by date
     Monitor
     Monitor and control processes
     Train
     Trigger and manage model training tasks
     Predictions
     View the predictions of the Tickets
2. **Logs Page (`/logs`)**
   - Displays all available log files organized by date.
   - Supports viewing both text logs and PNG files.
   - 

3. **Predictions Page (`/predictions`)**
   - Interface for viewing prediction results.
   - 

Tickets Dashboard

ID	Ticket ID	Processed	Prediction
1	19877	1	Predicted Category: Software \ Software Installation, Predicted Resource: ServiceDesk
2	19878	1	Predicted Category: Beratung, Predicted Resource: ServiceDesk
3	19884	1	Predicted Category: Software \ Handlingshilfe, Predicted Resource: ServiceDesk
4	19886	1	Predicted Category: Beratung, Predicted Resource: ServiceDesk
5	19890	1	Predicted Category: Beratung, Predicted Resource: ServiceDesk
6	19891	1	Predicted Category: Beratung, Predicted Resource: ServiceDesk
7	19893	1	Predicted Category: null, Predicted Resource: N/A
8	19894	1	Predicted Category: Software \ Software Installation, Predicted Resource: ServiceDesk
9	19898	1	Predicted Category: Software \ Software Installation, Predicted Resource: ServiceDesk
10	19899	1	Predicted Category: Beratung, Predicted Resource: ServiceDesk
11	19904	1	Predicted Category: Drucken \ Follow Me, Predicted Resource: ServiceDesk
12	19905	1	Predicted Category: Software \ Software Installation, Predicted Resource: ServiceDesk
13	19906	1	Predicted Category: Software \ Handlingshilfe, Predicted Resource: ServiceDesk

4. **Training Page (`/train`)**

- Interface for initiating and monitoring model training.
- 

Training Control Panel

Data Operation

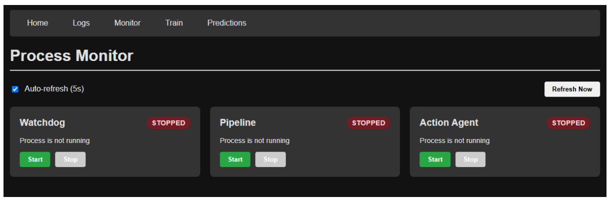
Fetch & Preprocess Data   Preprocess Only   [✓ No Operation]
Select the data operation to perform before training

Training Type

CPU Training   GPU Training   [✓ No Training]
Select the type of training to perform

Training Argument

Logistic Regression

5. **Monitor Page (`/monitor`)**
   - Displays status of running processes.
   - 

Process Monitor

✓ Auto-refresh (5s)

Watchdog STOPPED
Process is not running

Pipeline STOPPED
Process is not running

Action Agent STOPPED
Process is not running

API Endpoints

1. **Prediction Endpoint (`/predict/<ticket>`)**
   - Method: `GET`
   - Processes ticket predictions through the pipeline module.
   - Returns JSON response with prediction results.
   - Handles errors with appropriate HTTP status codes.
2. **Data Endpoint (`/data`)**
   - Method: `GET`
   - Retrieves all records from the tickets database.
   - Returns JSON response with ticket data.
3. **Training Control (`/api/train`)**
   - Method: `POST`
   - Accepts JSON payload with training parameters:
     - `data_operation`
     - `training_type`
     - `training_args`
   - Runs training in a separate thread.
   - Returns immediate response with training status.
4. **Process Control (`/api/control/<script_name>/<action>`)**
   - Method: `POST`
   - Controls background processes (start/stop).
   - Supports scripts: `watchdog`, `pipeline`, `action-agent`.
   - Returns JSON response with operation status.
5. **Status Endpoint (`/api/status`)**
   - Method: `GET`
   - Returns status of all monitored processes.
   - Includes PID and running time for active processes.

Process Management

The Monitoring component manages three main background processes:

1. Watchdog
2. Pipeline
3. Action Agent

Each process can be:

- **Started** via `start_script()`.
- **Stopped** via `stop_script()`.
- **Monitored** via `get_script_status()`.

Function Reference

Process Management Functions

- `get_script_status(script_name)`
  - **Description**: Checks if a script is running and returns its status.
  - **Parameters**:
    - `script_name` (str): Name of the script to check.
  - **Returns**: A dictionary with status, PID, and running time.
- `start_script(script_name)`
  - **Description**: Starts a specified script if not already running.
  - **Parameters**:
    - `script_name` (str): Name of the script to start.
  - **Returns**: A tuple `(success_boolean, message)`.
- `stop_script(script_name)`
  - **Description**: Stops a running script.
  - **Parameters**:
    - `script_name` (str): Name of the script to stop.
  - **Returns**: A tuple `(success_boolean, message)`.

Training Functions

- `run_training_script(data_op, training_type, training_args)`
  - **Description**: Executes the training pipeline with specified parameters.
  - **Parameters**:
    - `data_op` (str): Data operation argument.
    - `training_type` (str): Training type argument.
    - `training_args` (str): Additional training arguments.
  - **Returns**: A dictionary with success status and script output.

Log Management Functions

- `get_log_dates()`
  - **Description**: Retrieves available log dates.
  - **Returns**: A list of dates in reverse chronological order.
- `get_files_for_date(date)`
  - **Description**: Gets all log files for a specific date.
  - **Parameters**:
    - `date` (str): Date in `YYYY-MM-DD` format.
  - **Returns**: A list of filenames.

Error Handling

- Database connection errors return a 500 status code.
- Invalid predictions return a 400 status code.
- File access errors return appropriate error messages.
- Process management errors are logged and reported.

Part 4: Training Report

will follow