Hindi NLP Assignment Report
Word Vectors + LSTM Classification

NLP A2 Submission

February 18, 2026

1 Objective

The assignment required two tasks:

1. Build Hindi word vectors and compare them with provided pretrained vectors.
2. Build an LSTM-based Hindi news classifier using the BBC Hindi dataset.

After the instructor updates, the final required resources were:

- Pretraining corpus: `ai4bharat/sangraha` with `data_dir="verified/hin"`.
- Vector comparison baseline: Common Crawl Hindi fastText vectors (`cc.hi.300.vec`).

2 Data and Resources Used

2.1 Pretraining Corpus

- Source recorded in `data/corpus/corpus_meta.json`: `ai4bharat/sangraha (verified/hin, streaming)`.
- Target documents requested: 200,000.
- Collected documents: 196,549.
- Raw text line count in file: 1,565,935 (`data/corpus/hindi_corpus.txt`).
- Cleaned line count: 1,359,186 (`data/corpus/hindi_corpus_clean.txt`).

2.2 Pretrained Comparison Vectors

- Common Crawl Hindi fastText file: `models/pretrained/cc.hi.300.vec`.
- For memory/runtime balance, top 400,000 vectors were loaded during comparison/training.

2.3 Classification Dataset

- File: `bbc_hindi_articles_with_categories_cleaned.csv`.
- Valid classes used: 6 (Khel, Bharat, Manoranjan, Vigyan, Videsh, Social).
- After cleaning/length filtering: 4,790 samples.
- Split sizes (`data/bbc_hindi/processed/dataset_summary.csv`):
  Train 3,353, Val 718, Test 719.

3 Detailed Workflow and Implementation

3.1 Step 1: Corpus Acquisition

This step handles the downloading of the Hindi corpus from the AI4Bharat Sangraha dataset. We upgraded the download logic to prioritize the Sangraha source and handle the specific directory structure (`verified/hin`).

The implementation includes the following key features:

- **Source Prioritization**: The script explicitly targets the 'ai4bharat/sangraha' dataset, which provides high-quality, verified Hindi text data.
- **Directory Handling**: It correctly navigates the Hugging Face dataset structure, specifically looking for files within the 'verified/hin' directory to ensure we only download relevant Hindi content.
- **Streaming and Fallback**: The download process uses streaming to handle large files efficiently. It also includes robust error handling to manage potential API issues or network interruptions, ensuring the download can be retried or fail gracefully.
- **Metadata Logging**: Download metadata, including source and timestamp, is logged to `corpus_meta.json` for reproducibility.

3.2 Step 2: Corpus Preprocessing

The raw corpus requires significant cleaning to be suitable for training word vectors. Our preprocessing pipeline ensures that the text is clean, normalized, and tokenized correctly for Hindi.

The preprocessing steps are as follows:

1. **Noise Removal**: We remove URLs, email addresses, and HTML tags using regular expressions to clean the raw text.
2. **Character Filtering**: The script filters out non-Hindi characters while preserving Devanagari script (Unicode range `\u0900-\u097F`), common digits, and essential punctuation marks (danda, question mark, etc.). This removes noise from other languages or special symbols.
3. **Tokenization**: We perform whitespace-based tokenization, which is appropriate for Hindi. Punctuation is treated as separate tokens or removed depending on the context.
4. **Quality Filtering**: Lines with fewer than 5 tokens are discarded as they lack sufficient context for training meaningful word embeddings.
5. **Deduplication**: Identical lines are removed to prevent the model from overfitting to repetitive text and to ensure a diverse training set.

3.3 Step 3: Word Vector Training

We train two custom word embedding models: Word2Vec (Skip-gram) and FastText. Both are trained with a dimensionality of 300 to match the comparison baseline.

The training process involves:

- **Model Initialization**: We utilize the Gensim library to initialize Word2Vec and FastText models.

- **Hyperparameters**: Both models are configured with a vector size of 300, a window size of 5 (looking at 5 words before and after the target), and a minimum count of 5 (ignoring rare words). We use the Skip-gram architecture as it generally performs better for semantic tasks.
- **Training Data**: To balance performance and training time on local hardware, we use a large filtered subset of the preprocessed corpus (approx. 400,000 lines).
- **Output**: The trained models are saved to disk along with their key vectors for efficient loading in subsequent steps.

3.4 Step 4: Downloading Pretrained Vectors

We download the Common Crawl Hindi fastText vectors (`cc.hi.300.vec`) to serve as a strong baseline for comparison. This allows us to evaluate the quality of our custom-trained vectors against a standard, large-scale open-source model. The script checks for the existence of the file and downloads/unzips it only if necessary.

3.5 Step 5: Vector Comparison

We compare our custom models (FastText, Word2Vec) against the pretrained Common Crawl vectors across several dimensions to assess their quality.

The comparison includes:

1. **Vocabulary Overlap**: We calculate the intersection of vocabularies between our custom models and the pretrained model to understand coverage.
2. **Word Similarity**: We compute cosine similarity for a fixed set of word pairs (e.g., "king"-"queen", "India"-"Delhi") to verify that semantic relationships are captured.
3. **Analogy Tests**: We perform standard analogy tests (e.g., "King" - "Man" + "Woman" = ?) to evaluate the models' ability to capture algebraic relationships in the vector space.
4. **Visualizations**:
    - **Word Similarity Plot**: Figure 1 shows the similarity scores of different models on standard pairs.
    - **t-SNE Projection**: We use t-SNE to reduce the 300-dimensional vectors to 2 dimensions for visualization (Figure 2), allowing us to see clusters of semantically related words.

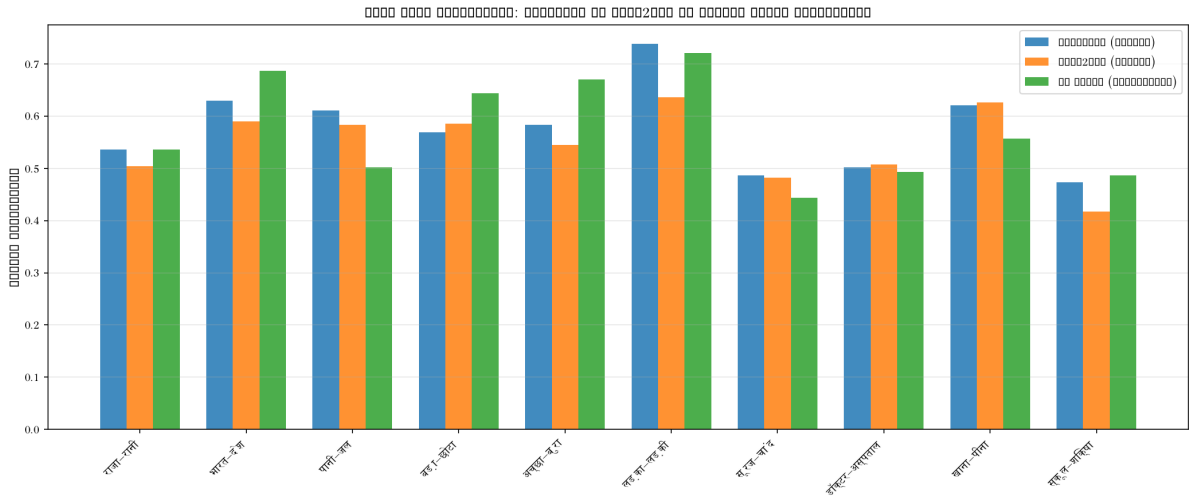

Figure 1: Word Similarity Comparison across Models

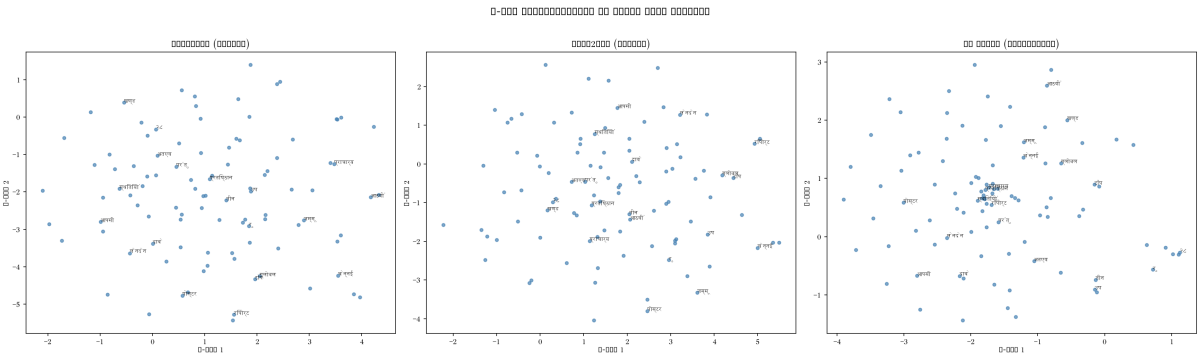

Figure 2: t-SNE Projection of Word Vectors

3.6 Step 6: Classification Data Preparation

We prepare the BBC Hindi news dataset for the LSTM model. This involves cleaning the text, label encoding the categories, and creating a stratified train/val/test split.

The preparation process includes:

- **Data Loading**: The script reads the raw CSV containing BBC Hindi articles and their categories.
- **Category Filtering**: We filter the dataset to include only valid categories: Khel (Sports), Bharat (India), Manoranjan (Entertainment), Vigyan (Science), Videsh (International), and Social.
- **Text Cleaning**: We apply the same cleaning pipeline as used for the corpus (removing HTML, non-Hindi chars) to ensure consistency.
- **Label Encoding**: Text labels are converted to integer IDs for the model.
- **Stratified Split**: We use 'StratifiedShuffleSplit' to split the data into training (70%), validation (15%), and test (15%) sets, ensuring that the class distribution is preserved in each split.

- **Vocabulary Building**: A vocabulary mapping (word to integer index) is built from the training data to be used by the embedding layer.

3.7 Step 7: LSTM Model Architecture

We design a Bidirectional LSTM model with an attention mechanism for text classification.
The model architecture consists of:

1. **Embedding Layer**: This layer converts integer word indices into dense 300-dimensional vectors. It can be initialized with our custom Word2Vec/FastText vectors, the pretrained Common Crawl vectors, or random weights.

2. **Bidirectional LSTM**: We use a 2-layer BiLSTM. Bidirectionality allows the model to capture context from both past and future words, which is crucial for understanding sentence semantics.

3. **Attention Mechanism**: An attention layer is added on top of the LSTM outputs. This computes a weighted sum of the hidden states, allowing the model to focus on the most relevant parts of the document for the classification task.

4. **Classifier**: The output of the attention layer is passed through a fully connected (Linear) layer followed by a LogSoftmax activation to produce class probabilities.

3.8 Step 8: Model Training

We train the LSTM model variants using the different embedding initializations.
Key aspects of the training loop:

- **Loss Function**: We use 'NLLLoss' (Negative Log Likelihood Loss) with class weights. The weights are inversely proportional to class frequencies to penalize errors on minority classes (like 'Social') more heavily, addressing class imbalance.

- **Optimizer**: The Adam optimizer is used for efficient gradient descent.

- **Training Loop**: The model is trained for a fixed number of epochs. In each epoch, we iterate through batches of data, compute gradients, and update weights.

- **Validation and Early Stopping**: After each epoch, we evaluate the model on the validation set. If the validation loss does not improve for a set number of epochs (patience), training is stopped early to prevent overfitting.

Figure 3 compares the training progression of the different models.

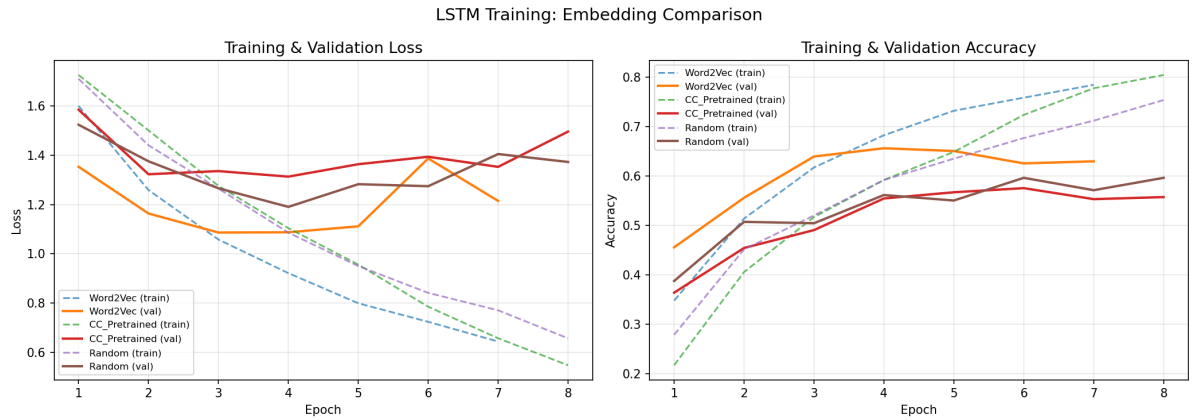

Figure 3: Training Curves Comparison

3.9 Step 9: Evaluation

Finally, we evaluate all model variants on the test set to verify their performance.

Evaluation involves:

- **Metrics**: We calculate overall Accuracy, Macro F1-score (to account for class imbalance), and weighted F1-score.
- **Confusion Matrices**: We generate confusion matrices for each model to visualize which classes are being confused. For example, similar topics like 'Bharat' and 'Social' might have higher confusion.
- **Comparison**: We aggregate the results to determine which embedding initialization strategy yields the best performance on this dataset.

Figures 4, 5, and 6 visualize these results.

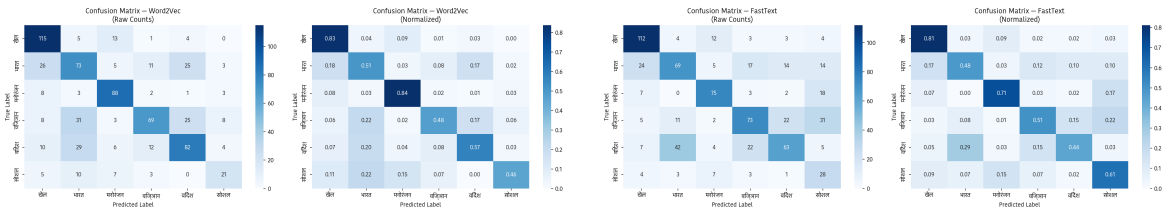

Figure 4: Confusion Matrices: Word2Vec (Left) vs FastText (Right)

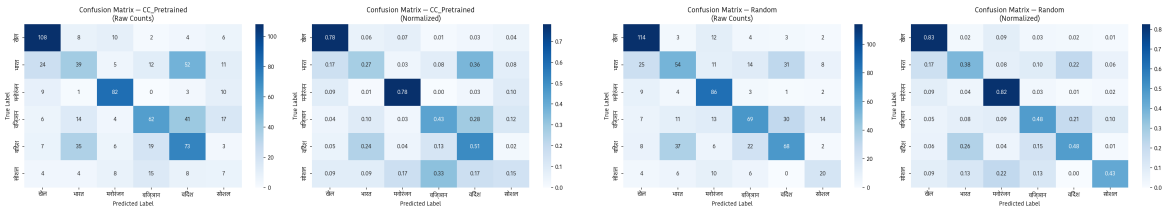

Figure 5: Confusion Matrices: CC Pretrained (Left) vs Random (Right)

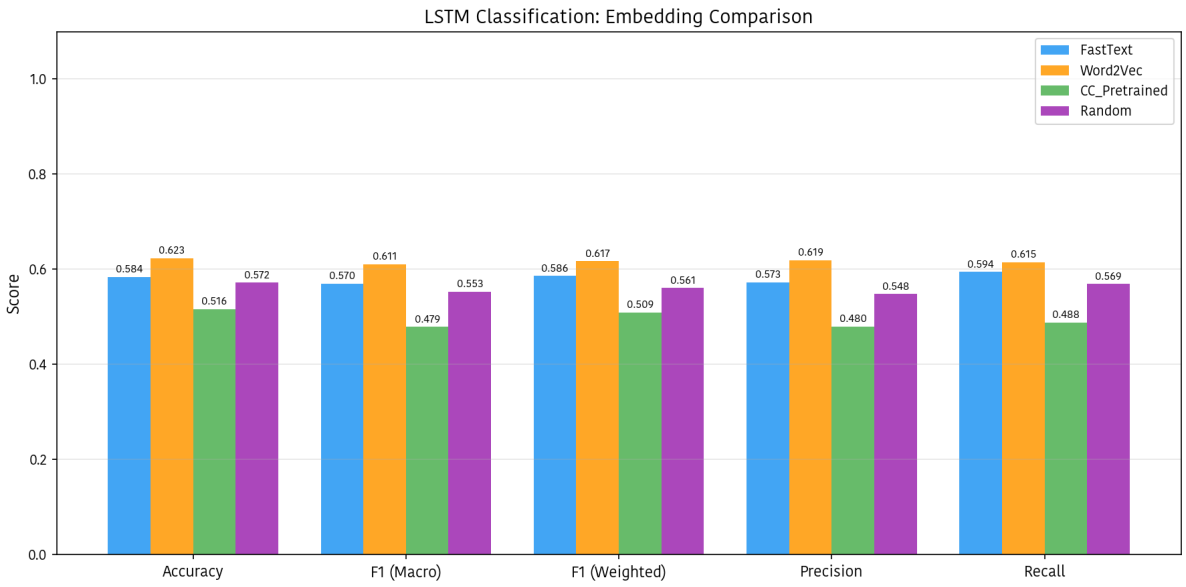

Figure 6: Overall Model Comparison (Accuracy and F1 Scores)

4 Methodology

4.1 Word Vector Training

- Algorithms: FastText and Word2Vec (gensim).
- Hyperparameters: dimension 300, window 5, min_count 5, skip-gram, 5 epochs.
- Custom vocabulary size after training: 90,211 for both models.

4.2 LSTM Classification

- Architecture: Embedding -¿ BiLSTM (2 layers, hidden 128, dropout 0.3) -¿ Attention -¿ MLP classifier.
- Max sequence length: 256 tokens.
- Batch size: 64, learning rate: 1×10^{-3}, early stopping enabled.
- Class imbalance handling: weighted cross-entropy.
- Embedding sources compared:
  - Custom FastText
  - Custom Word2Vec
  - Common Crawl pretrained vectors
  - Random initialization (baseline)

5 Results

5.1 Word Vector Comparison

Additional overlap results:

Model	Vocab Size	Avg Pair Similarity	Analogy Hits (out of 5)
FastText (custom)	90,211	0.5749	1
Word2Vec (custom)	90,211	0.5477	1
CC Hindi (pretrained)	400,000 (loaded)	0.5738	2

Table 1: Word vector summary from `outputs/word_vectors/comparison_report.json`.

- Custom FastText ∩ Custom Word2Vec: 90,211.
- Custom FastText ∩ CC Hindi: 76,759.
- Custom Word2Vec ∩ CC Hindi: 76,759.

5.2 LSTM Classification Comparison

Embedding Type	Accuracy	Macro F1	Weighted F1
Word2Vec	0.6231	0.6107	0.6167
FastText	0.5841	0.5697	0.5859
Random	0.5716	0.5532	0.5608
CC Pretrained	0.5160	0.4789	0.5089

Table 2: Classification metrics from `outputs/classification/evaluation_report.json`.

Best model: **Word2Vec embeddings** (Accuracy 62.31%, Macro F1 0.6107).

5.3 Per-Class Behavior (Best Model: Word2Vec)

- Strong classes: Manoranjan (F1 0.775), Khel (F1 0.742).
- Mid classes: Videsh (F1 0.586), Vigyan (F1 0.570).
- Weak classes: Bharat (F1 0.497), Social (F1 0.494).

6 Difficulties Encountered

1. **Data source changes**: Original IndicCorp route was deprecated; pipeline had to be adapted to Sangraha.
2. **HuggingFace API compatibility**: Older call patterns failed; loader needed updates and fallback safety.
3. **Compute constraints**: Full 300d training on all cleaned lines was too slow for laptop runtime; a filtered large subset (400k lines) was used.
4. **Large pretrained files**: Common Crawl vectors are large (multi-GB), so loading all vectors at once was not practical.
5. **Rendering issues**: Devanagari plotting fonts produced warnings in some matplotlib environments.

7 Why the Model Performance Is Not Higher

The classifier works but does not reach very high performance. Main reasons:

1. **Class imbalance**: Social class is much smaller (only 46 test samples), which hurts stable learning and macro F1.
2. **Very long documents**: Average article length is around 1,040 tokens, but model input is truncated to 256 tokens, losing context.
3. **Simple preprocessing/tokenization**: Whitespace tokenization for Hindi keeps many spelling/morphology variants separate.
4. **Domain/style mismatch**: Common Crawl vectors are broad web text, not specifically tuned to BBC article style.
5. **Model capacity limits**: A BiLSTM with static embeddings is weaker than modern transformer approaches for long, nuanced text.

8 Conclusion

The assignment requirements were implemented with the updated instructor constraints:

- Sangraha corpus used for custom Hindi vector training.
- Common Crawl Hindi vectors used for required comparison.
- LSTM-based classification completed with multi-embedding comparison and evaluation artifacts.

Current best result is the Word2Vec-initialized LSTM. The main path to better performance is handling long-document context and class imbalance more effectively (for example, larger max length with hierarchical modeling, improved Hindi normalization, and transformer-based fine-tuning).