Project Report: Property Extraction from WhatsApp Chats using Mistral-7B

1. Project Overview

The goal of this project was to extract structured property information from unstructured WhatsApp chat messages using Large Language Models (LLMs), specifically Mistral-7B. The pipeline processes chat logs, identifies property-related messages, and extracts relevant fields (e.g., property type, location, price, contact) into structured JSON using a combination of prompt engineering, batch processing, and post-processing.

2. Approaches Attempted

2.1. Model Loading and Inference

- Primary Model: Mistral-7B-Instruct-v0.3
- Frameworks Used:
  - HuggingFace Transformers (for model and tokenizer)
  - vLLM (for fast, batched inference; attempted)
  - LangChain (for prompt chaining and orchestration)
  - Outlines (for structured output, attempted)
- Batch Processing: Implemented to maximize GPU utilization and throughput.
- Sliding Window: Used for long messages to avoid truncation and maximize context.

2.2. Extraction Pipeline

- Message Extraction: Regex-based parsing to split WhatsApp chats into individual messages.
- Prompt Engineering: Custom prompts for residential, commercial, and land properties.
- Batch Inference: Messages are processed in batches for efficiency.
- Post-processing: Regex and JSON parsing to extract structured data from model outputs.
- Metrics Logging: System and extraction metrics are logged for monitoring.

3. Challenges and Failed Approaches

3.1. vLLM Integration

- Attempted: vLLM was considered for high-throughput, low-latency inference.
- Issue: The available GPU (Tesla T4, Compute Capability 7.5) is not supported by vLLM's V1 engine, which requires Compute Capability 8.0+.
- Result: vLLM fell back to V0 engine, but still encountered out-of-memory (OOM) errors and instability.
- Logs:
- Status: vLLM was ultimately disabled for stability.

3.2. Quantization

- Attempted: 4-bit and 8-bit quantization to reduce memory usage.
- Issue: Quantization was not always compatible with the model or device, and sometimes led to degraded performance or loading errors.
- Status: Defaulted to "none" (full precision) for reliability.

3.3. Flash Attention and BetterTransformer

- Attempted: Enabled for faster inference.
- Issue: Not supported or unstable on T4 GPUs.
- Status: Disabled for safety.

3.4. Outlines for Structured Output

- Attempted: Used for direct JSON output from LLM.
- Issue: Not always available or compatible; fallback to regex-based extraction was needed.

3.5. Device and Memory Constraints

- Device: Tesla T4 (15.83 GB VRAM)
- Issue: Model loading and inference are memory-intensive. Even with batch size tuning and quantization, OOM errors occurred with larger batches or longer contexts.
- Workarounds:
  - Reduced batch size (fixed at 8 for safety)
  - Sliding window for long messages
  - Aggressive garbage collection and CUDA cache clearing before model loading

4. Current Limitations

4.1. Hardware

- GPU: Tesla T4 (Compute Capability 7.5, 15.83 GB VRAM)
- CPU/RAM: Sufficient for preprocessing, but GPU is the bottleneck.
- Disk: No major issues, but fast SSD recommended for large models.

4.2. Software

- vLLM: Not fully usable due to GPU compatibility.
- Batch Size: Limited by VRAM; cannot fully utilize model throughput.
- Model Size: Mistral-7B is at the upper limit for T4; larger models are not feasible.
- Inference Speed: Acceptable for small batches, but slow for large datasets.

4.3. Extraction Quality

- LLM Output: Sometimes unstructured or partially structured; requires robust post-processing.
- Regex/JSON Extraction: Not 100% reliable; may miss or misparse some fields.

5. Results

- Extraction Output: Structured JSON files with property details (see output.json , batching_output.json).
- Logs: Detailed logs of system metrics, model loading, and extraction steps (output_logs.txt , batching_log.txt).
- Distribution: Model weights and tensors are loaded entirely on GPU 0 (see distribution.txt).

6. Recommendations for Future Upscaling

6.1. Hardware Upgrades

- GPU: Upgrade to A100, H100, or L4 (Compute Capability 8.0+) for:
  - Full vLLM support (V1 engine, FlashAttention-2)
  - Larger batch sizes and faster inference
  - Ability to run larger models (e.g., 13B, 70B)
- Multi-GPU: Consider multi-GPU setup for parallel processing and model sharding.
- RAM: 64GB+ recommended for preprocessing large files.

6.2. Software Improvements

- vLLM: Re-enable and tune for high-throughput batch inference.
- Quantization: Use 4-bit/8-bit quantization with compatible hardware for memory savings.
- FlashAttention: Enable for further speedup on supported GPUs.
- Distributed Inference: Use Ray or similar frameworks for distributed batch processing.

6.3. Extraction Pipeline

- Prompt Engineering: Further refine prompts for higher accuracy.
- Schema Enforcement: Use Outlines or Pydantic for stricter output validation.
- Error Handling: Improve fallback mechanisms for failed extractions.
- Monitoring: Enhance system and extraction metrics logging for better observability.

6.4. Data and Use Case Expansion

- Larger Datasets: With better hardware, process larger chat logs and more messages per batch.
- Model Fine-tuning: Fine-tune Mistral or similar models on domain-specific data for improved extraction accuracy.
- Multi-lingual Support: Expand to handle chats in multiple languages.

7. Requirements for Upscaling

Hardware

- NVIDIA GPU: A100, H100, L4, or better (Compute Capability 8.0+)
- VRAM: 40GB+ for large batch sizes and models
- RAM: 64GB+ system memory
- Storage: Fast SSD (NVMe) for model weights

Software

- CUDA: Version compatible with target GPU
- PyTorch: Latest stable version
- vLLM: Latest version (for V1 engine and FlashAttention-2)
- Transformers: Latest version
- LangChain, Outlines: For orchestration and structured output
- Ray: For distributed processing (optional)

Other

- Internet Access: For downloading models and dependencies

- Python 3.9+

8. Conclusion

Despite hardware limitations, the project successfully demonstrates property extraction from WhatsApp chats using Mistral-7B. The pipeline is robust and modular, ready to be upscaled with better hardware and software support. With the recommended upgrades, extraction speed and quality can be significantly improved, enabling processing of much larger datasets and more complex extraction tasks.

1. langchain_extractor.py Documentation

Overview

This module provides the LangchainPropertyExtractor class, which is designed to extract structured property information (residential, commercial, land) from unstructured chat messages using large language models (LLMs) via the HuggingFace Transformers and LangChain frameworks. It supports batch processing, asynchronous file reading, and robust error handling, and is optimized for use on T4 GPUs.

Main Components Imports

- Transformers & Torch : For model loading and inference.
- LangChain : For prompt templating and LLM chaining.
- Asyncio, aiofiles : For asynchronous file and batch processing.
- orjson : For fast JSON serialization/deserialization.
- Logging, tqdm, os, gc, re : For logging, progress, system operations, memory management, and regex.
- Custom Models & Utils : Imports property models and utility functions from local modules. Class: LangchainPropertyExtractor Purpose Extracts property data from chat messages using a language model pipeline, with support for batch processing, prompt engineering, and system resource monitoring. Constructor (**init**)
- Parameters :
    - model_path : HuggingFace model identifier or path.
    - batch_size : Number of messages to process in a batch.
    - sliding_window_size : For handling long texts.
    - cache_size , quantization , device_map , offload_folder : Model loading and optimization options.
    - use_flash_attn , use_bettertransformer : Performance flags.
    - max_new_tokens , temperature : Generation parameters.
- Initialization Steps :
    - Loads tokenizer and model.
    - Sets up a HuggingFace pipeline for text generation.
    - Wraps the pipeline in a LangChain LLM for prompt-based extraction.
    - Prepares property extraction chains for each property type. Key Methods
- _start_monitoring() : Initializes system resource monitoring (not threaded by default).
- _create_property_chains() : Sets up LangChain prompt templates and chains for property extraction, including few-shot examples.
- _extract_json_from_text(text, category) : Extracts and parses JSON blocks from LLM output.
- _normalize_property_data(property_data, category) : Ensures property data is a list of dicts and assigns property category if missing.
- _read_file_async(filepath) : Asynchronously reads a file in chunks.
- _extract_messages_async(chunks) : Asynchronously parses chat messages from text chunks using regex.
- _preprocess_item_async(item) : Preprocesses a message (ID assignment, text normalization, category detection).
- _preprocess_text(text) : Cleans and normalizes message text.
- _determine_property_category(text) : Heuristically determines property category (residential, commercial, land).
- _extract_property_with_vllm(text, category) : (If vLLM available) Extracts property data using vLLM.
- _extract_property_with_langchain(text, category) : Extracts property data using LangChain LLM chain.
- _process_batch_async(batch) : Processes a batch of messages using the pipeline, normalizes results, and handles errors.
- process_file_async(filepath, output_path, limit=None) : Orchestrates the full pipeline: reads file, extracts messages, processes in batches, saves results, and logs metrics. Error Handling
- Extensive try/except blocks with logging for model loading, file reading, batch processing, and LLM inference.
- Fallbacks to default/empty property objects if extraction fails. Performance & Monitoring
- Tracks metrics such as total messages, successful/failed extractions, LLM call count, and processing times.
- Periodically clears CUDA cache and performs garbage collection to manage GPU memory.

Usage Example

```
extractor = LangchainPropertyExtractor(model_path="mistralai/Mistral-7B-Instruct-v0.3")
await extractor.process_file_async("input.txt", "output.json")
```

```
### Design Notes
- The class is optimized for T4 GPUs and disables certain features for compatibility.
- Uses batch processing for efficiency.
- Prompts are engineered with few-shot examples to improve extraction accuracy.
- Asynchronous design allows for scalable file and batch processing.


# Documentation for langchain_main.py
## Overview
This script is the main entry point for running property extraction from chat files using a large language model (LLM) pipeline buil

## Main Components
### 1. Imports and Setup
- Standard Libraries : Handles argument parsing, async execution, logging, warnings, system info, and resource management.
- Torch : For GPU/CPU device checks and memory management.
- Custom Utilities : Imports functions for suppressing CUDA warnings and logging system metrics.
- TensorFlow Logging Suppression : Attempts to silence TensorFlow logs if present.
- LangchainPropertyExtractor : Imports the main extraction class from langchain_extractor.py .
### 2. Logging and Warning Suppression
- Sets up logging to both console and file.
- Suppresses various Python and CUDA/TensorFlow warnings for cleaner output.
### 3. Timeout Handling
- Defines a TimeoutException and a Windows-compatible time_limit context manager for model loading timeouts (though not actively use
### 4. Argument Parsing
- Uses argparse to define a rich set of command-line arguments, including:
  - Input/output file paths
  - Model selection and fallback options
  - Batch size, sliding window, quantization, device mapping
  - Performance options (Flash Attention, BetterTransformer, max tokens, temperature)
  - Timeout and offload folder
  - Option to disable vLLM
### 5. Main Async Function Steps:
1. Parse Arguments : Reads all command-line options.
2. Print GPU Info : If CUDA is available, prints device details.
3. Validate Input File : Checks that the input file exists.
4. Output Path Generation : If not provided, generates a timestamped output path.
5. Offload Folder Creation : Ensures the offload folder exists.
6. Model Loading :
   - Tries to load the main model, and falls back to alternative or smaller models if specified.
   - Logs system metrics before and after model loading.
   - Handles model loading errors and timeouts.
7. File Processing :
   - Calls process_file_async on the extractor to process the input file and save results.
   - If extraction fails, attempts to use a simpler fallback extractor.
   - Prints summary statistics and logs final system metrics.
### 6. Script Entry Point
- Runs the async main() function using asyncio.run .
- Handles uncaught exceptions and exits with appropriate status code.
## Key Functions and Classes
### TimeoutException
Custom exception for handling timeouts during model loading.

### time_limit(seconds)
A context manager for enforcing timeouts on code blocks (Windows-specific implementation).

### main()
The main async function that orchestrates argument parsing, model loading, file processing, and logging.

## Usage
Run the script from the command line with various options. Example:
```

Key arguments:

- --input (-i): Path to the input chat file (required)
- --output (-o): Path to save the extracted results (optional)
- --model (-m): Model identifier or path (default: Mistral-7B-Instruct-v0.3)
- --batch-size (-b): Number of messages to process per batch
- --limit (-l): Limit the number of messages to process
- --quantize (-q): Quantization level (4bit , 8bit , none)
- --device (-d): Device mapping (e.g., auto , cuda:0)
- --timeout (-t): Timeout for model loading (seconds)
- --smaller-model (-sm): Use a smaller model if main fails
- --alternative-model : Alternative model to try if main fails
- --offload-folder (-of): Folder for offloading model weights
- --use-flash-attn : Use Flash Attention for faster inference
- --use-bettertransformer : Use BetterTransformer for optimized inference
- --max-new-tokens : Maximum number of new tokens to generate
- --temperature : Sampling temperature

Design Notes

- Robustness : The script is designed to handle model loading failures gracefully, with fallbacks and detailed logging.
- Performance : Supports batch processing, GPU memory management, and system metrics logging.
- Extensibility : New models and extraction strategies can be added with minimal changes.
- Windows Compatibility : Special care is taken for Windows-specific timeout and stderr handling.

Example Workflow

1. User runs the script with desired arguments.
2. The script loads the specified model (with fallbacks if needed).
3. The input chat file is processed in batches, extracting property information.
4. Results are saved to the output file, and performance metrics are logged.

Documentation for models.py

Overview

This module defines the data models used for representing property information in a structured way. It uses Pydantic for data valida

Classes

Area

Represents the area of a property.

- Fields:
  - value (Optional[float]): The numeric value of the area.
  - unit (Optional[str]): The unit of measurement (e.g., "sqft", "sqmt", "acres").

Dimensions

Represents the dimensions of a property (for land).

- Fields:
  - length (Optional[float]): Length of the property.
  - width (Optional[float]): Width of the property.
  - unit (Optional[str]): Unit of measurement.

Price

Represents price-related information.

- Fields:
  - rent (Optional[float]): Rent amount.
  - deposit (Optional[float]): Deposit amount.
  - sale_price (Optional[float]): Sale price.
  - price_per_unit (Optional[float]): Price per unit area.
  - currency (Optional[str]): Currency code (default: "INR").

Amenities

Represents a list of amenities.

- Fields:
  - items (Optional[List[str]]): List of amenity names.

BaseProperty

The base class for all property types.

- Fields:

  - property_category (Optional[str]): Category of the property (overridden in subclasses).
  - property_type (str): Type of property (e.g., "apartment", "shop").
  - intent (str): Intent (e.g., "sale", "rent").
  - area (Optional[Area]): Area information.
  - location (Optional[str]): Location description.
  - price (Optional[Price]): Price information.
- Validators:

  - Ensures property_category is set, defaulting to "unknown" if not provided.
ResidentialProperty
Represents a residential property.

- Fields:

  - Inherits all from BaseProperty .
  - property_category : Always "residential" .
  - configuration (Optional[str]): Configuration (e.g., "2BHK").
  - bathrooms , floor , total_floors , facing , view , furnishing_status (Optional[str]): Various residential attributes.
  - amenities (Optional[Amenities]): Amenities list.
- Validators:

  - Ensures property_category is always "residential" .
CommercialProperty
Represents a commercial property.

- Fields:

  - Inherits all from BaseProperty .
  - property_category : Always "commercial" .
  - floor , furnishing_status (Optional[str]): Commercial-specific attributes.
  - amenities (Optional[Amenities]): Amenities list.
- Validators:

  - Ensures property_category is always "commercial" .
LandProperty
Represents a land property.

- Fields:

  - Inherits all from BaseProperty .
  - property_category : Always "land" .
  - dimensions (Optional[Dimensions]): Land dimensions.
- Validators:

  - Ensures property_category is always "land" .
Usage Example
Documentation for utils.py
Overview
This module provides utility functions for system resource monitoring, logging, and suppressing warnings (especially CUDA and Tensor

Functions
suppress_cuda_warnings()
Suppresses CUDA, TensorFlow, and other common warnings to keep logs clean.

- How it works:

  - Sets environment variables to reduce verbosity.
  - On Windows, attempts to redirect stderr at the OS level to NUL .
  - If that fails, falls back to redirecting sys.stderr in Python.
  - Returns a function to restore the original stderr .
- Usage:
get_system_metrics()
Collects and returns current system resource usage metrics.

- Returns:

  - Dictionary with:
    - CPU usage percent
    - Memory usage percent, available, and used (in GB)
    - Disk usage percent
    - Timestamp
    - If CUDA is available: GPU count, name, allocated/reserved/total memory per GPU
- Usage:
log_performance_metrics(metrics)
Logs performance metrics to both a file and the console.

- How it works:

  - Appends metrics as JSON lines to a daily file in the metrics/ directory.
  - Prints a summary to the console, including system and extraction metrics if available.
- Usage:
Design Notes
- The module is cross-platform but includes special handling for Windows.
- Intended to be used at the start and end of major processing steps for diagnostics and performance tracking.
- Helps in debugging and monitoring resource usage during large model inference.

Full Workflow of langchain_extractor.py and Its Interactions
1. Purpose and Role
langchain_extractor.py defines the LangchainPropertyExtractor class, which is the core engine for extracting structured property inf

2. Initialization and Model Loading
- Class Instantiation : When a LangchainPropertyExtractor object is created, it:
  - Loads a tokenizer and LLM model (using HuggingFace Transformers).
  - Sets up a text-generation pipeline.
  - Wraps the pipeline in a LangChain LLM (HuggingFacePipeline).
  - Prepares prompt templates and LangChain chains for property extraction.
  - Initializes metrics for performance and extraction tracking.
Interaction:

- Uses classes and functions from:
  - models.py for property data structures.
  - utils.py for system metrics and logging.
3. Prompt Engineering and Chains
- Prompt Templates : The extractor uses a carefully crafted prompt (with few-shot examples) to instruct the LLM on how to extract pr
- Chains : Sets up three LangChain LLMChain objects (for residential, commercial, and land), but all use the same prompt since messa
4. File Processing Workflow
a. Reading the File
- Asynchronous File Reading : Uses _read_file_async() to read the input file in chunks asynchronously, which is memory-efficient for
b. Message Extraction
- Message Parsing : _extract_messages_async() parses the raw text into individual chat messages using regex patterns (e.g., WhatsApp
- Yields : Each message is yielded as a dictionary with fields like id , timestamp , sender , text , and raw .
c. Preprocessing
- Text Cleaning : _preprocess_item_async() and _preprocess_text() clean the message text (remove URLs, emojis, normalize case/whites
- Category Detection : _determine_property_category() heuristically assigns a category (residential, commercial, land) based on keyw
d. Batching
- Batch Preparation : Messages are grouped into batches (default size 2 for T4 GPU safety).
- Batch Processing : _process_batch_async() processes each batch:
  - Preprocesses each message.
  - Prepares prompts for the LLM.
  - Runs the batch through the pipeline (using HuggingFace’s pipeline).
  - Parses and normalizes the LLM output into structured property data.
e. Property Extraction
- LLM Inference : For each batch, the pipeline generates outputs for each prompt.
- JSON Extraction : _extract_json_from_text() extracts the JSON array from the LLM’s output using regex.
- Normalization : _normalize_property_data() ensures the output is a list of property dicts and assigns the correct category.
Fallbacks : If extraction fails, default property objects (from models.py) are used to ensure output consistency.

5. Saving Results and Logging
- Results Aggregation : All batch results are collected and sorted by message ID.
- Performance Metrics : Success rate, average processing time, and other metrics are computed.
- Saving Output : Results are saved as a JSON file using orjson for speed.

- Saving Output : Results are saved as a JSON file using orjson for speed.
- Logging : System and extraction metrics are logged using log_performance_metrics() from utils.py .
6. Resource Management
- GPU/CPU Monitoring : Periodically clears CUDA cache and runs garbage collection to prevent memory leaks.
- System Metrics : Logs system resource usage before and after major steps.
7. Error Handling
- Robust Try/Except : Handles errors at every stage (file reading, model inference, batch processing).
- Fallback Extraction : If the main extraction fails, can fall back to a simpler extractor (if implemented).
8. Interaction with Other Files
a. models.py
- Provides the data models (ResidentialProperty , CommercialProperty , LandProperty) used to structure and validate extracted prop
- Used for fallback/default objects and for ensuring output consistency.
b. utils.py
- Supplies utility functions:
  - get_system_metrics() : For monitoring CPU, RAM, and GPU usage.
  - log_performance_metrics() : For logging metrics to file and console.
- Used throughout for diagnostics and performance tracking.
c. langchain_main.py
- Acts as the main script that:
  - Parses command-line arguments.
  - Instantiates LangchainPropertyExtractor .
  - Calls process_file_async() to run the full extraction workflow.
  - Handles model loading, fallback logic, and final reporting.
9. Summary of Workflow
1. Initialization : Load model, tokenizer, and set up chains.
2. File Reading : Asynchronously read and parse chat messages.
3. Preprocessing : Clean and categorize each message.
4. Batching : Group messages and prepare prompts.
5. LLM Inference : Run prompts through the LLM pipeline.
6. Extraction : Parse and normalize the LLM’s output into structured property data.
7. Aggregation : Collect, sort, and save results.
8. Logging : Log performance and system metrics.
9. Error Handling : Ensure robustness and fallback strategies.
10. Typical Usage Flow
- User runs langchain_main.py with input/output file paths and model options.
- langchain_main.py creates a LangchainPropertyExtractor and calls its process_file_async() method.
- The extractor processes the file as described above, interacting with models.py and utils.py as needed.
- Results are saved and metrics are logged.

Future Scope: Arbitrary Fields and Database Integration
- Challenge: As observed, the LLM can extract arbitrary fields not defined in your schema (e.g., features , negotiable , minimum_are
- Future Scope:

  - Schema Evolution: Consider using flexible database models (e.g., JSON fields in PostgreSQL, MongoDB’s document model) to accommo
  - Dynamic Schema Mapping: Implement a mapping/normalization layer that:
    - Maps arbitrary fields to known schema fields where possible.
    - Stores unknown fields in a generic “extra” or “metadata” column for future analysis.
  - Schema Governance: Develop a process to periodically review and update your schema based on new fields observed in production da
  - User Feedback Loop: Allow users or admins to flag useful new fields for formal inclusion in the schema.
Other Approaches and Methods Not Implemented (Due to Limitations)
1. Strict Schema Enforcement at Extraction Time

   - The current pipeline allows the LLM to output any field.
   - An alternative would be to strictly constrain the LLM’s output to only schema fields (via prompt engineering or structured outp
2. Post-Extraction Validation and Cleaning

   - Implement a post-processing step that:
     - Drops unknown fields.
     - Warns/logs about unexpected fields.
     - Attempts to intelligently map or merge similar fields (e.g., sale → sale_price).
3. Interactive Correction/Annotation

   - Build a UI or tool for manual review and correction of extracted fields before database insertion.
4. Advanced Structured Generation

   - Use tools like Outlines or OpenAI’s function calling to enforce output structure more strictly.
5. Automated Schema Migration

   - Implement scripts to automatically update the database schema as new fields are discovered.
Current Limitations
- LLM Output Variability: The LLM may output fields not present in the schema, or use inconsistent naming/nesting, making downstream
- Schema Drift: As new fields appear, the schema in `models.py` can quickly become outdated.
- Database Rigidity: Relational databases require a fixed schema, so arbitrary fields can’t be easily stored without a flexible desi
- Prompt Limitations: Even with prompt engineering, LLMs may hallucinate or ignore instructions, especially on ambiguous or noisy in
- Performance and Scalability: Current batch size and resource management are tuned for a specific hardware setup (e.g., T4 GPU). Sc
- Error Handling: Fallbacks are basic (default objects on failure), and there’s limited reporting on what fields failed or why.
How to Upscale and Improve in the Future
1. Flexible Data Storage:

   - Use NoSQL/document databases (e.g., MongoDB) or JSONB columns in PostgreSQL to store dynamic fields.
   - Store both normalized (schema-mapped) and raw extracted data for future reference.
2. Automated Schema Evolution:

   - Build tools to analyze incoming data and suggest schema updates.
   - Use versioned schemas and migration scripts.
3. Advanced Output Validation:

   - Integrate Pydantic or Marshmallow validation post-extraction.
   - Log and review all unknown or unmapped fields.
4. Better Prompt Engineering and Structured Output:

   - Use tools like Outlines, OpenAI function calling, or LangChain’s output parsers to enforce output structure.
   - Provide more examples and stricter instructions in prompts.
5. Monitoring and Analytics:

   - Track field frequency and new field emergence over time.
   - Alert when new fields appear frequently.
6. User/Admin Feedback:

   - Allow users to annotate or approve new fields for inclusion in the schema.
7. Scalability:

   - Move to distributed processing (e.g., with Ray, Dask, or cloud-based batch jobs).
   - Optimize batch sizes and resource allocation dynamically.
8. Error Reporting and Logging:

   - Improve error logs to include which fields failed and why.
   - Build dashboards for monitoring extraction quality and schema drift.