Urdu Story Generator

NLP Assignment 1 - Complete Project Documentation

Course: CS-462 Natural Language Processing

Semester: Spring 2026 - Semester 6

Repository: A1 - Urdu Story Generator

Date: February 20,2026

Submitted By

Muhammad Rehan Tariq	23I-0034
Abdur Rafay	23I-0117
Abuzar Mehdi	23I-0125

Table of Contents

1. Project Overview

The Urdu Story Generator is a full-stack Natural Language Processing (NLP) project built from the ground up without relying on any pre-built language models or NLP libraries. It scrapes Urdu moral stories from the web, preprocesses the raw text, trains a custom Byte-Pair Encoding (BPE) tokenizer, builds a trigram language model with interpolated smoothing, exposes the model through a FastAPI microservice, and presents a chat-style React user interface — all deployable via Docker Compose.

Key highlights:

- End-to-end pipeline: from web scraping to model deployment
- Custom BPE tokenizer trained on Urdu corpus (no external tokenizer library)
- Trigram language model with MLE + linear interpolation smoothing
- Production-ready FastAPI backend with Pydantic validation
- Chat-style React frontend with dark/light theme, session persistence, and word-streaming
- Docker Compose for one-command local deployment
- GitHub Actions CI/CD with automated testing and GHCR image publishing

2. Repository Structure

The repository is organized into logically separated directories, each responsible for a distinct stage of the pipeline:

```
A1/
├── Scraping/                    # Phase I  - web scraping
│   ├── urdupoint.py             #   scraper script
│   ├── Stories_Urls.csv         #   cached story URLs
│   └── Documents/               #   raw .txt story files
├── PreProcessing/               # Phase II - text preprocessing
│   ├── preprocessing.py         #   cleaning & tokenization
│   └── Preprocessed_documents/  #   cleaned .txt files
├── Tokenization/                # Phase III - BPE tokenizer
│   ├── BPE.py                   #   BPE training script
│   ├── vocab.json               #   learned vocabulary
│   ├── merges.txt               #   merge rules
│   └── encoded_dataset.txt      #   encoded corpus
├── models/                      # Phase IV - language model
│   ├── trigram_model.py         #   trigram model + BPE integration
│   ├── model.ipynb              #   training notebook
│   └── trigram_model.pkl        #   serialized model (generated)
├── backend/                     # Phase V  - FastAPI microservice
│   ├── app.py                   #   API endpoints
│   ├── asgi.py                  #   ASGI entry-point
```

```
│   └── requirements.txt
├── frontend/                    # Phase VI - React UI
│   ├── src/App.js               #   main React component
│   ├── Dockerfile               #   multi-stage build
│   └── nginx.conf               #   nginx config
├── tests/                       # Automated tests
│   └── test_api.py
├── .github/workflows/ci.yml     # CI/CD pipeline
├── Dockerfile                   # Backend container
├── docker-compose.yml           # Full-stack orchestration
└── requirements.txt             # Top-level deps
```

3. Pipeline Architecture

The project follows a sequential six-phase pipeline. Each phase produces artifacts consumed by the next phase:

Phase	Name	Input	Output	Script
I	Web Scraping	UrduPoint website URLs	Raw .txt story files	Scraping/urdupoint.py
II	Pre-Processing	Raw .txt stories	Cleaned .txt with special tokens	PreProcessing/preprocessing.py
III	BPE Tokenization	Preprocessed corpus	vocab.json, merges.txt	Tokenization/BPE.py
IV	Language Model	Preprocessed corpus + BPE data	trigram_model.pkl	models/trigram_model.py
V	REST API	trigram_model.pkl	JSON API (port 5000)	backend/app.py
VI	React Frontend	FastAPI endpoints	Web UI (port 3000)	frontend/src/App.js

The pipeline can also be triggered on-the-fly: the backend will train the trigram model from preprocessed documents at startup if no serialized model file is found.

4. Phase I - Data Scraping

4.1 Objective

Collect a large corpus of Urdu moral stories from https://www.urdupoint.com/kids/category/moral-stories to serve as training data for the language model.

4.2 Script: Scraping/urdupoint.py

The scraper is built using the Botasaurus headless browser framework and Pandas.

4.2.1 get_Story_Url()

Navigates to the UrduPoint moral stories category page and iterates through 40 pages, collecting href attributes from all <a class='sharp_box'> anchor tags. All collected URLs are saved to Scraping/Stories_Urls.csv.

```
driver.get('https://www.urdupoint.com/kids/category/moral-stories-page1.html')
for i in range(40):
    stories_anchor_tags = driver.select_all('a.sharp_box')
    urls = [i.get_attribute('href') for i in stories_anchor_tags]
    story_url.extend(urls)
    next_page_button.click()
```

4.2.2 Scrape_Data()

Reads the cached CSV, then visits each story URL and extracts the main text content from the element. Each story is saved as doc<N>.txt in Scraping/Documents/. The scraper resumes from document 201 onward, preserving previously scraped data.

4.3 Output

- Scraping/Stories_Urls.csv — all collected story URLs
- Scraping/Documents/doc<N>.txt — individual Urdu story files (approx. 400+ stories)

4.4 Configuration

Parameter	Value	Description
Headless mode	True	Runs Chrome without GUI
Pages scraped	40	Number of category pages iterated
Start document	201	Resume index for incremental scraping
CSS selector (URL)	a.sharp_box	Anchor tag for story links
CSS selector (text)	div.txt_detail	Story body container

5. Phase II - Text Pre-Processing

5.1 Objective

Clean raw scraped Urdu text by removing noise (author names, English characters, digits, diacritics) and annotate sentence, paragraph, and story boundaries with special tokens.

5.2 Script: PreProcessing/preprocessing.py

Each raw document is passed through a sequential pipeline of cleaning functions:

5.2.1 remove_writer_name(text)

Removes the first non-empty line of each document, which contains the author/writer name.

5.2.2 normalize_unicode(text)

Applies NFC Unicode normalization to standardize Urdu character forms (e.g., composed vs. decomposed Arabic letters).

5.2.3 remove_unwanted_chars(text)

Removes all characters outside the Urdu Unicode range U+0600–U+06FF plus allowed punctuation (۔ ، ؟ ! and whitespace). This strips English text, Latin digits, and miscellaneous symbols.

```
urdu_range = r"\u0600-\u06FF"
allowed_pattern = rf"[^{urdu_range}\s؟،.!]+"
text = re.sub(allowed_pattern, '', text)
```

5.2.4 normalize_spaces(text)

Collapses multiple spaces and tabs to a single space while preserving line-break structure for paragraph detection.

5.2.5 add_special_tokens(text)

Inserts boundary tokens:

Token	Meaning	Placement
<EOS>	End of Sentence	After every sentence-ending punctuation: ؟ ۔ !
<EOP>	End of Paragraph	After every blank-line-separated paragraph
<EOT>	End of Text/Story	At the very end of each document

```
# Example output snippet
ایک دن ایک بچہ جنگل میں گیا ۔ <EOS> ایک بوڑھا درخت ملا ۔ وہاں اسے <EOS> <EOP>
اس نے درخت سے کچھ سیکھا ۔ <EOS> <EOP> <EOT>
```

5.3 Output

- PreProcessing/Preprocessed_documents/doc<N>.txt — one cleaned file per story

6. Phase III - BPE Tokenization

6.1 Objective

Train a Byte-Pair Encoding (BPE) tokenizer from scratch on the preprocessed Urdu corpus to learn a subword vocabulary of size 250. BPE handles the morphological richness of Urdu by representing rare words as combinations of frequently occurring subword units.

6.2 Algorithm (Tokenization/BPE.py)

BPE training follows the standard iterative merge procedure:

1. Initialize the vocabulary with all individual Unicode characters found in the corpus.
2. Compute pair frequencies: count adjacent symbol pairs across all words, weighted by word frequency.

3. Select the most frequent pair.
4. Merge that pair into a new symbol throughout the corpus.
5. Add the new symbol to the vocabulary.
6. Repeat steps 2–5 until the vocabulary reaches the target size (250).

Special tokens (<EOS>, <EOP>, <EOT>) are never split or merged — they are kept intact as atomic units throughout training and inference.

6.3 Key Functions

Function	Purpose
tokenize_word(word)	Converts a word to a tuple of characters; special tokens return as a single-element tuple
split_special_tokens(token)	Handles edge cases where a special token is concatenated with adjacent text
get_word_freqs(corpus)	Builds a Counter of word-tuple → frequency from the full corpus
get_pair_counts(word_freqs)	Counts adjacent symbol pairs (skipping special tokens)
merge_pair(pair, word_freqs)	Applies a merge rule to all words in the corpus
train_bpe(vocab_size=250)	Orchestrates the full training loop
save_results(vocab, merges)	Persists vocab.json and merges.txt
save_encoded_dataset(word_freqs)	Saves final subword encodings to encoded_dataset.txt

6.4 Output Files

File	Description
Tokenization/vocab.json	JSON array of all vocabulary tokens (characters + merged subwords)
Tokenization/merges.txt	Ordered merge rules — one rule per line in the format 'a b'
Tokenization/encoded_dataset.txt	Each unique word represented as its final subword sequence, with frequency count

7. Phase IV - Trigram Language Model

7.1 Objective

Build a statistical n-gram language model (trigram) using Maximum Likelihood Estimation (MLE) with linear interpolation smoothing, operating on BPE subword tokens.

7.2 Classes (models/trigram_model.py)

7.2.1 BPETokenizer

Loads the pre-trained vocab.json and merges.txt from Phase III and provides tokenize() and detokenize() methods.

- tokenize(text): splits text by whitespace, applies BPE merges to each word, and marks word boundaries with the ▁ prefix on first subword tokens.

- detokenize(tokens): reconstructs readable text from subword tokens using the ▁ boundary marks and special token spacing.

7.2.2 TrigramLanguageModel

Core statistical model. Stores:

- unigram_counts — token → frequency
- bigram_counts — context token → {next token → frequency}
- trigram_counts — (context1, context2) → {next token → frequency}
- is_trained flag and a reference to the BPETokenizer used during training

Interpolated probability formula:

$$P(w \mid w_{-2}, w_{-1}) = \lambda_1 \cdot P(w) + \lambda_2 \cdot P(w \mid w_{-1}) + \lambda_3 \cdot P(w \mid w_{-2}, w_{-1})$$

Parameter	Default	Description
λ_1 (lambda1)	0.1	Unigram weight
λ_2 (lambda2)	0.3	Bigram weight
λ_3 (lambda3)	0.6	Trigram weight

Note: $\lambda_1 + \lambda_2 + \lambda_3 = 1.0$ (enforced by assertion at construction time).

7.2.3 UrduStoryGenerator

Wraps TrigramLanguageModel and implements the generate() method, which auto-regressively samples the next subword token until max_length tokens are produced. Temperature scaling controls randomness: lower temperature → more deterministic, higher → more creative.

7.2.4 StoryGeneratorAPI

High-level API class used by the backend. It loads a serialized model from a .pkl file and exposes a generate(prefix, max_length, temperature) method returning a dict with keys success, story, and error.

7.3 Training

The model trains on the full preprocessed corpus in a single pass:

7. Each document is tokenized with BPETokenizer.
8. Padded with two <START> tokens at the beginning.
9. Unigram, bigram, and trigram counts are accumulated.
10. The serialized model is saved to models/trigram_model.pkl with pickle.

8. Phase V - REST API (FastAPI Backend)

8.1 Objective

Expose the trained trigram language model as a RESTful HTTP service, making it accessible to the frontend and any external client.

8.2 File: backend/app.py

Built using FastAPI with Pydantic for request/response validation.

8.2.1 Model Loading Strategy

At startup the backend calls ensure_model():

11. If models/trigram_model.pkl exists, load it directly.
12. Otherwise, scan PreProcessing/Preprocessed_documents/, load all .txt files, train a TrigramLanguageModel, serialize it, and save the .pkl.

This ensures the API is always ready, even in a fresh Docker container with no pre-built model.

8.2.2 CORS Middleware

CORS is configured to allow all origins (*), methods, and headers, enabling the React frontend and Render-hosted deployment to communicate without restrictions.

8.3 Pydantic Schemas

Schema	Fields	Constraints
GenerateRequest	prefix (str), max_length (int), temperature (float)	max_length: 1–5000; temperature: 0.1–2.0
GenerateResponse	success (bool), story (str|None), prefix (str), error (str|None)	—
ModelInfoResponse	model_type, vocabulary_size, total_tokens, interpolation_weights, is_trained	—

8.4 API Endpoints

Method	Path	Description	Response
GET	/	Root — redirects browsers to /docs	{"status": "ok", "message": "..."}
GET	/health	Health check for load balancers	{"status": "ok", "message": "Backend is running"}
POST	/generate	Generate an Urdu story	GenerateResponse JSON
GET	/model-info	Model metadata and statistics	ModelInfoResponse JSON
GET	/docs	Auto-generated Swagger UI	Interactive HTML documentation

8.5 Sample Request / Response

POST /generate

```
{
  "prefix": "ایک دن",
  "max_length": 300,
```

```
  "temperature": 0.8
}
```

Response:

```
{
  "success": true,
  "story": "ایک دن ایک بچہ جنگل میں گیا <EOS> وہاں اسے...",
  "prefix": "ایک دن",
  "error": null
}
```

8.6 Running the Backend

```
# From the repository root:
uvicorn backend.asgi:app --host 0.0.0.0 --port 5000 --reload

# Or directly:
python backend/app.py
```

9. Phase VI - React Frontend

10. Containerization with Docker

10.1 Backend Dockerfile

Located at the repository root. Key steps:

```
FROM python:3.11-slim
ENV PYTHONDONTWRITEBYTECODE=1 PYTHONUNBUFFERED=1
WORKDIR /app
COPY backend/requirements.txt ./requirements.txt
RUN pip install --no-cache-dir -r requirements.txt
COPY models/                        ./models/
COPY backend/                       ./backend/
COPY Tokenization/                  ./Tokenization/
COPY PreProcessing/Preprocessed_documents/
./PreProcessing/Preprocessed_documents/
EXPOSE 5000
CMD ["uvicorn", "backend.asgi:app", "--host", "0.0.0.0", "--port", "5000"]
```

10.2 Frontend Dockerfile

Located at frontend/Dockerfile. Multi-stage build (described in Section 9.4).

10.3 docker-compose.yml

Orchestrates both services with a single command:

```
services:
  backend:
```

```
    build: .
    ports: ['5000:5000']
    volumes:
      - ./models:/app/models              # persist trained model
      - ./PreProcessing:/app/PreProcessing
      - ./Tokenization:/app/Tokenization

  frontend:
    build: { context: ./frontend, dockerfile: Dockerfile }
    ports: ['3000:80']
    depends_on: [backend]
    environment:
      - REACT_APP_API_BASE_URL=http://localhost:5000
```

```
# Launch entire stack:
docker compose up --build
```

11. CI/CD Pipeline (GitHub Actions)

11.1 Workflow Overview

File: .github/workflows/ci.yml — triggered on every push or pull request to the main branch.

Job	Trigger	Steps
test	push / PR to main	1. Checkout code 2. Set up Python 3.11 3. pip install backend/requirements.txt + pytest + httpx 4. pytest tests/ -v
docker	push to main only (after test passes)	1. Checkout code 2. Log in to GitHub Container Registry (GHCR) 3. docker build -t urdu-story-generator:<sha> 4. Tag as :latest and :<sha> 5. Push both tags to ghcr.io

11.2 Docker Image Registry

The backend Docker image is pushed to GitHub Container Registry (ghcr.io) under the repository owner's namespace. Images are tagged with both the commit SHA and latest, enabling reproducible deployments and rollback.

11.3 Secrets Used

Secret	Description
GITHUB_TOKEN (auto)	Automatically provided by GitHub Actions; used to authenticate with GHCR

12. API Reference

GET /health

Field	Value
Method	GET
Path	/health
Description	Health check; returns 200 when service is up
Response body	{"status": "ok", "message": "Backend is running"}

POST /generate

Field	Value
Method	POST
Path	/generate
Content-Type	application/json
Request: prefix	string — optional Urdu starting phrase (default: empty)
Request: max_length	integer 1–5000 — maximum tokens to generate (default: 500)
Request: temperature	float 0.1–2.0 — sampling temperature (default: 0.8)
Response: success	boolean
Response: story	string — the generated Urdu story
Response: prefix	string — echoes back the input prefix
Response: error	string|null — error message if success is false
Error 422	Pydantic validation failure (out-of-range parameters)
Error 500	Internal model error

GET /model-info

Field	Value
Method	GET
Path	/model-info
Response: model_type	string — 'Trigram Language Model (MLE + Interpolation)'
Response: vocabulary_size	integer — total BPE vocabulary tokens known to model
Response: total_tokens	integer — total training tokens seen
Response: interpolation_weights	dict {lambda1_unigram, lambda2_bigram, lambda3_trigram}
Response: is_trained	boolean

13. Testing

13.1 Test File: tests/test_api.py

Uses pytest with FastAPI TestClient (via httpx) for in-process integration testing.

Test Function	What It Validates
test_health()	GET /health returns 200 with status: ok
test_root()	GET / returns 200 with a status field
test_generate_no_prefix()	POST /generate with empty prefix returns a non-empty story string

test_generate_with_prefix()	POST /generate with Urdu prefix correctly echoes prefix in response
test_generate_validation()	POST /generate with max_length=9999 (>5000) returns HTTP 422
test_model_info()	GET /model-info returns vocabulary_size and is_trained fields

13.2 Running Tests

```
# From repository root:
pytest tests/ -v

# Expected output (all passing):
tests/test_api.py::test_health                  PASSED
tests/test_api.py::test_root                    PASSED
tests/test_api.py::test_generate_no_prefix      PASSED
tests/test_api.py::test_generate_with_prefix    PASSED
tests/test_api.py::test_generate_validation     PASSED
tests/test_api.py::test_model_info              PASSED
6 passed in X.XXs
```

14. Setup & Running the Project

14.1 Prerequisites

- Python 3.8 or newer
- Node.js 18+ and npm (for frontend development only)
- Docker Desktop (for containerized deployment)
- Git

14.2 Full Manual Setup (Step by Step)

Step 1 — Clone and create virtual environment

```
git clone <repository-url>
cd A1
python -m venv A1
.\A1\Scripts\Activate.ps1        # Windows (PowerShell)
# source A1/bin/activate           # macOS / Linux
pip install -r requirements.txt
```

Step 2 — Scrape data (optional — skip if Documents/ already contains files)

```
python Scraping/urdupoint.py
```

Step 3 — Preprocess the corpus

```
python PreProcessing/preprocessing.py
```

Step 4 — Train the BPE tokenizer

```
python Tokenization/BPE.py
```

Step 5 — Train the language model (optional — backend trains automatically if .pkl is absent)

```
# Open and run models/model.ipynb in Jupyter, or:
# The backend will auto-train on first startup.
```

Step 6 — Start the backend

```
uvicorn backend.asgi:app --host 0.0.0.0 --port 5000 --reload
```

Step 7 — Start the frontend

```
cd frontend
npm install --legacy-peer-deps
npm start
# Open http://localhost:3000
```

14.3 Docker Compose (Recommended)

```
# From the repository root:
docker compose up --build

# Backend:  http://localhost:5000
# Frontend: http://localhost:3000
# Swagger:  http://localhost:5000/docs
```

14.4 Live Deployment

Service	Platform	URL
Backend API	Render (Docker)	https://urdu-story-generator.onrender.com
Frontend	Vercel	Deployed via Vercel (auto-deploy on git push)
Docker Image	GHCR	ghcr.io/<owner>/a1:latest

15. Technologies & Dependencies

15.1 Backend Python Packages

Package	Purpose
fastapi	Async web framework for REST API
uvicorn	ASGI server to run FastAPI
pydantic	Data validation and serialization
python-multipart	Form data support (Pydantic extension)
httpx	Async HTTP client (used by pytest TestClient)
pytest	Test runner

15.2 Scraping Packages

Package	Purpose
botasaurus	Headless browser automation framework

pandas	CSV reading/writing for URL cache

15.3 Frontend Packages

Package	Purpose
react@18	UI component library
react-scripts	CRA build tooling
nginx (Docker)	Static file server for production build

15.4 Infrastructure

Tool	Purpose
Docker / Docker Compose	Container build and orchestration
GitHub Actions	CI/CD — automated testing and image publishing
GitHub Container Registry (GHCR)	Docker image storage
Render	Cloud deployment for the backend API
Vercel	Cloud deployment for the React frontend