NLP Project 2 Report (Tasks 1–4)

February 26, 2026

1 Experimental Setup

I prepared this report from the saved run outputs in `test-results.txt` and the task scripts in `Task1--Task4` (no reruns). Table 1 summarizes how each task was split into train/dev/test.

Table 1: Train/dev/test characteristics

Task	Data source	Split strategy	Actual counts
Task 1	`corpus.txt`	Random sentence split: 80% train / 10% dev / 10% test (seed 42)	96,000 / 12,000 / 12,000 (from 120,000)
Task 2	`corpus.txt`	Random sentence split: 80% train / 10% dev / 10% test (seed 42)	96,000 / 12,000 / 12,000 (from 120,000)
Task 3	`dataset_v1.csv`	Stratified 80% pool + 20% test, then 20% of pool for dev	25,600 / 6,400 / 8,000 (from 40,000)
Task 4	`dot_labeled_data.csv`	Stratified 70% train + 30% temp, then temp split 50/50 into dev/test	140,000 / 30,000 / 30,000 (from 200,000)

2 Task 1: N-gram Language Modeling (MLE)

Modeling. I trained unigram, bigram, and trigram language models with plain MLE. Tokenization and sentence splitting came from shared utilities; rare training words (frequency < 2) were mapped to `<UNK>`, and the same vocabulary mapping was applied to dev/test.

Result.

Table 2: Task 1 test results (MLE perplexity)

Metric	Value
Number of sentences	120,000
Vocabulary size (after `<UNK>` handling)	58,418
Unigram MLE perplexity	3296.2567
Bigram MLE perplexity	∞
Trigram MLE perplexity	∞

The behavior is expected: without smoothing, one unseen bigram/trigram in the test sentence makes the sentence probability zero, so perplexity becomes infinite.

3 Task 2: Smoothing for Bigram/Trigram LMs

Modeling. I reused the same LM pipeline and compared four smoothing methods: Laplace, linear interpolation, absolute-discount backoff, and Kneser–Ney. Interpolation weights were tuned on dev using grid search with step 0.1. Best dev lambdas were $(\lambda_1, \lambda_2) = (0.2, 0.8)$ for bigram interpolation and $(\lambda_1, \lambda_2, \lambda_3) = (0.2, 0.3, 0.5)$ for trigram interpolation.

Result.

Table 3: Task 2 test perplexity by smoothing method

Method	Bigram PPL	Trigram PPL
Laplace	3572.4528	12556.2180
Interpolation	167.9878	88.0896
Backoff (discount 0.75)	158.2786	74.7584
Kneser–Ney (discount 0.75)	**150.0579**	**70.7616**

Kneser–Ney was best for both bigram and trigram. Also, trigram + smoothing clearly outperformed bigram + smoothing, which means the extra context helped once sparsity was handled correctly.

4 Task 3: Sentiment Classification

Modeling. I trained three classifiers: Multinomial NB, Bernoulli NB, and Logistic Regression. For text representation, I used three feature sets: `bow`, `lexicon`, and `bow_lexicon`.

The `bow` part uses `CountVectorizer` with unigram+bigram features (`ngram_range=(1,2)`), `min_df=2`, and `max_features=30000`. The lexicon block has 6 handcrafted sentiment cues (positive/negative counts, polarity difference, normalized polarity, exclamation flag, and negation-aware signal). I tuned NB $\alpha \in \{0.05, 0.1, 0.3, 0.5, 1.0, 2.0\}$, and LR $C \in \{0.1, 0.3, 0.5, 1, 2, 5, 10\}$ with `class_weight` in {`None`, `balanced`}, selecting by dev macro-F1.

Result (best feature set per model).

Table 4: Task 3 model comparison

Model	Best feature set	Best hyperparameter(s)	Dev macro-F1	Test macro-F1	Test Acc.
Multinomial NB	`bow`	$\alpha = 0.05$	0.8491	0.8490	0.8490
Bernoulli NB	`bow`	$\alpha = 0.10$	0.8528	0.8547	0.8550
Logistic Regression	`bow_lexicon`	$C = 0.3$, balanced weight	**0.8974**	**0.9009**	**0.9010**

Table 5: Task 3 feature-set effect (test macro-F1)

Feature set	MNB	BNB	LR
`bow`	0.8490	0.8547	0.8992
`lexicon`	0.6404	0.6441	0.6441
`bow_lexicon`	**0.8510**	0.8539	**0.9009**

Two quick observations match the numbers. First, lexicon-only features were too weak as a standalone representation, but adding them on top of BoW helped LR a bit (0.8992 $\rightarrow$ 0.9009 macro-F1). Second, McNemar tests ($p_{LRvsMNB} = 0.0000$, $p_{LRvsBNB} = 0.0000$, $p_{MNBvsBNB} = 0.0485$) support that LR was not just numerically better but statistically stronger on this split.

5 Task 4: Dot-as-Sentence-Boundary Classification (V2)

Modeling. This task is a binary classifier over dot contexts from `dot_labeled_data.csv`. I used all fields except `label` as input (`prev_token`, `next_token`, `prev_len`, `next_len`, `is_digit_before`) and converted them with `DictVectorizer`. Then I trained Logistic Regression with `liblinear` solver for both L2 and L1 regularization, tuned $C \in \{0.01, 0.1, 1, 10, 100\}$ on dev F1, and tuned the decision threshold on dev by scanning 0.10 to 0.89 with step 0.01.

Result.

Table 6: Task 4 L1 vs L2 comparison

Penalty	Best C	Threshold	Dev F1	Test Acc.	Test Precision	Test Recall	Test F1
L2	10	0.64	0.9993	0.9992	0.9980	0.9990	0.9985
L1	10	0.59	0.9993	**0.9993**	**0.9985**	0.9990	**0.9988**

Both variants were extremely strong, but L1 had the best final F1 and was selected as the final model. The high scores suggest that the available context features already separate boundary vs non-boundary cases very well for this dataset.

6 UI Note

I implemented two interfaces: a desktop Tkinter dashboard (`results_ui.py`) and a local Streamlit app (`localhost_ui.py`). Both support per-task execution and metric inspection, while the Streamlit app additionally includes small interactive demos for LM sentence analysis, sentiment prediction, and dot-boundary testing.