# Explanation: Enhanced Multi-Provider Data Collector

## What This Collector Does

This script is designed to automatically gather conversational data from a variety of AI chat model datasets, especially focusing on both OpenAI and non-OpenAI sources. Its main purpose is to create a diverse and high-quality dataset for research, benchmarking, or building AI detection tools1.

## How It Works

* **Dataset Selection**: The collector targets several major datasets, including both public and gated (restricted access) sources. Examples include LMSYS-Chat-1M, Chatbot Arena, OpenOrca, Anthropic HH-RLHF, and OpenAssistant1.
* **Data Extraction**: For each dataset, it reads conversations and extracts pairs of prompts (user questions) and responses (AI answers). The extraction logic adapts to each dataset’s unique structure1.
* **Model & Provider Tagging**: It tries to identify which AI model (e.g., GPT-4, Claude, Gemini, Llama, Mistral) generated each response and assigns a provider label (like OpenAI, Anthropic, Google, Meta, Mistral)1.
* **Cleaning & Quality Checks**: The script cleans the text (removing formatting artifacts), checks for minimum length and diversity, and calculates simple quality scores to filter out poor-quality responses1.
* **Metadata & Reporting**: Each data sample is stored with detailed metadata: prompt, response, model, provider, quality scores, dataset source, and more. After collection, the script summarizes how many samples were collected per model, provider, and dataset, and reports average quality statistics1.

## Key Features

* **Multi-Source**: Collects from both open and gated datasets, supporting authentication for restricted sources1.
* **Flexible Model Detection**: Uses pattern matching to standardize model names across different datasets1.
* **Quality Filtering**: Applies basic checks to ensure only useful data is kept (e.g., not too short, not repetitive)1.
* **Automatic Saving**: Outputs the collected data and summary report in both JSON and CSV formats for easy analysis1.

## Typical Usage

1. **Prepare Environment**: Install dependencies and, if needed, log in to HuggingFace for gated datasets.
2. **Run the Script**: Launch the script; it will process each dataset, extract and clean data, and save the results.
3. **Review Output**: Check the generated JSON/CSV files and the summary statistics to understand the data collected1.

## Why Use This Collector?

* **Diversity**: Ensures your dataset isn’t biased toward a single provider or model.
* **Quality**: Filters out low-quality or irrelevant samples automatically.
* **Transparency**: Provides clear reports on what was collected and from where.

This collector is ideal for anyone needing a broad, high-quality sample of AI-generated conversations, especially when building tools that need to distinguish between outputs from different AI models1.

Trainer:  
**Explanation: Multi-Class AI Model Classifier Training Script**

**What This Script Does**

This script is designed to train a text classifier that can distinguish between responses generated by various AI models (such as GPT-4, Claude, Llama, Mistral, and others). It uses transformer-based architectures (DistilBERT, RoBERTa, or DeBERTa) and applies techniques to handle class imbalance, ensure robust evaluation, and save all necessary training artifacts for future inference or analysis.

**Key Steps and Features**

**1. Data Loading and Validation**

* Loads training data from a CSV file named training\_data.csv.
* Checks that the dataset contains the required columns: cleaned\_response (the AI-generated text) and final\_class (the model label).
* Ensures all model classes in the data match a predefined set, preventing accidental mislabeling or inclusion of unexpected classes.

**2. Label Encoding and Splitting**

* Converts model class names to integer labels using LabelEncoder.
* Splits the data into training, validation, and test sets, using stratification to preserve class distribution across splits for fair evaluation.

**3. Class Imbalance Handling**

* Computes class weights so that underrepresented classes are not neglected during training.
* Uses a custom WeightedTrainer that applies these weights to the loss function, making the model more robust to imbalanced datasets.

**4. Tokenization and Sequence Length**

* Automatically determines the 95th percentile of tokenized text lengths in the training data, setting this as the maximum sequence length for efficient training without excessive truncation.
* Tokenizes all text data using the selected transformer model’s tokenizer.

**5. Model Selection and Training**

* Supports three transformer architectures: DistilBERT, RoBERTa, and DeBERTa.
* Initializes the model for multi-class classification with the correct number of labels.
* Configures training parameters, such as batch size, learning rate (with sensible defaults), number of epochs, and evaluation strategy.
* Uses mixed-precision (FP16) training if a compatible GPU is available for speed and memory efficiency.

**6. Metrics and Evaluation**

* Computes accuracy, macro F1, and weighted F1 scores during evaluation for a comprehensive view of model performance across all classes.
* Handles GPU out-of-memory errors by automatically reducing batch size and retrying, making the script robust to hardware limitations.

**7. Saving Outputs**

* Saves the trained model and tokenizer to disk for later use.
* Exports a detailed classification report (per-class precision, recall, F1, and support) as a CSV file.
* Saves the confusion matrix and the mapping of class indices to class names for transparency and reproducibility.

**Typical Usage**

1. **Prepare Data:** Place your labeled and cleaned data in training\_data.csv.
2. **Run the Script:**

bash

python your\_script.py --model roberta --batch\_size 8 --learning\_rate 2e-5

* + Choose the model (distilbert, roberta, or deberta), batch size, and optionally a learning rate.

1. **Review Results:**
   * Find your trained model and evaluation artifacts in the output directory (e.g., roberta\_final/).
   * Use the saved CSV and NPY files for further analysis or to inform downstream inference pipelines.

**Why Use This Script?**

* **Handles Class Imbalance:** Ensures fair training even if some models are underrepresented in your data.
* **Automates Evaluation:** Provides detailed metrics and artifacts to help you understand model performance.
* **Robust and Flexible:** Automatically adapts to your hardware and dataset, making it suitable for a range of research or production tasks.

This script is ideal for anyone building a classifier to distinguish between outputs from different AI models, especially in research, benchmarking, or AI detection contexts.

Inference:

**Explanation: ModelPredictor Inference Utility**

**Overview**

The ModelPredictor class is a Python utility designed for detecting the most likely source AI model behind a given text response using a transformer-based classifier. It loads a pre-trained model and tokenizer, validates inputs, and provides single and batch predictions, as well as top-k results for any input text.

**Key Components**

**1. Initialization**

* **Model & Tokenizer Loading:**  
  Loads a transformer model and tokenizer from a specified directory (default: 'roberta\_final'). It automatically detects whether to use GPU (CUDA) or CPU for inference.
* **Class Names:**  
  Loads model class names from either a .npy or .csv file in the model directory. These names are used as prediction labels.
* **Max Sequence Length:**  
  Sets the maximum number of tokens for model input (default: 512), ensuring compatibility with the model's training setup.

**2. Input Validation**

* **Type & Content Check:**  
  Ensures input is a non-empty string and not excessively long (max 10,000 characters). Raises errors for invalid inputs to prevent runtime issues.

**3. Prediction Methods**

* **Single Prediction (predict):**
  + Validates and tokenizes the input text.
  + Runs the model in inference mode (no gradients).
  + Applies softmax to output logits to get class probabilities.
  + Returns:
    - The most likely model label.
    - The confidence score for the prediction.
    - A dictionary of all model scores (probabilities).
* **Batch Prediction (predict\_batch):**
  + Accepts a list of texts.
  + Validates and tokenizes all inputs.
  + Processes the batch in a single model call for efficiency.
  + Returns a list of prediction dictionaries (same structure as single prediction).
* **Top-K Predictions (top\_k\_predictions):**
  + Returns the top-k most likely model labels and their confidence scores for a single text.
  + Useful for understanding the model’s uncertainty or for multi-class scenarios.

**4. Utility Method**

* **Get Class Names (get\_class\_names):**  
  Returns the list of model class names that the classifier can predict.

**Typical Workflow**

1. **Instantiate the Predictor:**

python

predictor = ModelPredictor(model\_dir="roberta\_final")

1. **Single Prediction:**

python

result = predictor.predict("Sample AI-generated text.")

1. **Batch Prediction:**

python

results = predictor.predict\_batch(["Text 1", "Text 2", ...])

1. **Top-K Results:**

python

top\_k = predictor.top\_k\_predictions("Sample text", k=3)

1. **Class Names:**

python

labels = predictor.get\_class\_names()

**Why Use This Utility?**

* **Fast & Flexible:** Handles both single and batch inference efficiently.
* **Robust Validation:** Prevents invalid or problematic inputs from causing errors.
* **Comprehensive Output:** Provides full probability distributions, not just the top class.
* **Easy Integration:** Designed for plug-and-play use in larger AI detection or analytics pipelines.

**Notes**

* Ensure the model directory contains the correct .npy or .csv file for class names.
* The utility assumes the model was trained on the same class label set as provided in the directory.
* For best performance, use a compatible GPU if available.

This explanation should help you understand how the ModelPredictor class works and how to use it effectively in your AI model detection projects.

Provider\_config:

**Explanation: provider\_config.py – AI Provider Configuration Utility**

**Overview**

The provider\_config.py module defines a standardized configuration dictionary for interacting with various AI model APIs, such as OpenAI's GPT series and Anthropic's Claude. It provides a unified way to format requests, set headers, extract responses, and handle both callable and non-callable (offline) providers in your application[1](https://api.openai.com/v1/chat/completions).

**Key Components**

**1. AI\_PROVIDER\_CONFIG Dictionary**

* **Purpose:**  
  Stores configuration for each supported AI provider/model as a nested dictionary.
* **Contents for Each Provider:**
  + url: The API endpoint for the provider (or None if not available).
  + headers: A function that returns the required HTTP headers, typically for API authentication.
  + format\_request: A function that creates the correct request payload given a prompt.
  + extract\_response: A function that extracts the model's text response from the raw API output.
  + class\_name and family: Standardized names for downstream use (e.g., for classification or analytics).

**Example Entry: OpenAI GPT-3.5 Turbo**

python

"openai-gpt-3.5-turbo": {

"url": "https://api.openai.com/v1/chat/completions",

"headers": **lambda** key: {"Authorization": f"Bearer {key}"},

"format\_request": **lambda** prompt: {

"model": "gpt-3.5-turbo",

"messages": [{"role": "user", "content": prompt}],

"max\_tokens": 256,

"temperature": 0.8,

},

"extract\_response": **lambda** data: data["choices"][0]["message"]["content"],

"class\_name": "gpt-3.5-turbo",

"family": "gpt-3.5-family"

}

* This entry enables automated, consistent requests and parsing for GPT-3.5 Turbo[1](https://api.openai.com/v1/chat/completions).

**Placeholder Entries**

* For providers like Meta Llama and Mistral (which lack public APIs as of June 2025), the url is set to None, and other fields are placeholders. This signals that these models are only available for offline or simulated detection, not live API calls[1](https://api.openai.com/v1/chat/completions).

**2. get\_provider\_config Function**

* **Purpose:**  
  Retrieves the configuration for a given provider name.
* **Behavior:**
  + If the provider is not found or not callable (i.e., url is None), it raises a clear error message.
  + Otherwise, it returns the configuration dictionary for use in making API calls or formatting requests[1](https://api.openai.com/v1/chat/completions).

**Usage Example**

python

cfg = get\_provider\_config("openai-gpt-4")

headers = cfg["headers"]("YOUR\_API\_KEY")

payload = cfg["format\_request"]("Hello, AI!")

*# Use 'url', 'headers', and 'payload' to make the API request*

* This approach abstracts away provider-specific quirks, making it easy to swap between models or add new ones[1](https://api.openai.com/v1/chat/completions).

**Why Use This Pattern?**

* **Unified Interface:**  
  Simplifies code that interacts with multiple AI APIs by standardizing request formatting and response extraction[1](https://api.openai.com/v1/chat/completions).
* **Extensibility:**  
  New providers or models can be added by simply extending the configuration dictionary[1](https://api.openai.com/v1/chat/completions).
* **Error Handling:**  
  Prevents accidental attempts to call unsupported APIs, guiding users to use offline detection when needed[1](https://api.openai.com/v1/chat/completions).

**Notes**

* Ensure that API keys are handled securely and never hardcoded.
* For providers with no API, offline detection or simulated responses should be used for consistency.
* This module is intended to be imported and used by higher-level scripts or services that orchestrate prompt submission and response handling[1](https://api.openai.com/v1/chat/completions).

This configuration utility is essential for maintaining scalable, maintainable, and provider-agnostic AI application codebases[1](https://api.openai.com/v1/chat/completions).

Prompt generator:  
**Explanation: paste.txt – Curated Prompt Generator for AI Model Testing**

**Overview**

The PromptGenerator class in paste.txt is a Python utility for generating high-quality, diverse prompts to test and differentiate AI language models. It provides a structured way to produce prompts across various domains (like technical, creative, reasoning, business, and educational) and complexity levels (basic, intermediate, advanced), supporting robust benchmarking and evaluation of AI model capabilities1.

**Key Components**

**1. Enumerations**

* **ComplexityLevel:** Defines three levels of prompt complexity: basic, intermediate, and advanced1.
* **TopicDomain:** Defines five domains for prompts: technical, creative, reasoning, business, and educational1.

**2. Prompt Storage**

* Prompts are organized in a nested dictionary structure, mapping each domain and complexity level to a curated list of example prompts1.
* Each prompt is designed to challenge different aspects of AI models, from simple factual tasks to advanced reasoning and creative generation1.

**Core Methods**

**generate\_prompts(domain, complexity, count)**

* Returns a list of prompts from a specified domain and complexity level.
* If more prompts are requested than are available, it repeats prompts to meet the requested count1.

**generate\_mixed\_prompts(count, domain=None)**

* Generates prompts across all complexity levels within a specified domain (or a random domain if none is provided).
* Distributes the requested number of prompts as evenly as possible across complexity levels1.

**generate\_diverse\_prompts(count)**

* Produces prompts randomly sampled from all domains and complexity levels, ensuring maximum diversity in the prompt set1.

**Utility Methods**

* get\_available\_domains(): Returns a list of all available domains.
* get\_available\_complexities(): Returns a list of all available complexity levels1.

**Example Usage**

python

pg = PromptGenerator()

*# Get 3 intermediate technical prompts*

prompts = pg.generate\_prompts(TopicDomain.TECHNICAL, ComplexityLevel.INTERMEDIATE, 3)

*# Get 6 prompts across all complexities in the 'creative' domain*

mixed = pg.generate\_mixed\_prompts(6, TopicDomain.CREATIVE)

*# Get 10 prompts from random domains and complexities*

diverse = pg.generate\_diverse\_prompts(10)

**Why Use This Utility?**

* **Benchmarking:** Enables fair, systematic evaluation of AI models on a wide range of tasks and difficulty levels1.
* **Diversity:** Ensures prompt sets are not biased toward a single domain or complexity, supporting more robust model comparisons1.
* **Reproducibility:** Provides a consistent, transparent way to generate test prompts for experiments or competitions1.

**Notes**

* The prompt lists can be easily extended or customized for new domains or additional complexity levels1.
* This class is ideal for researchers, developers, or anyone needing structured, high-quality prompts for AI model assessment1.

Main/FastApi:

**Explanation: AI Model Detection Orchestrator API**

**Overview**

This FastAPI application serves as an orchestrator for AI model detection, prompt generation, and online/offline inference. It integrates a prompt generator, a model prediction utility, and provider configuration to support both local and API-based detection workflows. The API is designed for robustness, extensibility, and ease of use in research or production settings.

**Key Components**

**1. Application Startup**

* **Lifespan Event:**  
  On startup, the app loads a PromptGenerator for producing diverse prompts and a ModelPredictor for offline model inference. This ensures both are ready for all subsequent requests.
* **Static Files:**  
  The /frontend route serves static files, supporting any web-based UI.

**2. Data Models**

* **PromptGenRequest:** For generating prompts, specifying domain, complexity, and count.
* **OfflineDetectRequest:** For single offline detection, accepting a text input.
* **BatchOfflineDetectRequest:** For batch offline detection, accepting a list of texts.
* **OnlineDetectRequest:** For online detection via provider APIs, including provider name, API key, and prompt settings.
* **TopKRequest:** For retrieving the top-k most likely models for a given text.

**3. Endpoints**

**/generate\_prompts (POST)**

* **Purpose:** Generate prompts based on domain, complexity, or both.
* **Logic:**
  + If both domain and complexity are specified, generates prompts accordingly.
  + If only one is specified, fills in the other with a random or mixed choice.
  + If neither is specified, generates a diverse set.
* **Returns:** A list of prompts.

**/predict (POST)**

* **Purpose:** Predict the most likely model for a single text input (offline).
* **Logic:**
  + Uses the loaded ModelPredictor to classify the input text.
* **Returns:** The most likely model, confidence, and all model scores.

**/predict\_batch (POST)**

* **Purpose:** Predict models for a batch of texts (offline).
* **Logic:**
  + Processes the batch efficiently using the model predictor.
* **Returns:** A list of prediction results.

**/predict\_topk (POST)**

* **Purpose:** Retrieve the top-k most likely models for a single text.
* **Logic:**
  + Uses the predictor’s top-k method to return the highest scoring models and their confidences.
* **Returns:** List of top-k models and scores.

**/classes (GET)**

* **Purpose:** List all model class names supported by the current predictor.
* **Returns:** List of class names.

**/detect\_online (POST)**

* **Purpose:**
  + Generate a prompt.
  + Send it to a specified AI provider via their API.
  + Receive the AI’s response.
  + Run offline detection on the response to predict which model produced it.
* **Logic:**
  + Handles prompt generation based on user preferences.
  + Looks up provider API configuration and formats the request.
  + Makes an async HTTP call to the provider API.
  + Extracts the AI’s response and metadata.
  + Runs offline detection on the returned text.
* **Returns:** The prompt, AI response, provider class/family, and detection result.

**/health (GET)**

* **Purpose:** Simple health check endpoint.
* **Returns:** Status message.

**Error Handling**

* All endpoints use structured exception handling.
* Errors are logged and returned as HTTP errors with appropriate status codes and messages.
* Provider API errors, prompt generation errors, and detection failures are all handled gracefully.

**Usage Highlights**

* **Flexible Prompting:** Easily generate prompts for diverse benchmarking or testing scenarios.
* **Offline & Online Detection:** Supports both local model inference and real-time API-based detection.
* **Batch and Top-K Support:** Efficiently handle multiple texts and retrieve detailed prediction insights.
* **Extensible Provider Support:** Easily add new providers or models by updating the provider configuration module.

**Typical Workflow**

1. **Generate prompts** using /generate\_prompts for benchmarking or testing.
2. **Detect model origins** for user-provided or generated texts using /predict or /predict\_batch.
3. **Benchmark providers** by sending prompts to live APIs via /detect\_online and analyzing both the AI response and offline detection result.
4. **Explore model classes** supported by the current detector with /classes.
5. **Monitor service health** with /health.

This orchestrator API is ideal for research, competitive benchmarking, or production environments where robust, explainable AI model detection and prompt management are required.