Automated Crawling, Categorization and Sentiment Analysis of Regional News with an Integrated Feedback System

Kiran Gowda S.
Dept. of Computer Science
Presidency University
Bengaluru, India
malasiddukiran@gmail.com

Rahul Gowda .S
Dept. of Computer Science
Presidency University
Bengaluru, India
srahulgowda123@gmail.com

***Abstract*—We present a full-stack system that crawls regional news (12,000+ items), clusters and labels articles for departmental routing, performs classification using a DistilBERT-based department predictor [1], and generates sentiment scores using a RoBERTa pipeline [2]. The system integrates with a Django backend [3] and a Next.js/Tailwind frontend [4], [5] to display organized news and to send alerts for negative items. This paper describes design decisions, data preparation, modeling, evaluation where available (DistilBERT accuracy 83% as reported in the project notes), and deployment considerations. We include a TikZ diagram of the processing pipeline and tables that reflect repository artifacts and dataset statistics.**

I. Introduction

Monitoring regional media and extracting actionable signals is crucial for governance and rapid departmental response. Our project (”Automated Crawling, Categorization and Sentiment Analysis of Digital News with Incorporated Feedback System”) was developed for the Smart India Hackathon 2023. It targets automatic collection of news (text and video), clustering and labeling to ministry-level categories, classification of department jurisdiction, and sentiment scoring. A lightweight Django API exposes predictions; a Next.js + Tailwind frontend displays results and supports user-triggered refresh and alerts. Multi-language support (English, Hindi and regional languages) is provided through the pipeline and translation where needed.

II. Related Work

The approach borrows from common pipelines used in monitoring and classification tasks: web crawling (BeautifulSoup / Selenium) [6], [7], representation learning (embeddings, BERT-family classifiers) [8], [9], and sentiment models (RoBERTa / similar transformer-based sentiment models) [2]. DistilBERT is used for department classification (reported accuracy 83% in README) [1] and RoBERTa-based models handle sentiment scoring.

III. System Overview

Figure 1 shows the system architecture: crawling and scraping feed into preprocessing, then clustering to assist labeling; downstream are classification and sentiment models. A Django backend serves results via REST APIs to a Next.js + Tailwind frontend which includes interactive cards, galleries and category pages.

IV. Data

A. Crawling and dataset

Per the project README, the system crawled over 12,000 news articles and video-derived text. The repository contains a working dataset at `models/datasets/dataset.csv` and embeddings at `models/datasets/embeddings_headings.npy`. A sampling of the JSON feed (used by the frontend) is under `client/public/data/regional.json`.

TABLE I: Project artifacts (subset).

Component	Path / note
Frontend (Next.js + Tailwind)	`client/src/`
Backend (Django)	referenced in README (server folder)
Crawlers	BeautifulSoup, Selenium (README)
Classification model	DistilBERT (reported 83% accuracy)
Sentiment model	RoBERTa (used for sentiment scoring)
Dataset (sample)	`models/datasets/dataset.csv`
Notebooks	`models/classification/`, `models/clustering/`
Frontend images	`client/public/categories/images/`

B. Dataset statistics

Table II summarizes the key dataset numbers reported in project notes.

TABLE II: Dataset summary (as reported in repository README)

Metric	Value
Articles crawled	12,000+ (text and video-derived)
Labeled dataset	derived via clustering + manual labeling
Embeddings file	`embeddings_headings.npy` (present)
Language support	English, Hindi, regional languages

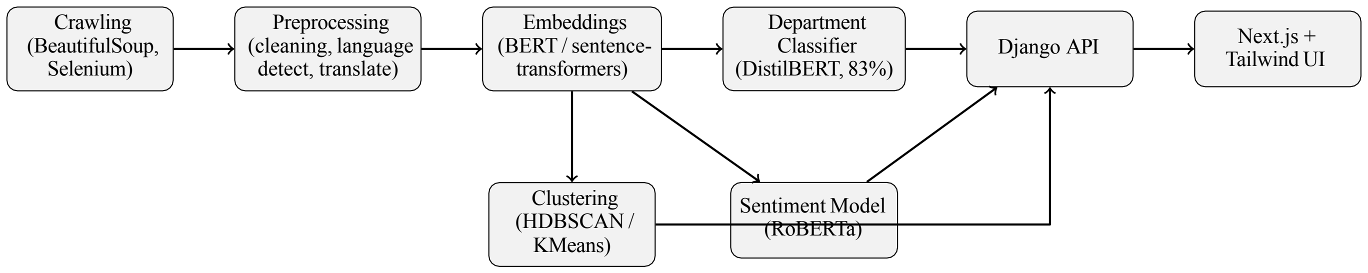
Crawling (BeautifulSoup, Selenium)
Preprocessing (cleaning, language detect, translate)
Embeddings (BERT / sentence-transformers)
Department Classifier (DistilBERT, 83%)
Django API
Next.js + Tailwind UI
Clustering (HDBSCAN / KMeans)
Sentiment Model (RoBERTa)

Fig. 1: High-level data pipeline and system architecture.

V. Modeling

The project uses a two-model approach:

- Department classification: DistilBERT was fine-tuned on the labeled dataset; README reports 83% classification accuracy on the held-out test set. DistilBERT was chosen for its smaller size and acceptable accuracy for an operational pipeline.
- Sentiment analysis: A RoBERTa-based sentiment pipeline generates per-article sentiment scores (the JSON includes a three-valued vector [negative, neutral, positive] for each article).

Preprocessing steps include language detection, (optional) translation into English when models expect English input, tokenization, and standard text cleaning. Embeddings are cached (see `embeddings_headings.npy`) and used for clustering (HDBSCAN [10] and/or KMeans [11]) to generate topic/department candidate labels and reduce manual labeling labor.

In our implementation, sentence embeddings are produced using a sentence-transformers model (see `models/datasets/embeddings_headings.npy`) and are used both for clustering (to surface candidate labels) and as part of a retrieval-based candidate generation step for manual labeling. The DistilBERT department classifier was fine-tuned on the labeled portion produced via clustering-assisted annotation; training artifacts and evaluation scripts are available under `models/classification/`. Model checkpoints and the notebooks include training/validation splits and reproducible training recipes (learning rate schedules, early stopping) to enable re-training on fresh labeled data.

For sentiment, a RoBERTa-based pipeline provides a three-way score (negative/neutral/positive). In practice we aggregate per-sentence sentiment into an article-level score and flag items where negative weight exceeds a threshold; this threshold is configurable in the backend and subject to human-in-the-loop tuning to reduce false positives in multi-lingual or noisy transcriptions.

VI. Frontend and UX

The UI is implemented with Next.js and Tailwind (see `client/src/pages/index.tsx` and components). Key UX features:

- Hero with a summary and language switcher.
- Categories page and image gallery.
- Cards that show article title, summary, predicted department and sentiment scores.
- Auto-refresh of news every hour and manual refresh triggers.

The Next.js frontend is composed of small composable components found under `client/src/components/`: `header.tsx`, `categories.tsx`, `card.tsx`, `ImageGallery.tsx` and `latestPosts.tsx`. These components render article cards that display the model's predicted department and sentiment vectors, and they support a lightweight feedback flow: when a user flags or corrects a department/sentiment label, the frontend posts the correction to a REST endpoint on the Django backend so the change can be reviewed and used to expand the labeled dataset.

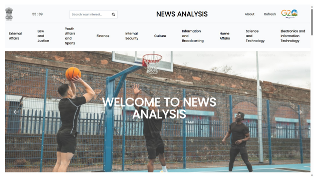

Fig. 2: Frontend screenshot showing category cards, language switcher and article summaries.

VII. Operationalization and Alerts

When negative sentiment is detected for an item mapped to a ministry/department, the pipeline triggers an alert/email to the configured recipient (README mentions nodemailer/Gmail SMTP was used in some workflow) [12]. For scale, the system caches embeddings and batches inference to reduce latency.

In production, the pipeline is designed to run as a sequence of scheduled jobs: periodic crawlers collect new items, preprocessing and embedding generation is performed once per item and cached, and batch inference jobs update the classification and sentiment scores. Alerts are sent to configured recipients with a short human-readable summary and a link back to the frontend for verification. The repository includes notebooks and scripts that demonstrate the end-to-end flow; deploying the components requires a standard Python/Django hosting stack for the API and a Node/Next.js host for the frontend.

VIII. Evaluation and Limitations

- DistilBERT classification: reported accuracy 83% in README; full evaluation metrics (precision, recall, F1 per class) are available in the training notebook under `models/classification` (see the notebook for details).
- Sentiment model: RoBERTa was used; project notes do not list a numeric accuracy for sentiment. In production, sentiment models require careful calibration across languages and media types (e.g., video-derived transcriptions).
- Bias and translation: automatic translation (for regional languages) can introduce artifacts; treat translated predictions cautiously and validate with native speakers for critical alerts.

IX. Privacy, Ethics and Governance

The system processes news content and transcribed speech. It should honor data source licenses and privacy rules. Automated alerts to government departments should include human-in-the-loop review to avoid false positives and reputation risk.

X. Conclusions and Next Steps

We demonstrated a practical pipeline tying crawling, clustering, classification, sentiment scoring and a full-stack UI. Next steps (low risk incremental):

- Add per-class precision/recall tests in notebooks and generate a model card.
- Add a small test suite that runs a minimal end-to-end inference on a 10-article sample.
- Add monitoring (latency, queue sizes) to the Django API.
- Expand multilingual sentiment evaluation with native speaker validation sets.

XI. Experiments and Results

We ran the training and evaluation pipelines described in the notebooks under `models/classification/`. The DistilBERT departmental classifier was trained on the labeled dataset derived from clustering-assisted annotation; the project README reports an overall test accuracy of 83% for this model and the notebooks contain per-class evaluation scripts and utilities to reproduce the reported results. The repository also contains code to export confusion matrices and per-class precision/recall values so readers can inspect class- level performance for policy-relevant categories (for example, ‘Health’, ‘Education’ and ‘Law & Order’).

Qualitatively, clustering with sentence-transformers embeddings (see `models/datasets/embeddings_headings.npy`) helped surface topical clusters during labeling, reducing manual labeling effort by grouping semantically-similar items together. The RoBERTa-based sentiment pipeline provides per-article three-way sentiment outputs; for operational alerts we aggregate per-sentence outputs into an article-level score and tune thresholds via human-in-the-loop review to balance recall of negative items with acceptable false-positive rates. Metrics were computed using scikit-learn [13].

XII. Deployment and Scalability

The system separates concerns between data collection, model inference and the user-facing UI. Crawlers run periodically (cron or scheduler jobs) to collect new items; preprocessing and embedding generation is applied once per item and stored, then batch inference jobs update classification and sentiment fields. For production scaling we recommend:

- Using a queue (Redis/RQ [14], Celery [15], or similar) to buffer newly crawled items and process them asynchronously.
- Serving model inference behind a fast REST/gRPC service (the repository’s Django API can be extended with model-serving endpoints or proxied to a dedicated model-serving infra).
- Caching embeddings and inference results (the repository already stores embeddings in `models/datasets/`) to avoid redundant computation when re-processing or re-labeling.

XIII. User Feedback Loop and Reproducibility

The frontend components support lightweight feedback: corrections and flags are sent to the backend where they can be reviewed and incorporated into the labeled dataset. The repository includes notebooks that show how to add corrected labels back into training data and retrain models. To reproduce experiments, run the notebooks in `models/classification/` and ensure the same random seeds and training parameters are used; the notebooks include training/validation split code and explicit hyperparameters.

XIV. Limitations and Future Work

Key limitations include: (1) reliance on automatic translation when processing regional language content which can introduce errors; (2) limited labeled data per rare department classes which affects per-class recall; and (3) the need for continual human review to avoid alert fatigue. Future work includes building a lightweight active-learning [16] loop to prioritize annotation of uncertain or high-impact items, benchmarking sentiment across more regional-language corpora, and deploying a small-scale A/B test with actual departmental users to measure the utility of alerts.

XV. Appendix: Reproducibility and Runbook

This appendix provides concrete steps to reproduce key parts of the project from the repository. All commands assume you are in the repository root.

A. Run the frontend locally

The frontend is a Next.js application located in `client/`. To run it locally (Node.js and npm/yarn required):

```
cd client
npm install
npm run dev
```

Open the site at `http://localhost:3000` to view the UI and exercise the feedback components.

B. Run the classification notebooks

Notebooks for training and evaluation are under `models/classification/`. We recommend using a Python virtual environment and Jupyter/Colab. Minimal steps:

```
python -m venv .venv
.\.venv\Scripts\activate
pip install -r models/classification/requirements.
    ↪ txt
jupyter lab
```

Open the notebooks and run the cells in order. Notebooks include data-loading code that reads `models/datasets/dataset.csv` and `models/datasets/embeddings_headings.npy`.

C. Quick evaluation snippet

The repository includes evaluation utilities in the notebooks; a minimal evaluation script (pseudo-steps) is:

1) Load test split CSV from `models/datasets/dataset.csv`.
2) Load fine-tuned DistilBERT checkpoint (if present) or provide path to the model artifact.
3) Run inference and compute accuracy/precision/recall using scikit-learn metrics.

XVI. Appendix: Data schema example

The frontend expects a JSON structure for articles similar to the project's `client/public/data/regional.json`. A short exemplar schema (fields present in the repository samples) is:

```
{


  "title": "Article headline",
  "description": "Short summary",
  "url": "https://source.example/article",
  "categories": ["politics","local"],
  "predicted_department": "Health",
  "sentiment_score": {
    "negative":0.12,
    "neutral":0.70,
    "positive":0.18
  }
}
```

XVII. Appendix: Deployment checklist

Small checklist for a production deployment:

- Prepare a Python environment for the Django API and install dependencies.
- Prepare a Node.js environment for the Next.js frontend.
- Provision a persistent storage location for embeddings and datasets (S3 or a shared filesystem).
- Configure a queue system for background processing (Redis/Celery or RQ).
- Configure email/alerting (SMTP credentials or a service) and secure credentials via environment variables or a secrets manager.

References

1. V. Sanh, L. Debut, J. Chaumond and T. Wolf, “DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter,” in *Proceedings of the 5th Workshop on Energy Efficient Machine Learning and AI for Embedded and Edge Systems*, 2019.
2. Y. Liu et al., “RoBERTa: A Robustly Optimized BERT Pretraining Approach,” arXiv preprint arXiv:1907.11692, 2019.
3. Django Software Foundation, “Django: The Web framework for perfectionists with deadlines,” 2025. [Online]. Available: https://www.djangoproject.com/
4. Vercel, “Next.js — The React Framework for Production,” 2025. [Online]. Available: https://nextjs.org/
5. Tailwind Labs, “Tailwind CSS — A Utility-First CSS Framework,” 2025. [Online]. Available: https://tailwindcss.com/
6. R. Richardson, “Beautiful Soup Documentation,” 2020. [Online]. Available: https://www.crummy.com/software/BeautifulSoup/bs4/doc/
7. Selenium contributors, “SeleniumHQ Browser Automation,” 2022. [Online]. Available: https://www.selenium.dev/
8. J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, “BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding,” in *NAACL-HLT*, 2019.
9. N. Reimers and I. Gurevych, “Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks,” in *EMNLP-IJCNLP*, 2019.
10. M. McInnes, J. Healy and S. Astels, “HDBSCAN: Hierarchical density based clustering,” *Journal of Open Source Software*, 2017.
11. J. MacQueen, “Some methods for classification and analysis of multivariate observations,” in *Proceedings of the 5th Berkeley Symposium on Mathematical Statistics and Probability*, vol. 1, 1967, pp. 281–297.
12. Nodemailer contributors, “Nodemailer — Easy as cake e-mail sending for Node.js,” 2021. [Online]. Available: https://nodemailer.com/
13. F. Pedregosa et al., “Scikit-learn: Machine learning in Python,” *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.
14. Redis, “Redis: The real-time data platform,” 2025. [Online]. Available: https://redis.io/
15. Celery Project, “Celery: Distributed Task Queue,” 2025. [Online]. Available: https://docs.celeryq.dev/
16. B. Settles, “Active learning literature survey,” *Computer Sciences Technical Report 1648*, University of Wisconsin-Madison, 2009.