NLP Workbench Toolkit: Architecture, Methods, and UX Rationale

Author: Mohamad Hijazi and Haidar Saleh
Date: December 8, 2025

Introduction

This brief explains three practical decisions behind the submitted NLP Workbench: (i) implementing the toolkit as a *class-based architecture in a single notebook cell*; (ii) selecting specific libraries—`TfidfVectorizer`, `CountVectorizer`, `SGDClassifier`, `LatentDirichletAllocation`, `WordNetLemmatizer`, and `TextBlob`—to cover core text analytics tasks; and (iii) triggering model training from a GUI button (background thread) rather than inline code execution. The aim is to keep the interface responsive, the code tidy and reproducible, and the workflow professional. Screens and outputs are embedded for completeness.

1 Why a Class Architecture in One Cell

1.1 Encapsulation and State

Event-driven GUIs maintain state (e.g., whether a model is trained, whether a cache was loaded). Consolidating the app into a single `NLPWorkbench` class gathers that state in one place: widgets, the pipeline, cache paths, and flags such as `model_ready`. Methods map cleanly to callbacks (`train_classifier_async`, `text_classify`, `visualize_metrics`). A single cell reduces accidental redefinition of globals and "run-order" confusion.

1.2 Reproducible and Readable

Running one self-contained cell makes the notebook deterministic: the GUI is built, events are wired, and the app is ready without hunting through multiple code shards. It also enables artifact caching so later sessions load instantly.

1.3 Safety for Background Work

Training executes in a background thread to avoid blocking Tkinter's main loop. The class shields concurrency details, while the UI thread updates status and progress. That design prevents freeze-ups and improves perceived performance.

2 Why These Packages

2.1 Vectorizers

TF–IDF (`TfidfVectorizer`). Produces weighted sparse features that favor informative terms; supports stop words and n-grams; ideal for linear classifiers at scale.
Bag-of-Words (`CountVectorizer`). Supplies discrete counts that `LDA` expects, complementing TF–IDF used for classification.

2.2 Models

`SGDClassifier` **(hinge).** A linear SVM trained with stochastic gradient descent is efficient on TF–IDF features and offers fast, interpretable predictions. With `class_weight=balanced` it mitigates label imbalance.

`LatentDirichletAllocation`. Unsupervised topic discovery with interpretable top words; perplexity offers a rough model-fit signal.

2.3 NLTK/Text Utilities

`WordNetLemmatizer`. POS-aware normalization reduces inflected forms, improving both vectorizers.
`TextBlob`. A pragmatic sentiment baseline yielding polarity in $[-1, 1]$, easy to explain to non-specialists and light enough for real-time UI feedback.

3 Why Train via a Button (Not Inline Code)

3.1 Responsiveness

Training can take minutes (dataset load, TF–IDF fit, optimization). Triggering it from the GUI in a background thread keeps the window responsive and avoids the operating system's "Not Responding" state.

3.2 Professional Workflow and Caching

Users decide when to pay the training cost; the app then caches the model so later sessions enable classification immediately. This mirrors how production tools let operators control costly jobs.

3.3 Separation of Concerns

The GUI owns operations (train, classify, visualize). Notebook cells simply launch the app; the class handles the rest. This keeps code cleaner and portable to a script.

4 Screens and Outputs

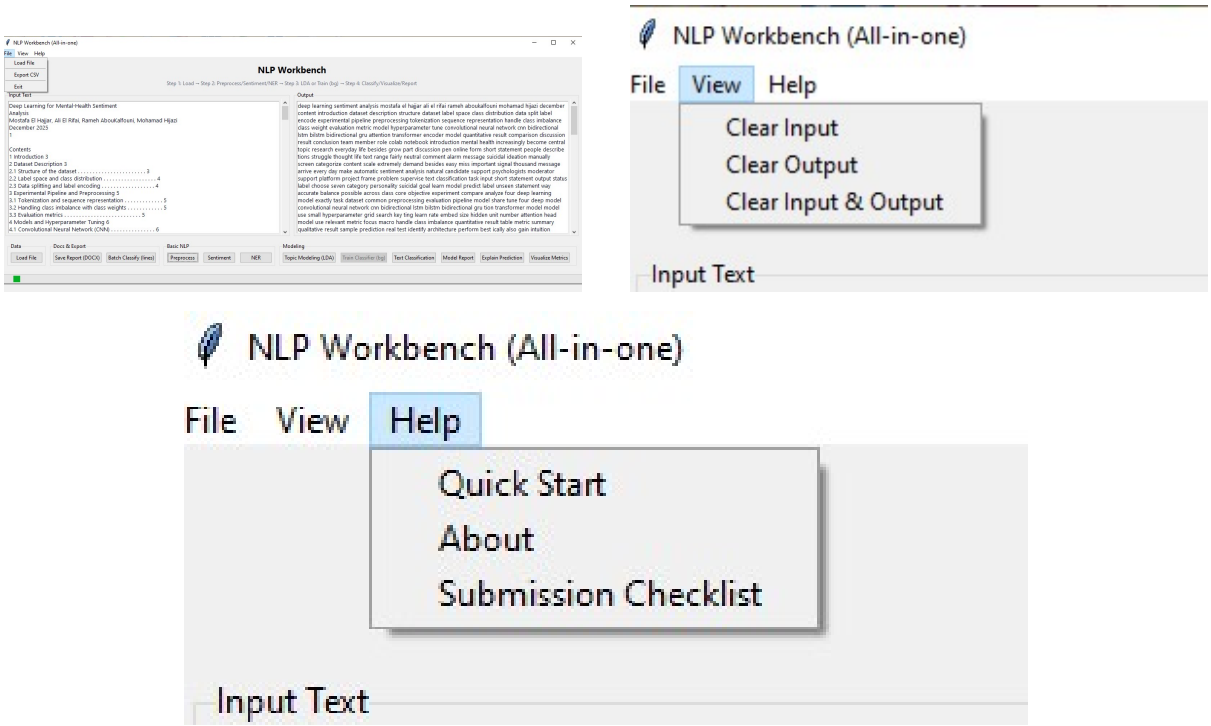


Figure 1: The NLP Workbench: input/output panes, menus, and modeling controls.

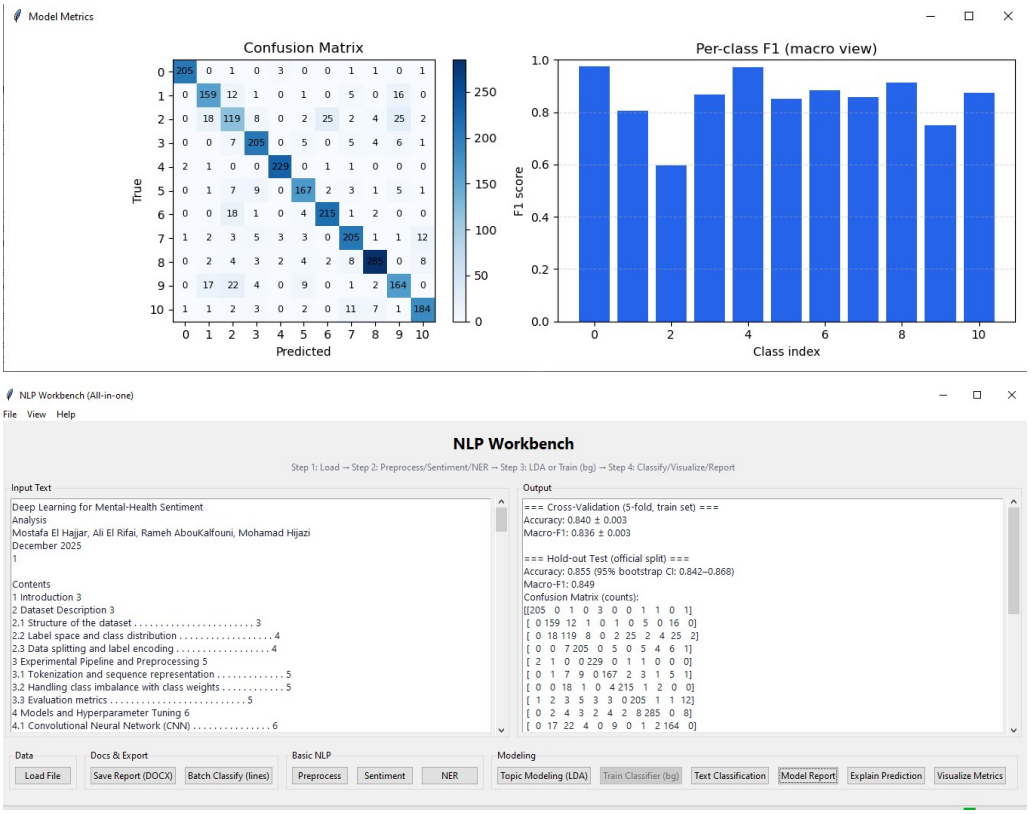


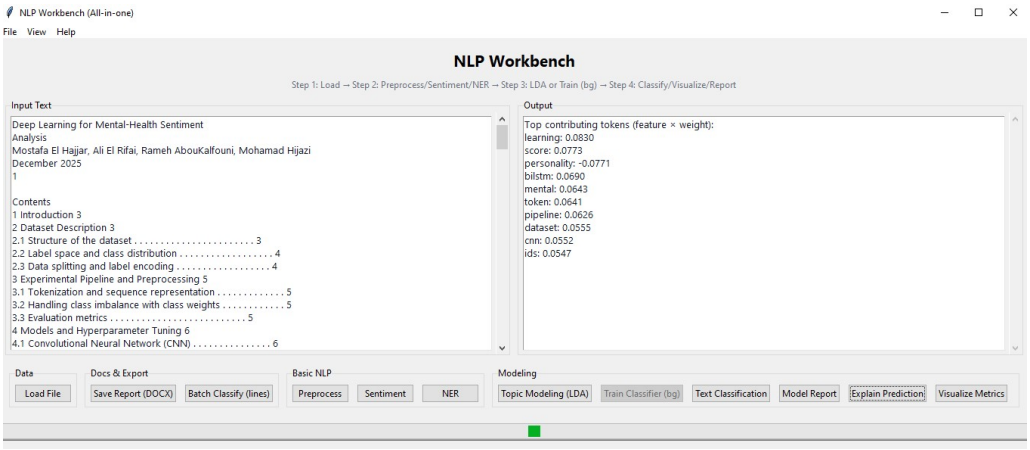
Figure 2: (Top) Confusion matrix and per-class F1. (Bottom) Model report with CV and hold-out metrics.

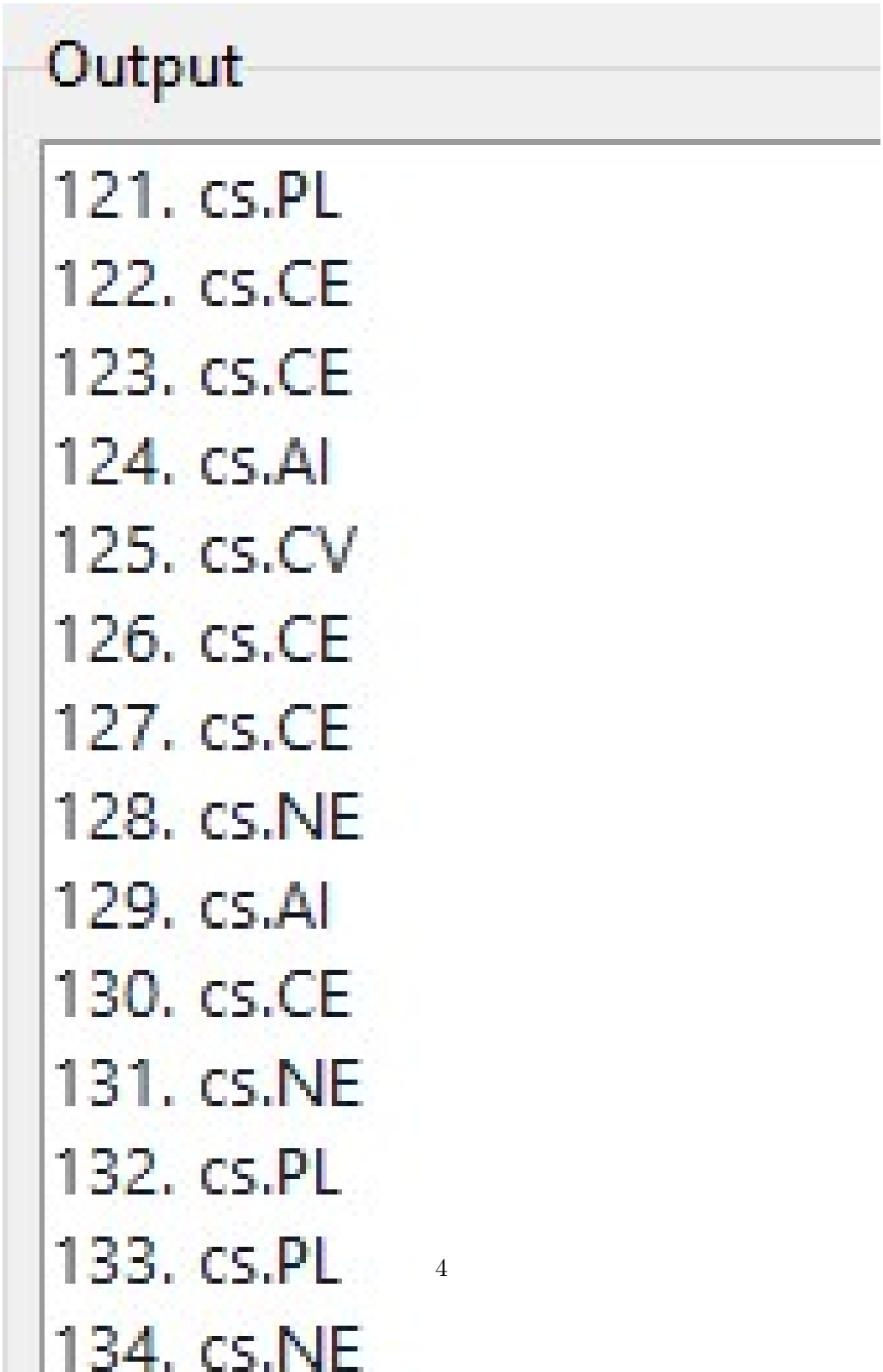
5 Method Choices at a Glance

Component	Rationale
`TfidfVectorizer`	Weighted sparse features; robust baseline for linear models; n-grams and stop-words supported.
`CountVectorizer`	Discrete counts required by LDA; fast and interpretable for topics.
`SGDClassifier`	Linear SVM (hinge) efficient on TF–IDF; good scalability and interpretability.
`LatentDirichletAllocation`	Discovers themes; top-words lists aid qualitative review.
`WordNetLemmatizer`	POS-aware normalization reduces vocabulary noise.
`TextBlob`	Lightweight polarity for real-time UI hints.

Limitations and Next Steps

The classifier is linear; calibration and domain-adaptive embeddings may lift performance. For topics, coherence (e.g., NPMI) can complement perplexity. Shipping a prebuilt model artifact would skip the first training round entirely.


NLP Workbench (All-in-one)
File View Help
NLP Workbench
Step 1: Load → Step 2: Preprocess/Sentiment/NER → Step 3: LDA or Train (bg) → Step 4: Classify/Visualize/Report
Input Text
Deep Learning for Mental-Health Sentiment
Analysis
Mostafa El Hajjar, Ali El Rifai, Rameh AbouKalfouni, Mohamad Hijazi
December 2025
1
Contents
1 Introduction 3
2 Dataset Description 3
2.1 Structure of the dataset 3
2.2 Label space and class distribution 4
2.3 Data splitting and label encoding 4
3 Experimental Pipeline and Preprocessing 5
3.1 Tokenization and sequence representation 5
3.2 Handling class imbalance with class weights 5
3.3 Evaluation metrics 5
4 Models and Hyperparameter Tuning 6
4.1 Convolutional Neural Network (CNN) 6
Output
Top contributing tokens (feature × weight):
learning: 0.0830
score: 0.0773
personality: -0.0771
bilstm: 0.0690
mental: 0.0643
token: 0.0641
pipeline: 0.0626
dataset: 0.0555
cnn: 0.0552
ids: 0.0547
Data
Load File
Docs & Export
Save Report (DOCX)
Batch Classify (lines)
Basic NLP
Preprocess
Sentiment
NER
Modeling
Topic Modeling (LDA)
Train Classifier (bg)
Text Classification
Model Report
Explain Prediction
Visualize Metrics


Output
121. cs.PL
122. cs.CE
123. cs.CE
124. cs.AI
125. cs.CV
126. cs.CE
127. cs.CE
128. cs.NE
129. cs.AI
130. cs.CE
131. cs.NE
132. cs.PL
133. cs.PL
134. cs.NE

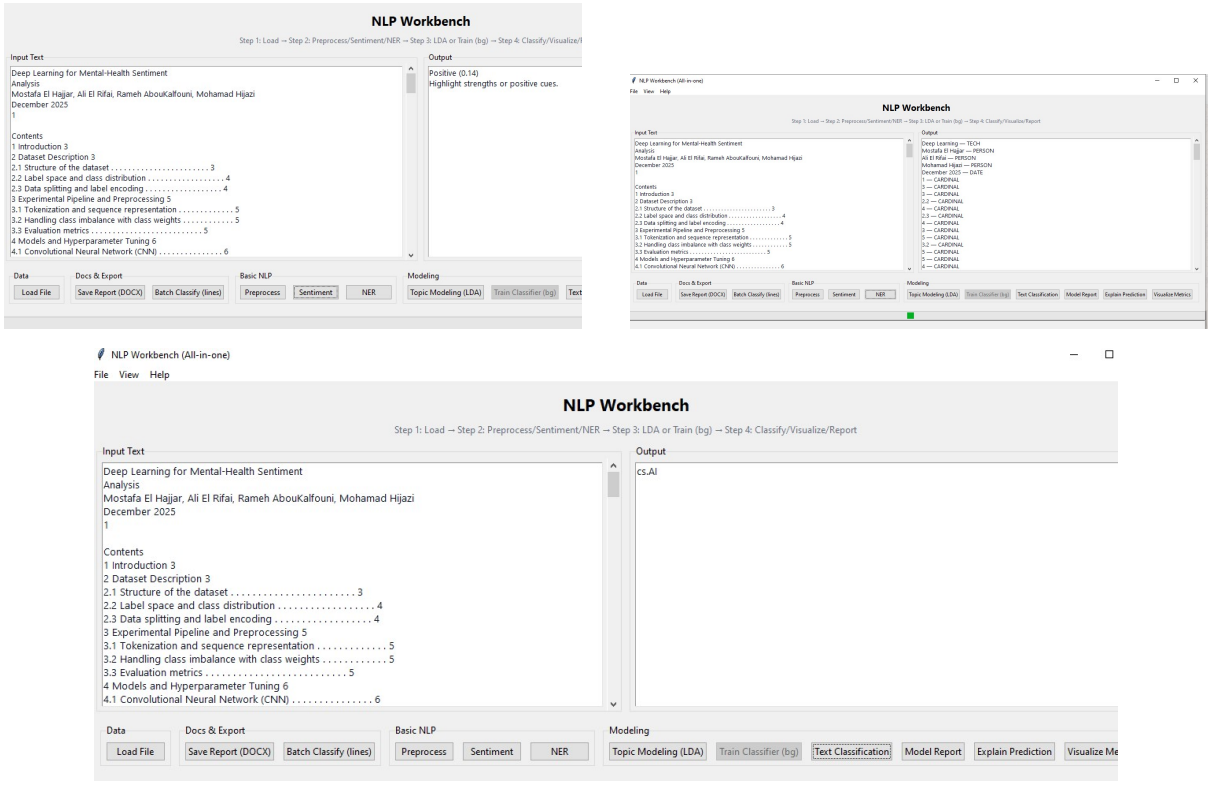


Figure 4: Basic NLP utilities: sentiment (left), NER (right), and a classification snapshot (bottom).

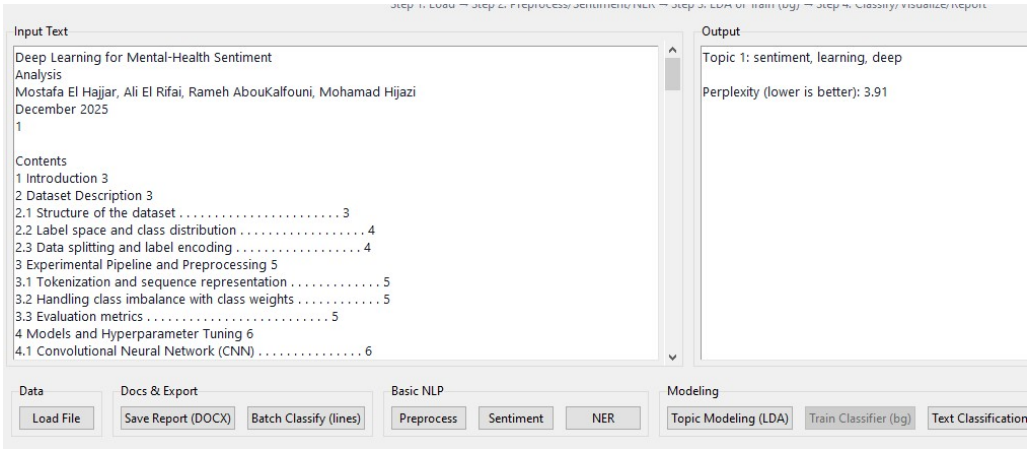


Figure 5: Topic modeling output with top words per topic and perplexity.