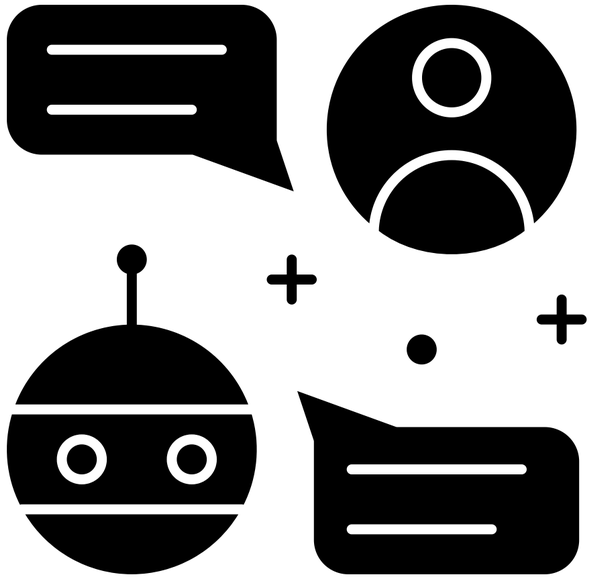
Course: Build a Small Language Model from scratch

Module 2: Data Preparation Pipeline



Instructor: Nusrat Jahan Lia

Learning Objectives

- Download and manage large datasets from HuggingFace Hub
- Preprocess text data for language model training
- Implement tokenization for Bangla text
- Create train/validation splits
- Convert text data to binary format for efficient training

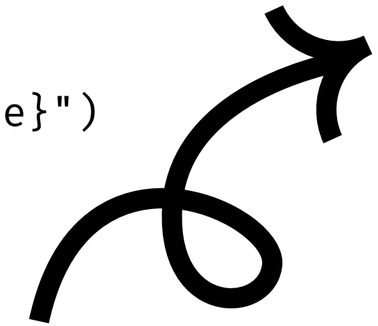
Data Collection and Loading

Complete Code Walkthrough

```
import os
import pandas as pd
from huggingface_hub import login, list_repo_files, hf_hub_download from google.colab
import userdata

# Authentication
hf_token = userdata.get('HF_TOKEN')
login(token=hf_token)

# Dataset configuration
repo_id = "ashtrayAI/Bangla_Financial_news_articles_Dataset"
folder = "Bangla_fin_news_articles"

# Step 1: List all files

files = list_repo_files(repo_id=repo_id, repo_type="dataset", token=hf_token)
csv_files = [f for f in files if f.startswith(folder + "/") and f.endswith(".csv")]
print(f"Found {len(csv_files)} CSV files.")
```

Data Collection and Loading

```
# Step 2: Download and load
dfs = []
for file in csv_files:
    try:
        local_path = hf_hub_download(
            repo_id=repo_id,
            filename=file,
            repo_type="dataset",
            token=hf_token
        )
        df = pd.read_csv(local_path)
        dfs.append(df)
    except Exception as e:
        print(f"Error processing {file}: {e}")
        continue

# Step 3: Combine
full_df = pd.concat(dfs, ignore_index=True)
print("Combined DataFrame shape:", full_df.shape)
```



Why ignore_index=True?

- Resets index to avoid conflicts
- Creates continuous numbering
- Prevents duplicate indices

Text Preprocessing

Raw Text vs. Model-Ready Text

Raw Text Issues:

- Inconsistent formatting
- Missing values
- Mixed data types
- Irregular column names

Model Requirements:

- Clean, consistent text
- Proper data types
- Standardized format
- No missing values

Column Selection and Cleaning

Identifying Text Columns:

```
# Method 1: Check data types
text_columns = full_df.select_dtypes(include=['object']).columns
print("Text columns:", text_columns)

# Method 2: Manual inspection
print(full_df.columns.tolist())

# Method 3: Automated detection
text_col = next((c for c in full_df.columns if full_df[c].dtype == "object"), None)
```

Renaming Columns for Consistency:

```
# In our case, we use 'News' column
text_col = 'News'
assert text_col in full_df.columns, f"Column '{text_col}' not found!"
print(f"Using column '{text_col}' as text.")

# Rename for standardization
df_clean = full_df[[text_col]].rename(columns={text_col: "text"})
```

Train vs. Validation Split Concept

Why Split Data?

- Training Set: Used to train the model. The model **"learns"** patterns from this data.
- Validation Set: This separate subset is used during training to evaluate the model's performance on **unseen data**. It helps assess how well the model generalizes beyond the training examples.
- Prevents **Overfitting**: Model learns to generalize, not memorize

Underfitting, Overfitting & Just-Right Fitting

Underfitting: The Lazy Learner

"Cats? Dogs? Same thing, they all have four legs!"

- **Definition:** The model is too simple to capture the underlying pattern in the data.
- **Analogy:** Like a student who just skims the textbook and makes wild guesses.
- **Model Behavior:** Performs poorly on both training and validation data.

Why it happens:

- Model is too simple (for example, linear regression for non-linear data).
- Not trained long enough.
- Too few features or too much regularization.

Example:

- You try to predict house prices using only one feature: square footage. But you ignore location, number of bedrooms, etc.
- Result? The model guesses poorly : it underfits.

Overfitting: The Overenthusiastic Memorizer

"This is Whiskers, a Maine Coon with a birthmark on his left paw who likes tuna at 8 AM."

- **Definition:** The model learns too much, even the noise in the data.
- **Analogy:** Like a student who memorizes the practice test answers without understanding concepts.
- **Model Behavior:** Great on training data, terrible on unseen data.

Why it happens:

- Model is too complex (for example, deep neural network on a small dataset).
- Too many features.
- Trained too long without validation checks.

Example:

- You train a model to detect spam emails. It memorizes specific spam phrases but fails to catch new types of spam.
- The result? Looks smart, but it crumbles when facing real-world data.

Appropriate Fitting: The Balanced Learner

"Most cats have pointy ears, whiskers, and meow. I can recognize different breeds even if they don’t look exactly like the ones I studied."

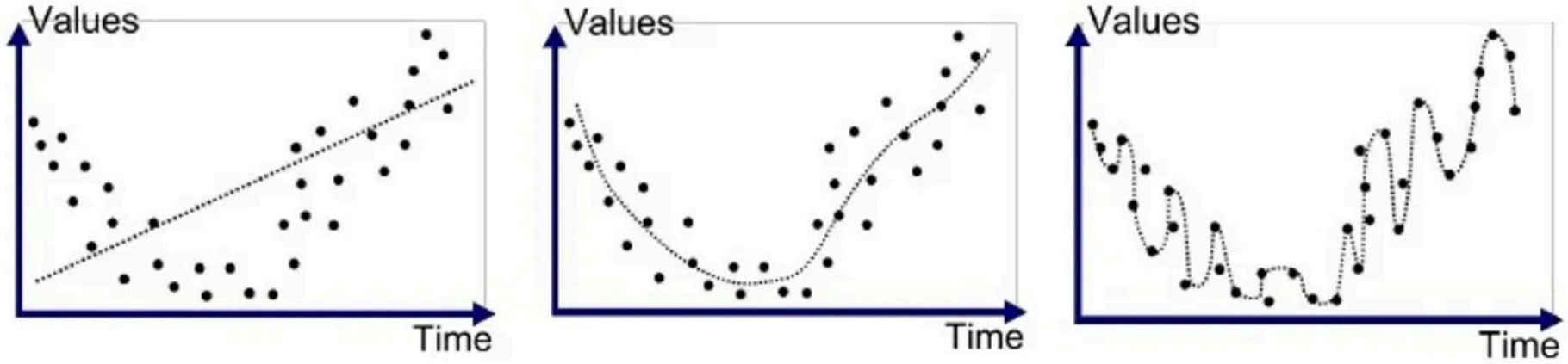
- **Definition:** The model captures the general patterns without memorizing noise.
- **Analogy:** Like a student who understands the subject deeply and can solve new problems using logic.
- **Model Behavior:** Good performance on both training and validation data.

Achieved by:

- Choosing the right model complexity.
- Using proper train/validation splits.
- Techniques like regularization, early stopping, dropout.

Example:

- Netflix’s recommendation system learns your general taste (e.g., you like witty comedies with strong female leads), not just the exact shows you watched.
- This model generalizes well across millions of users.



Underfitted

Good Fit/Robust

Overfitted

source: https://medium.com/greyatom/what-is-underfitting-and-overfitting-in-machine-learning-and-how-to-deal-with-it-6803a989c76

Now, Let’s split

```
TEST_SPLIT = 0.05  # 5% for validation

# Reasoning:
# - Large dataset (7,695 files) = enough validation data
# - More training data = better model performance
# - 5% still provides reliable evaluation metrics
```

```
from datasets import Dataset, DatasetDict

# Convert to HuggingFace Dataset
ds_full = Dataset.from_pandas(df_clean)

# Shuffle for randomness
ds_full = ds_full.shuffle(seed=42)

# Split the dataset
splits = ds_full.train_test_split(test_size=TEST_SPLIT, seed=42)
ds = DatasetDict({
    "train": splits["train"],
    "validation": splits["test"]
})

print(ds)
```

Why Use Random Seeds?

- Ensures reproducible results
- Same split every time you run the code
- Important for comparing model versions

Tokenization Fundamentals

What is **Tokenization?**

Converting Text to Numbers

The Problem:

- Neural networks only understand numbers. Why? [we will learn in the next modules]
- Text is made of characters and words
- Need a bridge between text and numbers

The Solution:

- Tokenization converts text into numerical tokens
- Each token represents a piece of text
- Tokens can be words, subwords, or characters

Word-level Tokenization:

```
text = "আমি বাংলায় কথা বলি"
tokens = text.split()
# ['আমি', 'বাংলায়', 'কথা', 'বলি']
```

Problems with Word-level:

- Huge vocabulary size
- Out-of-vocabulary (OOV) words
- Poor handling of rare words

Subword-level Tokenization:

```
# BanglaT5 tokenizer uses SentencePiece
# "বাংলাদেশ" might become ["বাংলা", "দেশ"]
# Handles rare words better
```

Bangla Tokenization Challenges

- Complex script with conjuncts
- Vowel diacritics (মাত্রা)
- Multiple forms of same character
- Contextual character shaping

"ক্ষ" = "ক" + "্" + "ষ" (conjunct)
"কি" = "ক" + "ি" (vowel diacritic)

Subword Tokenization Benefits

- Handles unseen words
- Smaller vocabulary size
- Better performance on morphologically rich languages
- Reduces OOV(out-of-vocab) issues

What is SentencePiece?

- Unsupervised tokenization algorithm
- Language-agnostic
- Handles any text without pre-tokenization
- Used by many modern language models

Using BanglaT5 Tokenizer

```
from transformers import AutoTokenizer

TOKENIZER = "csebuetnlp/banglat5"
tokenizer = AutoTokenizer.from_pretrained(TOKENIZER)

print(f"Vocabulary size: {tokenizer.vocab_size}")
print(f"Model max length: {tokenizer.model_max_length}")
```

 # What happens behind the scenes:
1. Downloads tokenizer files from HuggingFace Hub
2. Loads vocabulary and merge rules
3. Initializes tokenizer with proper configuration
4. Ready to tokenize text

Understanding Token IDs and Vocabulary Size

```
# Example tokenization
text = "বাংলাদেশ একটি সুন্দর দেশ"
tokens = tokenizer.tokenize(text)
token_ids = tokenizer.convert_tokens_to_ids(tokens)

print(f"Original text: {text}")
print(f"Tokens: {tokens}")
print(f"Token IDs: {token_ids}")

# Reverse process
decoded = tokenizer.decode(token_ids)
print(f"Decoded: {decoded}")
```

Tokenization Parameters

```
def tokenize_function(batch):
    texts = [str(x) if x is not None else "" for x in batch["text"]]

    out = tokenizer(
        texts,
        add_special_tokens=False,    # Don't add [CLS], [SEP] tokens,We can handle special tokens manually
        truncation=True,             # Cut text if too long
        max_length=512               # Maximum sequence length based on model
    )

    return {
        "ids": out["input_ids"],
        "len": [len(x) for x in out["input_ids"]],
    }
```

Special token?

- <s> = Start of Sentence → “The opening act of your text concert.”
- </s> = End of Sentence → “The mic drop moment.”
- <pad> = Padding → “The invisible seat filler in the token stadium.”
- <unk> = Unknown token → “The mysterious stranger at the party.”
- <mask> = Mask token → “The hidden ninja, waiting for you to guess what it is.”

- **[CLS]** → carries the overall meaning for downstream tasks.
- **[SEP]** = Separator / Boundary Marker: Stands for “separate sentences or segments

[CLS] Sentence A [SEP] Sentence B [SEP]

The [CLS] / <s> and [SEP]/ </s> confusion

- **[CLS]** ➜ mainly for BERT-like encoders, used in classification or embeddings.
- **<s>** ➜ mainly for decoder-style or GPT-like models, marks start of text.

[SEP] = “I’m a separator; I split things into chunks.”

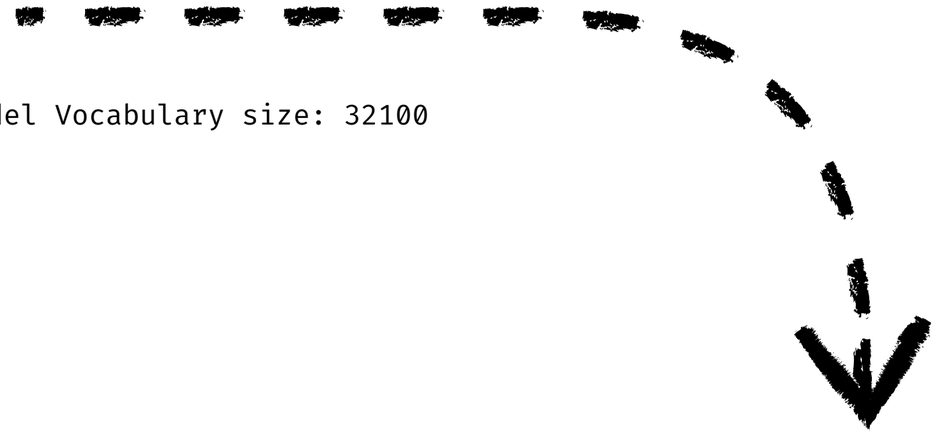
</s> = “This sequence is done; stop generating.”

Back to Applying Tokenization to Dataset

```
# Apply tokenization to entire dataset, ds is the dataset split we made earlier
tokenized_ds = ds.map(
    tokenize_function,
    batched=True,                    # Process in batches for efficiency
    num_proc=4,                      # Use 4 CPU cores
    remove_columns=["text"],         # Remove original text column to keep tokens only
    desc="Tokenizing",               # Progress bar description
)

print(tokenized_ds)
```

Saving Tokenized Data

```
import numpy as np
from pathlib import Path

# Configuration
SAVE_DIR = "/content/bangla_gpt_data"
Path(SAVE_DIR).mkdir(parents=True, exist_ok=True)
BIN_DTYPE = np.uint16  # 16-bit integers (vocab < 65536), for our model Vocabulary size: 32100

# Create memory-mapped files for efficient loading
for split in ("train", "validation"):
    tokens = tokenized_ds[split]
    total_tokens = int(np.sum(tokens["len"], dtype=np.uint64))

    # Create memory-mapped file
    mmap = np.memmap(
        Path(SAVE_DIR) / f"{split}.bin",
        dtype=BIN_DTYPE,
        mode="w+",
        shape=(total_tokens,)
    )

    # Write tokens to file
    idx = 0
    for token_list in tokens["ids"]:
        buf = np.asarray(token_list, dtype=BIN_DTYPE)
        mmap[idx:idx+len(buf)] = buf
        idx += len(buf)

    mmap.flush()
    print(f"Saved {split}.bin — {total_tokens:,} tokens")
```



Originally we will use drive mounting and saving the tokens in a folder, so that we can use these tokens however we want without requiring to download the whole dataset everytime.

Creating Metadata

```
import json

# Save important information
meta = {
    "tokenizer": TOKENIZER,
    "vocab_size": tokenizer.vocab_size,
    "train_tokens": int(np.sum(tokenized_ds["train"]["len"])),
    "val_tokens": int(np.sum(tokenized_ds["validation"]["len"])),
    "block_size": 128,              # Model context length
    "bin_dtype": "uint16",          # Data type used for storage
}

with open(Path(SAVE_DIR) / "meta.json", "w") as f:
    json.dump(meta, f, indent=2)
```

And.... we are done for this module! When you use drive mounting, the training.bin, validation.bin and meta.json will be saved in your drive and you can use those at any case without requiring to download the whole dataset .

Code for this module can be found here

code

Practical Exercises

*Apply these on any other dataset for practice

Exercise 1: Data Exploration

1. Load the first 100 CSV files from the dataset
2. Analyze the data distribution
3. Identify any data quality issues

Exercise 2: Custom Preprocessing

1. Create a function to clean Bangla text
2. Remove English text if present
3. Handle special characters and numbers

Exercise 3: Tokenization Comparison

1. Compare word-level vs. subword tokenization
2. Analyze vocabulary size differences
3. Test with out-of-vocabulary words

Exercise 4: Efficient Data Loading

1. Implement batch processing for large files
2. Add progress tracking
3. Handle memory limitations

Key Takeaways

- **Data Quality Matters:** Clean, consistent data leads to better models
- **Tokenization is Critical:** Proper tokenization is essential for Bangla text
- **Preprocessing Pipeline:** Systematic approach ensures reproducible results
- **Memory Efficiency:** Binary formats enable efficient training on large datasets
- **Metadata Tracking:** Save configuration for model reproduction