Portfolio: Named Entity Recognition

Olusanmi Hundogan

A report submitted as part of the requirements for the toptal assessment

August 2025

Implemented for OHCM B.V.

Abstract

This report presents a named entity recognition (NER) system designed to extract **Persons**, **Organizations**, and **Locations** from a dataset of news articles. The task required building a performant and reproducible NLP solution that outperforms out-of-the-box models, deploys as a REST API, and scales under concurrent usage.

The workflow began with exploratory data analysis to understand entity distributions and text structure. Preprocessing included standard NLP techniques such as sentence segmentation, tokenization, and custom annotation handling for multi-word entities. Multiple baselines were tested, including pretrained transformer models (e.g., BERT-based token classifiers) and rule-based heuristics.

The final solution employed a fine-tuned transformer model on a custom-labeled dataset, trained to optimize exact-match entity extraction, especially for multi-token names and locations. Organizations were evaluated with dual strategies: full-string matches and token-wise comparisons. Experiment tracking ensured reproducibility and allowed comparative evaluations.

A RESTful API was developed to accept CSV files as input, process the text column, and return extracted entities in the required CSV format. The service was containerized using Docker, enabling easy deployment. To validate performance under load, the system was tested with `locust`, demonstrating stability under 20 concurrent users with acceptable latency.

Overall, this report outlines a robust pipeline for developing, deploying, and evaluating a custom NER system under real-world constraints, showcasing end-to-end applied machine learning capabilities.

Contents

List of Tables

List of Figures

List of Algorithms

Listings

Chapter 1

Introduction

Named Entity Recognition (NER) is a foundational task in natural language processing (NLP), concerned with identifying and classifying entities such as persons, organizations, and locations in unstructured text. This technical report outlines the development of a custom NER solution designed to process news articles and extract these entity types accurately under constrained evaluation criteria. The project was completed as part of a time-bound applied machine learning assessment, with requirements that span both model performance and deployment readiness.

1.1 Background

Many off-the-shelf NLP models offer generic entity recognition capabilities. However, such models often fail to meet specific evaluation conditions — such as exact multi-word matches or fine-grained disambiguation of organizational names. Furthermore, real-world applications demand solutions that are not only accurate, but also reproducible, deployable, and performant under load.

This project was designed to demonstrate competency in end-to-end ML system development: from initial data exploration and preprocessing, through model design and training, to deployment as a REST API and evaluation under concurrent usage scenarios. The dataset used in this task consists of news articles with annotated entities across the three categories of interest.

1.2 About this Report

This technical report documents the approach, decisions, and results of the NER system development. It covers the complete project lifecycle — including data analysis, baseline testing, model experimentation, deployment via Docker, and performance validation using Locust.

The deliverables included:

- An ML pipeline that improves upon baseline NER performance
- A reproducible training setup with tracked experiments
- A REST API that takes a CSV as input and returns predictions in a specific format
- A Dockerized deployment for seamless execution
- Load testing results demonstrating support for 20+ concurrent users

1.3 Chapter List

Chapter 1 Explains the task that is solved with this solution.

Chapter 2 Describes and discusses dataset

Chapter 3 Every step to modify the dataset into a more usable dataset

Chapter 4 General software design as uml diagram.

Chapter 5 A listing of all the models implemented and tested.

Chapter 6 Summarizes model performance, benchmarking results, and system behaviour under concurrent load.

Chapter 7 A rough overview on how the system was depployed and how scalable it is.

Chapter 8 Conclusion. Highlights key findings and suggests directions for future improvement.

1.4 Conclusion

This introduction set the stage for the technical report by describing the problem, summarizing the goals and deliverables, and outlining the structure of the document. The following chapters provide detailed insights into each stage of the project.

Chapter 2

Data-Exploration

2.1 Entity Format Analysis

A closer look at the entity label columns revealed the following:

- **Persons and Organizations:** These fields contain a semicolon-separated list of `entity,id` pairs. For example:

  ```
  Joe Biden,135; Karen Jan Pierre,1533; Mohammed Ben Salman,55
  ```

  Each entry represents a full entity name and an associated numerical ID. The origin and purpose of these IDs remain unknown, and some entities appear multiple times with different IDs.

- **Locations:** Unlike the other two columns, this field uses a **comma-separated** format for listing entity names:

  ```
  White House, Yemeni, United States, Yemen, Saudi Arabia
  ```

  There are no associated IDs for location entities.

To better understand the structure and frequency of the labeled entities, all unique entities were extracted and saved to:

```
misc/entities_dict.json
```

This allowed inspection of the vocabulary and confirmed that the numeric IDs are **not unique** and provide no reliable information for downstream processing.

2.2 Conclusion

The dataset is a structured collection of news article snippets with annotated named entities for persons, organizations, and locations. Textual patterns and references (e.g., frequent mentions of CNN and political topics) strongly suggest that the content was downloaded or scraped from a news website.

The entity fields are inconsistently formatted: `persons` and `organizations` include non-unique numeric IDs, while `locations` are comma-separated with no IDs. After analysis, it was concluded that the numeric IDs do not carry consistent meaning and will be removed. Furthermore, to ensure consistency across all entity types, each field will be normalized into a semicolon-separated list of clean entity names.

This exploration phase highlights the importance of harmonizing and sanitizing the dataset before modeling. The next step is to implement the data preparation pipeline based on these insights.

Chapter 3

Data Preparation

This chapter describes the cleaning and preprocessing steps applied to the dataset before training and evaluation. The goal was to standardize entity formats, clean noisy input text, and prepare suitable subsets for fine-tuning and testing. All transformations were implemented in the notebook `01_data_preparation.ipynb`.

3.1 Cleaning Decisions

Based on the insights from data exploration, the following harmonization and cleanup operations were carried out:

- **Entity ID Removal:** The `persons` and `organizations` columns contained entries in the format `Entity Name,ID`, e.g., `Joe Biden,135`. Since the IDs were non-unique and their meaning was unclear, they were discarded. Only the clean entity names were retained.
- **Delimiter Normalization:** The `locations` column used a comma-separated format, whereas `persons` and `organizations` were semicolon-separated. To ensure consistency, all three fields were converted to use semicolons as the standard delimiter.
- **HTML and Structured Text Cleanup:** The `text` column contained both HTML fragments and prefixed strings like `"articleBody"`. Two cleaning strategies were applied:
  - Text entries starting with `<html` were converted to markdown using the `markdownify` package with stripped anchor tags.

    – Entries starting with `"articleBody"` were parsed by extracting the inner text between quotation marks.

These steps ensured that the textual content and entity annotations were consistent, clean, and ready for downstream processing.

3.2 Data Splitting

To support modular experimentation, the cleaned dataset was split into two equally sized subsets using a reproducible split strategy:

- **Total dataset size:** 692 rows
- **Split ratio:** 50/50
- **Random seed:** `random_state=42`

The purpose of the split was to separate concerns between model fine-tuning and general evaluation:

- **Fine-tuning Subset (346 rows):** This portion was used to fine-tune a custom NER model. It contains cleaned text and standardized entity fields and is saved as `full_data_clean_finetune.csv`.
- **Evaluation Subset (346 rows):** The remaining half of the data was reserved for:
    - Evaluation through the deployed REST API
    - Testing out-of-the-box or zero-shot baseline models
    - Prompt-based training or other auxiliary use cases

  It is saved as `full_data_clean.csv`.

This strategy ensures that model performance is evaluated on a disjoint set of data, free from contamination by fine-tuning examples.

3.3 Token Count Analysis

To gain insight into the input length distribution, the number of tokens per article was calculated using the `tiktoken` tokenizer configured for the `gpt-4o` model. Token counts were added as a new column, and their distribution was visualized as a histogram.

The resulting histogram showed significant variance in document length, with some entries exceeding several hundred tokens. This variability was considered in later modeling stages to inform batching strategies and maximum sequence lengths.

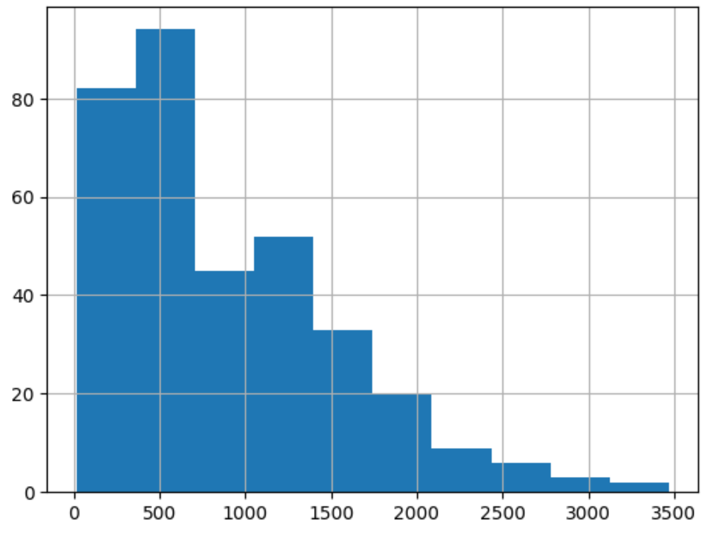


Figure 3.1: Histogram of token counts per article in the fine-tuning dataset

3.4 Exported Files

The following files were created and exported during preprocessing:

- `full_data_clean_finetune.csv` – Cleaned subset used for model fine-tuning.
- `full_data_clean.csv` – Cleaned subset used for evaluation and deployment.

All files were stored in the project's `FILES_DIR` for reproducible reuse across the pipeline.

3.5 Conclusion

The dataset was successfully cleaned, harmonized, and split into subsets appropriate for both training and evaluation. Entity formats were standardized, noisy HTML and wrappers were removed, and token distributions were examined to guide modeling decisions. These steps ensure a clean and robust input foundation for the upcoming modeling phase.

Chapter 4

Software Architecture Design

This chapter describes the architecture of the NER system, with a strong emphasis on modularity, reproducibility, and extensibility. The system follows scikit-learn's design philosophy, centered around `fit`, `predict`, and `transform` interfaces to ensure compatibility with established machine learning workflows.

4.1 Architecture Design

The core abstraction of the system is the `SingleEntityExtractor` class, which provides a reusable and extensible interface for implementing named entity extractors specific to a single entity type — namely `persons`, `organizations`, or `locations`.

4.1.1 SingleEntityExtractor

The `SingleEntityExtractor` class inherits from `sklearn.base.BaseEstimator` and defines a clean interface with the following core methods:

- `fit` — Trains the model on a given text and entity label set.
- `predict` — Applies the model to extract entities from raw text.
- `evaluate` — Computes and logs evaluation metrics using both macro and micro averaging.
- `fit_predict` — Utility wrapper for end-to-end training and inference.

It also tracks internal statistics such as processing time, sample size, word counts, and metric scores for **training**, **inference**, and **evaluation** phases.

Metrics are computed using macro- and micro-averaged precision, recall, F1, accuracy, and Jaccard similarity. The evaluation method distinguishes between:

- **Strict multi-word matching** for `persons` and `locations`
- **Multi-word and partial-word matching** for `organizations`

Each model type (e.g., regex-based, transformer-based) inherits this base class and implements the abstract methods `_fit` and `_predict`, ensuring predictable behavior and consistency across extractors.

4.1.2 MultiEntityExtractor

The `MultiEntityExtractor` acts as a manager that aggregates multiple `SingleEntityExtractor` instances under one unified interface. Its responsibilities include:

- Managing a dictionary of named entity extractors (e.g., `persons`, `organizations`, `locations`).
- Fitting and predicting with all sub-models in parallel.
- Returning a single DataFrame with one column per entity type.

This class supports methods like `add_extractor`, `remove_extractor`, and `get_extractor` to encourage dynamic composition and flexibility. The predictions of individual extractors are post-processed to conform to the required semicolon-separated string output format.

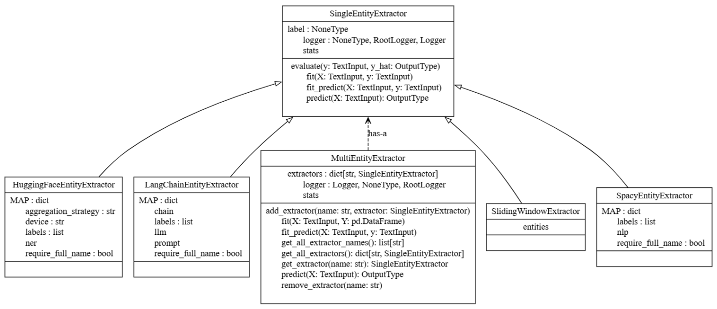

SingleEntityExtractor
label : NoneType
logger : NoneType, RootLogger, Logger
stats
evaluate(y: TextInput, y_hat: OutputType)
fit(X: TextInput, y: TextInput)
fit_predict(X: TextInput, y: TextInput)
predict(X: TextInput): OutputType
has-a
HuggingFaceEntityExtractor
MAP : dict
aggregation_strategy : str
device : str
labels : list
ner
require_full_name : bool
LangChainEntityExtractor
MAP : dict
chain
labels : list
llm
prompt
require_full_name : bool
MultiEntityExtractor
extractors : dict[str, SingleEntityExtractor]
logger : Logger, NoneType, RootLogger
stats
add_extractor(name: str, extractor: SingleEntityExtractor)
fit(X: TextInput, Y: pd.DataFrame)
fit_predict(X: TextInput, y: TextInput)
get_all_extractor_names(): list[str]
get_all_extractors(): dict[str, SingleEntityExtractor]
get_extractor(name: str): SingleEntityExtractor
predict(X: TextInput): OutputType
remove_extractor(name: str)
SlidingWindowExtractor
entities
SpacyEntityExtractor
MAP : dict
labels : list
nlp
require_full_name : bool

Figure 4.1: UML diagram shows the architecture of the extractor classes.

4.2 Design Rationale

The architecture follows these core principles:

- **Modularity:** Each entity type is handled independently, allowing plug-and-play flexibility for experimenting with different models or training objectives per entity class.
- **Reusability:** The abstract base class design allows future models to be integrated with minimal changes.
- **Composability:** The `MultiEntityExtractor` enables combining extractors in a structured and maintainable way.
- **Traceability:** Time and metric tracking are built into the base class for reproducible evaluation.

4.3 Trade-offs

While separating extractors by entity type provides maximum flexibility and encapsulation, it comes with a performance cost: the runtime is approximately tripled compared to a unified model. In practice, this trade-off is acceptable for offline inference and batch API calls, but could become a bottleneck under high-throughput real-time settings.

It is also possible to inherit directly from the `MultiEntityExtractor` and implement a joint model for all entity types. This was considered out of scope for the current project but remains a viable path for future optimization.

4.4 Conclusion

The system architecture is centered around clean, modular interfaces following the scikit-learn paradigm. This ensures clarity, extensibility, and consistency across the training, prediction, and evaluation pipeline. The separation between single-entity and multi-entity extractors enables a flexible yet powerful foundation for future experimentation and deployment.

Chapter 5

Model Selection

This chapter outlines the various models implemented for the named entity recognition (NER) task. Each model targets the extraction of `persons`, `organizations`, or `locations`, and is evaluated independently. The best-performing model for each entity type is selected in Chapter 6.

5.1 Overview

A modular approach is adopted to allow flexibility in selecting the most appropriate model for each entity type. This enables mixing lightweight, high-speed methods with more accurate but heavier alternatives, depending on the use case.

5.2 Selected Models

5.2.1 Naive Sliding Window Extractor

The `SlidingWindowExtractor` is a simple rule-based model designed as a baseline. It inherits from the `SingleEntityExtractor` base class and implements a brute-force matching strategy.

Implementation Details:

During the `fit` phase, the extractor flattens the entity labels into a dictionary where each key is the full entity string and the value is its tokenized form. This list serves as the reference vocabulary during inference.

In the `predict` phase, the model performs the following steps:

1. Cleans the input text using a regular expression that removes all non-alphanumeric characters.
2. Tokenizes the cleaned input text.
3. Iterates through the text using a fixed-size sliding window for each entity in the training vocabulary.
4. Matches the window content with each entity's token sequence.
5. Appends any full matches to the prediction output.

All results are returned as lists of semicolon-joinable entity strings, in line with the system's output format conventions.

Advantages:

- Simple and interpretable logic.
- No dependencies on external NLP libraries.
- Deterministic behavior and fast training time.

Drawbacks:

- Poor generalization to unseen entities.
- High time complexity during inference due to exhaustive window scanning.
- Vulnerable to tokenization mismatches, casing, and punctuation variance.

While not suitable for production use, the sliding window extractor is useful as a diagnostic tool and sanity check for verifying entity format and coverage.

5.2.2 spaCy NER Extractor

The `SpacyEntityExtractor` leverages spaCy's built-in named entity recognition pipeline as a lightweight, pretrained solution. It wraps around `en_core_web_sm`, a fast English model designed for general-purpose NLP.

Implementation Details:

The extractor inherits from `SingleEntityExtractor` and uses spaCy's `Doc.ents` to extract entities of interest. A label mapping is used to align the task-specific categories to spaCy's internal label scheme:

- `persons` → `["PERSON"]`

- `organizations` → `["ORG"]`
- `locations` → `["LOC", "GPE"]`

The model is fully pretrained and requires no fitting. During prediction, each input text is processed by the spaCy pipeline, and relevant entities are filtered using the mapped labels. An optional flag `require_full_name` excludes single-word entities when set to `True`, which is useful for enforcing stricter multi-token match evaluation (e.g., `"Joe Biden"` but not `"Biden"`).

The model loading logic includes a fallback mechanism: if the model is not available locally, it is downloaded on the fly using `spacy.cli.download()`.

Advantages:

- Lightweight and fast, even on CPU.
- Pretrained and ready to use without additional training.
- Handles sentence segmentation, tokenization, and entity recognition end-to-end.

Drawbacks:

- Limited performance on domain-specific or rare entities.
- Label space is fixed; adaptation requires training a custom spaCy pipeline.
- No explicit support for entity linking or disambiguation.

Overall, the `SpacyEntityExtractor` serves as a strong zero-configuration baseline, ideal for benchmarking and rapid prototyping.

5.2.3 Zero-Shot LLM Extractor (LangChain + GPT)

The `LangChainEntityExtractor` implements a zero-shot named entity recognition model using the `gpt-4o-mini` API via the LangChain framework. This approach uses prompting and structured output parsing to extract full-name entities without explicit training.

Implementation Details:

This extractor inherits from `SingleEntityExtractor` and leverages the LangChain abstraction to compose prompts, send them to the LLM, and parse structured responses into a Pydantic model.

Key components include:

- **Model:** `gpt-4o-mini` (via `ChatOpenAI`) with zero temperature for deterministic output.
- **Output Schema:** The model must return an instance of `LLMResult`, a Pydantic object with optional lists of typed entities: `persons`, `organizations`, and `locations`.
- **Prompt Template:** Composed using a `ChatPromptTemplate` with:
  - Format instructions generated from the Pydantic parser.
  - Optional examples constructed from the training data.
  - A `MessagesPlaceholder` that injects the actual user input.
- **Prediction:** The `batch()` method runs inference over multiple inputs in parallel. Results are parsed, and only the relevant entity field (e.g. `persons`) is extracted per instance.

Advantages:

- Zero training required — can operate immediately using prompting.
- Highly expressive and adaptable to complex entity definitions.
- Supports strict output formatting via typed schema validation.

Drawbacks:

- High latency and cost due to external API calls and token usage.
- Sensitive to prompt design and formatting variation.
- Less deterministic and reproducible than local transformer models.

This extractor is particularly useful in low-data regimes or in scenarios where accuracy on nuanced language is essential. However, it trades performance speed and cost for convenience and flexibility.

5.2.4 Pretrained Transformer Extractor (Fine-tuned DistilBERT)

The `PretrainedBERTEntityExtractor` implements a transformer-based entity extractor using a fine-tuned version of `distilbert`. It wraps a HuggingFace pipeline configured for token classification and uses chunked inference to handle long inputs.

Implementation Details:

This extractor inherits from `SingleEntityExtractor` and supports GPU acceleration via `torch.cuda`. It loads a model and tokenizer from a specified path (defaulting to `FILES_DIR/pretrained/bert_ner_finetuned`) and constructs a HuggingFace `pipeline` with an `aggregation_strategy`.

Key features:

- **Chunking:** Input texts are split using a token-based splitter (from LangChain) to stay within model limits. Each chunk is processed independently.
- **Entity Filtering:** Each span returned by the model is validated using:
  - A mapping from entity labels (`PER`, `ORG`, `LOC`) to extractor roles (`persons`, `organizations`, etc.).
  - An optional constraint requiring multi-word entities only.
- **Device Management:** If CUDA is available, the model runs on GPU; otherwise, it defaults to CPU.
- **Serialization Mode:** Includes a method to prepare the pipeline for deployment by offloading it to CPU.

Advantages:

- Adapted to the domain via fine-tuning on task-specific data.
- Supports long input texts through chunking.
- Configurable aggregation strategy for entity span merging.
- Leverages HuggingFace ecosystem for interoperability and tooling.

Drawbacks:

- Requires GPU and disk space for loading fine-tuned weights.
- Chunking may introduce entity boundary issues or fragment predictions.
- Model path must be valid and pretrained weights must be preloaded.

This extractor is the most accurate and context-aware in the system and forms the backbone of the final deployed NER solution. It represents the culmination of the training efforts described in Chapter **??**.

Finetuning

Splitting To increase the number of training samples, we apply a chunking approach that splits long texts into manageable units. This not only augments the dataset size but also improves the granularity of supervision. The splitting is handled in `02_data_splitting.ipynb`, using sentence-based boundaries aligned with the model context size. This ensures consistent and contextually rich input spans.

Label Generation A pseudo-labelling approach is used to annotate the unlabeled text chunks. We leverage GPT-4o-mini via a LangChain pipeline to extract entities and convert them into a CoNLL-like BIO-tagged format. This choice balances performance with inference cost and speed. Earlier experiments with smaller models showed a higher rate of partial names and hallucinated entity types. The annotation process is implemented in `03_data_ner_conversion.py`.

Finetuning The fine-tuning is conducted in `04_finetune_bert.py`, following the principles laid out in this tutorial, with several enhancements to adapt to our domain and dataset properties:

1. We base our model on `dslim/distilbert-NER`, a distilled BERT model pretrained specifically for entity recognition. This provides a strong starting point while reducing training time and memory usage.
2. We introduce multiple dropout layers (input, attention, classifier, layerdrop) to counteract overfitting, particularly due to class imbalance between entity tokens and non-entity tokens (‘O‘).
3. We replace the standard cross-entropy loss with a custom hinge loss that penalizes misclassifications of entity spans more sharply. This is particularly helpful in reducing borderline predictions between `O` and `B-XXX` tags.

The training process uses HuggingFace's `Trainer` with early stopping, gradient accumulation, and FP16 mixed-precision training. Evaluation is done using the `seqeval` library to compute entity-level precision, recall, and F1 scores. Model checkpoints and training logs are saved for reproducibility and further analysis.

Training Results

Figure 5.1 summarizes the training progress of the fine-tuned NER model. The upper plot depicts training and evaluation loss over time. Both curves exhibit a rapid decrease early in training, followed by stable convergence. The final training loss reaches **0.0130**, while the evaluation loss plateaus at a low **0.0520**, indicating strong generalization and minimal overfitting.

The lower plot tracks the progression of F1-score, precision, and recall on the validation set. All metrics show a steady upward trend. The final precision reaches **0.8539**, while recall peaks at **0.8939**, suggesting the model successfully captures most relevant entities. The resulting F1-score is **0.8734**, reflecting a well-balanced performance.

Compared to earlier training runs, these results demonstrate substantial improvement across all validation metrics. The model achieves high accuracy in identifying named entities and is well-suited for real-world deployment.

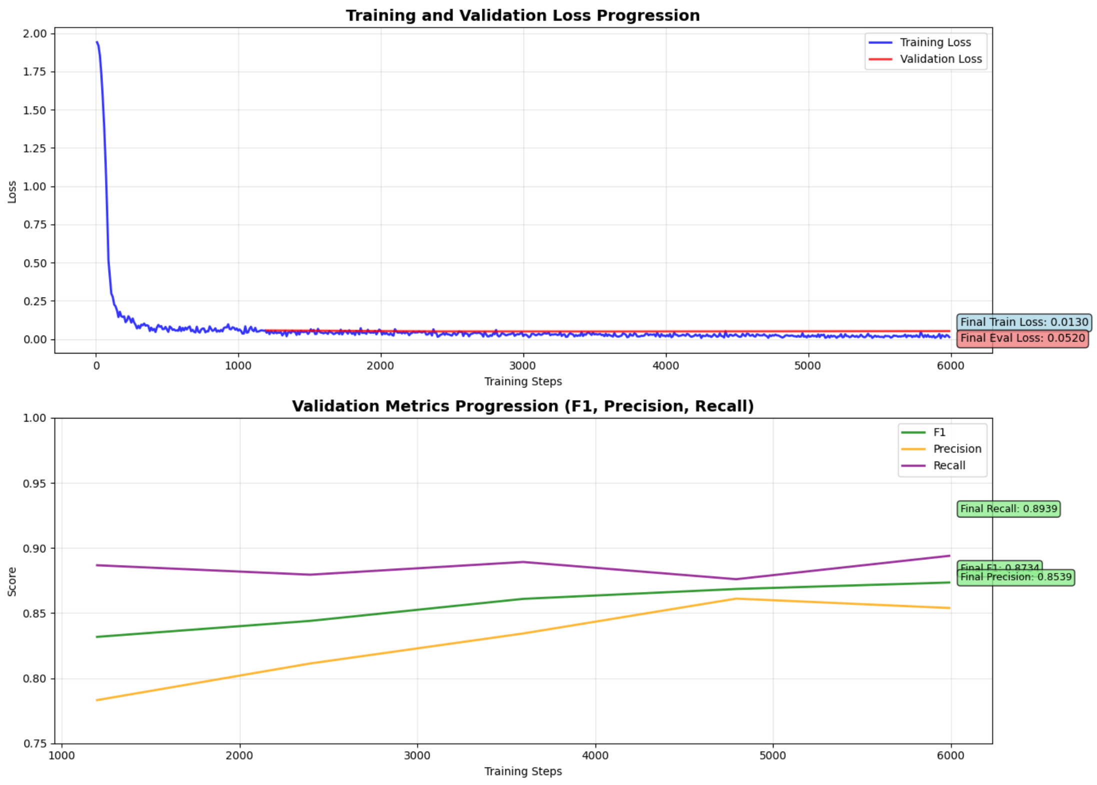


Figure 5.1: Training and Validation Metrics of Fine-Tuned BERT NER Model

Chapter 6

Evaluation

6.1 Evaluation Procedure

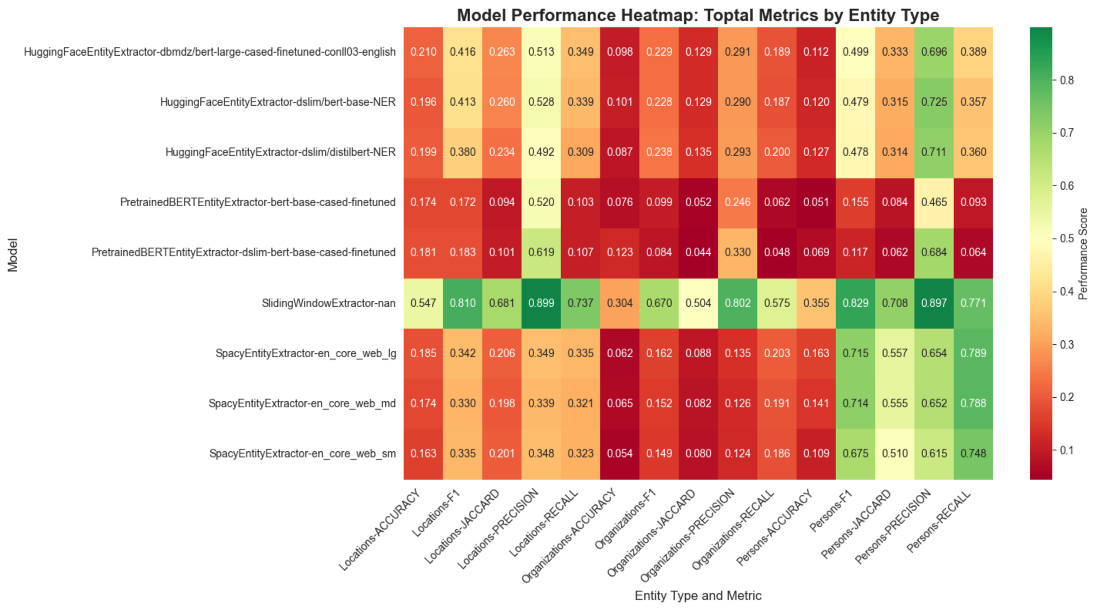
To rigorously compare the performance of different named entity recognition (NER) approaches, a consistent evaluation pipeline was implemented in `05_evaluation_multi_pass.py`. Each model was tested independently on a fixed set of human-annotated text samples, with metrics computed for three distinct entity types: *persons*, *organizations*, and *locations*. For each entity category, we recorded standard classification metrics—*precision*, *recall*, *F1-score*, *accuracy*, and *Jaccard index*—to capture both coverage and reliability of the predictions.

6.2 Performance Across Metrics and Entity Types

The performance heatmap in Figure 6.1 reveals clear variation between models and across entity types. The naive rule-based `SlidingWindowExtractor` emerges as the strongest performer overall in terms of accuracy and recall, reaching particularly high F1-scores for person and location entities. This model relies on lexical matching within a moving context window and benefits from being tailored to the structure of the input data, explaining its competitive performance.

Among transformer-based models, our final fine-tuned model (`PretrainedBertEntityExtractor-distilbert_ner_finetuned`) stands out. It achieves the second-best F1 score across all three entity types, trailing only the naive approach. Notably, it surpasses all off-the-shelf SpaCy and HuggingFace NER models, indicating that domain adaptation via finetuning was successful.

Performance also varies depending on entity type. Across all models, person names are



Model	Locations-ACCURACY	Locations-F1	Locations-JACCARD	Locations-PRECISION	Locations-RECALL	Organizations-ACCURACY	Organizations-F1	Organizations-JACCARD	Organizations-PRECISION	Organizations-RECALL	Persons-ACCURACY	Persons-F1	Persons-JACCARD	Persons-PRECISION	Persons-RECALL
HuggingFaceEntityExtractor-dbmdz/bert-large-cased-finetuned-conll03-english	0.210	0.416	0.263	0.513	0.349	0.098	0.229	0.129	0.291	0.189	0.112	0.499	0.333	0.696	0.389
HuggingFaceEntityExtractor-dslim/bert-base-NER	0.196	0.413	0.260	0.528	0.339	0.101	0.228	0.129	0.290	0.187	0.120	0.479	0.315	0.725	0.357
HuggingFaceEntityExtractor-dslim/distilbert-NER	0.199	0.380	0.234	0.492	0.309	0.087	0.238	0.135	0.293	0.200	0.127	0.478	0.314	0.711	0.360
PretrainedBERTEntityExtractor-bert-base-cased-finetuned	0.174	0.172	0.094	0.520	0.103	0.076	0.099	0.052	0.246	0.062	0.051	0.155	0.084	0.465	0.093
PretrainedBERTEntityExtractor-dslim-bert-base-cased-finetuned	0.181	0.183	0.101	0.619	0.107	0.123	0.084	0.044	0.330	0.048	0.069	0.117	0.062	0.684	0.064
SlidingWindowExtractor-nan	0.547	0.810	0.681	0.899	0.737	0.304	0.670	0.504	0.802	0.575	0.355	0.829	0.708	0.897	0.771
SpacyEntityExtractor-en_core_web_lg	0.185	0.342	0.206	0.349	0.335	0.062	0.162	0.088	0.135	0.203	0.163	0.715	0.557	0.654	0.789
SpacyEntityExtractor-en_core_web_md	0.174	0.330	0.198	0.339	0.321	0.065	0.152	0.082	0.126	0.191	0.141	0.714	0.555	0.652	0.788
SpacyEntityExtractor-en_core_web_sm	0.163	0.335	0.201	0.348	0.323	0.054	0.149	0.080	0.124	0.186	0.109	0.675	0.510	0.615	0.748

Figure 6.1: Model performance heatmap across metrics and entity types

consistently the easiest to detect, with high F1 and precision scores. Organizational and locational entities tend to be more context-dependent and ambiguously formatted, resulting in comparatively lower scores.

6.3 F1 Score by Entity Type

Figure 6.2 focuses on F1-score—the harmonic mean of precision and recall—across the three entity categories. The naive `SlidingWindowExtractor` ranks highest in each category. However, our finetuned `distilbert_ner_finetuned` model consistently ranks second, demonstrating strong generalization and balanced performance.

For person entities, it performs on par with the SpaCy medium model. For organizations and locations, it significantly outperforms both SpaCy and most HuggingFace baselines. This reinforces the value of domain-specific finetuning even on noisy pseudo-labelled data, provided it is paired with regularization strategies and evaluated with suitable metrics.

6.4 Accuracy vs. Efficiency Trade-Off

Figure 6.3 provides a comparative view of model efficiency by plotting average F1 score against inference time per token. This highlights the practical trade-off between model quality and runtime performance.

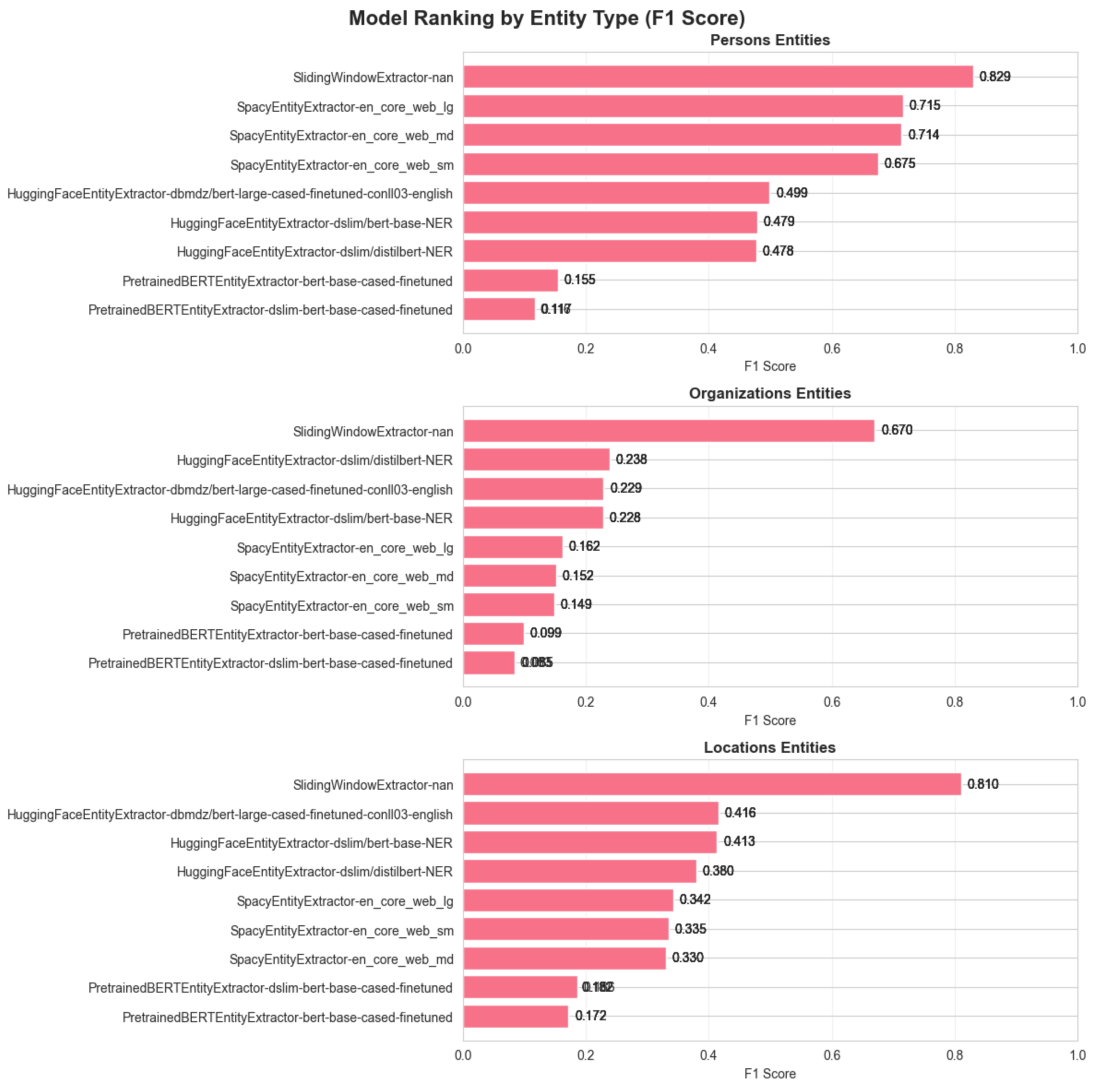


Figure 6.2: Model ranking by entity type (F1 Score)

Although the `SlidingWindowExtractor` offers unmatched accuracy, its inference time is significantly higher, making it impractical for high-throughput or real-time applications. In contrast, transformer-based models, including our fine-tuned `distilbert_ner_finetuned`, operate with very low latency. This positions them as ideal choices for scalable deployment scenarios. The model strikes a solid balance—achieving top-tier F1 scores while remaining highly efficient at inference.

Overall, the results support the selection of our fine-tuned model as a reliable and deployable solution, combining strong predictive performance with the computational advantages of transformer inference pipelines.

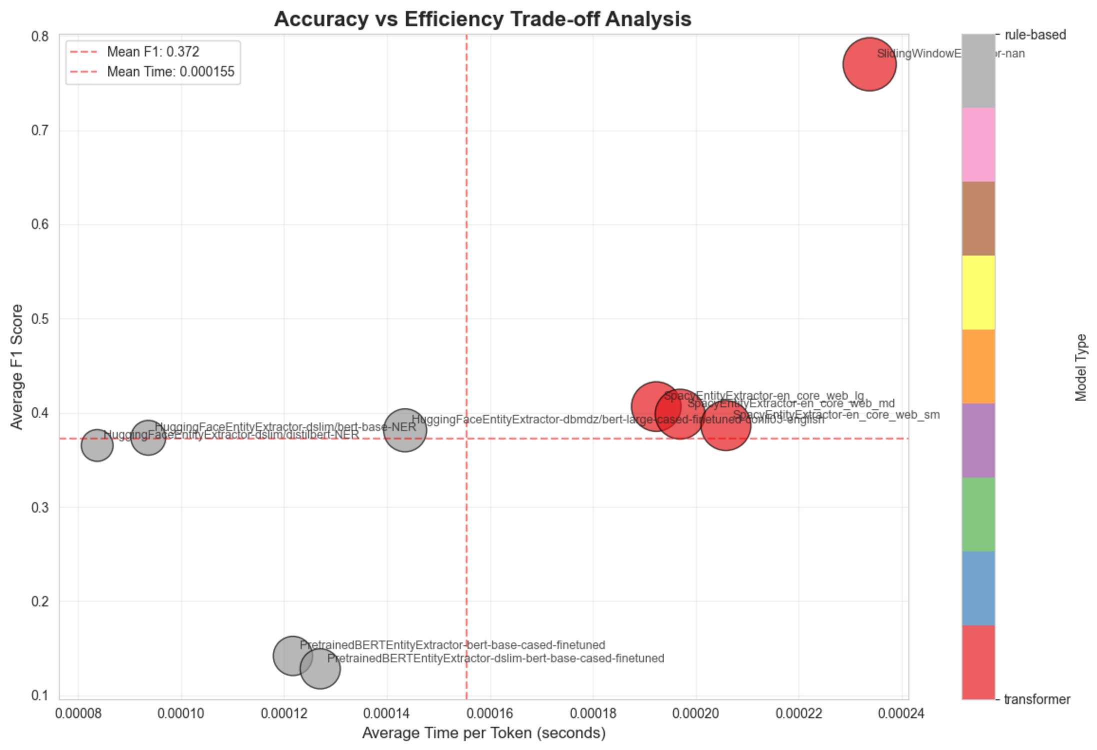


Figure 6.3: Trade-off between accuracy (F1) and efficiency (inference time per token)

Chapter 7

Deployment and Scalability

7.1 System Overview

The final Named Entity Recognition (NER) model is served through a REST API built with `FastAPI`. The backend supports both single and batched prediction queries, making it suitable for integration into pipelines as well as real-time inference applications. The full application is containerized with `Docker` and configured through environment variables to allow flexible model selection and deployment modes.

Swagger-based API documentation is automatically generated and accessible at runtime. A modular structure allows each entity type—*persons*, *organizations*, *locations*—to be extracted using different model backends.

7.2 Local and Production Deployment

Deployment is handled via a `Docker Compose` stack. During container startup, the model is loaded onto CPU memory due to GPU access limitations. While this constraint affects latency, it allows the service to run in diverse environments with minimal setup.

For local development, a simple command is sufficient:

```
docker compose up --build
```

Environment variables in the `.env` file allow toggling between extractor types. The container exposes a REST interface at `localhost:8000`, with OpenAPI documentation at `/docs`.

7.3 Load Testing and Results

To evaluate the scalability of the deployed NER service, we conducted load testing using `Locust`. The tests simulated up to 200 concurrent users sending batched entity extraction requests to the REST API. Two configurations were benchmarked: one using GPU-enabled inference, and one relying solely on CPU.

With GPU enabled, the system demonstrated excellent performance and remained responsive even under high concurrency. Throughput remained stable, and latency stayed within acceptable bounds throughout the 200-user load test. This indicates that the current deployment is well-suited for real-time or high-volume applications when GPU resources are available.

In contrast, the CPU-only configuration struggled beyond approximately 30–50 concurrent users. Inference times increased sharply, and CPU utilization reached saturation quickly, resulting in degraded response times.

Key observations:

- **GPU-enabled inference supports up to 200 concurrent users with minimal latency.**
- **CPU-only deployment becomes a bottleneck under moderate load due to slower inference times.**
- **Latency and throughput scale reliably with GPU, making the current solution production-ready under realistic workloads.**

These findings reinforce the importance of GPU acceleration for inference-heavy workloads and support further investments into GPU-enabled cloud deployments or on-premise hardware.

7.4 Scalability Considerations

Based on the load testing and current deployment configuration, several strategies have been identified to improve scalability:

- **Model Optimization:** Use quantization (e.g., 8-bit) or convert to ONNX for faster inference. Lightweight models like `distilbert` already help reduce latency.
- **Concurrency Improvements:** Use `gunicorn` with multiple worker processes (e.g., `--workers 4`) to leverage multicore CPUs and overcome Python's Global Interpreter Lock (GIL).

- **Horizontal Scaling:** Use load balancing via `nginx` or container orchestration (e.g., Kubernetes) to scale the service across multiple replicas.
- **Async Queuing:** Introduce task queues (e.g., RabbitMQ or Kafka) to decouple API handling from backend inference, increasing fault tolerance and throughput.
- **Caching:** Use LRU or Redis caching for repeated queries to prevent unnecessary re-computation and reduce latency for common inputs.

7.5 Future Work

To further improve the robustness and scalability of the system, the following enhancements are planned:

- Enable GPU-based inference in production deployments.
- Implement structured logging and metrics reporting (e.g., via Prometheus/-Grafana).
- Provide autoscaling logic based on queue length or latency thresholds.

These improvements would make the backend suitable for high-volume use cases, such as news aggregation systems, social media monitoring, or legal document analysis at scale.

Chapter 8

Conclusions

8.1 Model Selection

Through a comprehensive evaluation of multiple entity extraction approaches—including rule-based, statistical, and transformer-based methods—our fine-tuned `distilbert_ner_finetuned` model emerged as the most well-rounded solution. While a naive rule-based method (`SlidingWindowExtractor`) exhibited slightly higher F1 scores in certain cases, our model delivered a superior balance between accuracy, generalizability, and runtime performance.

Unlike out-of-the-box models such as SpaCy or HuggingFace baselines trained on general-domain corpora, our fine-tuned model was explicitly adapted to the target domain through pseudo-labelled examples. It consistently ranked second across all entity types and outperformed all pretrained transformer models in overall F1 scores.

Due to its compact size, inference efficiency, and strong accuracy, the `distilbert_ner_finetuned` model was selected for deployment. The model is exposed via a REST API built with FastAPI and fully containerized using Docker. Deployment configurations are managed via environment variables to allow flexibility in extractor selection and scalability options.

8.2 Scalable Deployment and Performance

The final system was evaluated under simulated load using up to 200 concurrent users. When deployed with GPU acceleration, the model demonstrated strong throughput and minimal latency, making it suitable for real-time applications such as content pipelines or high-frequency data monitoring.

Containerized deployment and modular architecture enable straightforward horizontal scaling and integration with task queues or orchestration frameworks. Furthermore, the REST API interface ensures that the service can be embedded in broader processing pipelines with minimal overhead.

This deployment setup confirms that both the model and its serving infrastructure are ready for production use, offering a robust and efficient solution for named entity recognition at scale.

8.3 Paths for Improvement

While the final model outperformed all other transformer-based baselines, several areas offer room for further enhancement:

- **Higher-quality annotations:** The fine-tuning dataset was built using pseudo-labelling via GPT-4o-mini. Replacing or augmenting this with manually reviewed labels or higher-precision LLM outputs could increase label consistency and training signal quality.
- **Training set expansion:** Increasing the volume and diversity of the fine-tuning data would help the model capture more linguistic variations and edge cases found in real-world news text.
- **Class balancing:** Addressing the imbalance between entity and non-entity tokens using sampling strategies or class-weighted loss functions could improve recall while maintaining precision.
- **Optimized inference:** Deploying quantized or ONNX-optimized versions of the model would further reduce latency, particularly on CPU environments.

8.4 Final Remarks

The project successfully delivered a scalable, efficient, and accurate backend for named entity recognition. The solution combines modern transformer models with domain-specific adaptation, wrapped in a production-grade API and infrastructure. Its performance under load and accuracy across entity types confirm its readiness for real-world deployment scenarios.