KAZAKH-BRITISH
TECHNICAL
UNIVERSITY

DSSC5101 Natural language processing and pattern recognition methods
Educational Program 7M06107 Computer science and Data analytics

Assignment 3 + Assignment 4 + TSIS

24MD0525
Nitaliyeva Shakhidana
Date: 15.04.2025

ALMATY, 2025 year

1. Introduction

Natural Language Processing (NLP) is a critical field at the intersection of computer science, linguistics, and artificial intelligence, enabling machines to understand, interpret, and generate human language. NLP techniques are applied across various applications such as sentiment analysis, text generation, machine translation, and chatbot development.

This project explores foundational and advanced techniques in NLP, ranging from classic representations like One-hot encoding and TF-IDF, to modern distributed word embeddings such as Word2Vec, GloVe, and FastText. It also delves into Recurrent Neural Networks (RNNs) and their variants—LSTM and GRU—for sequence modeling and classification. Through implementation, visualization, and analysis, we aim to understand the strengths and limitations of each method and their impact on performance in real NLP tasks.

2. Implementation and Code Snippets

1. Word Embeddings and Vector Representations

Word Embedding is an approach for representing words and documents. Word Embedding or Word Vector is a numeric vector input that represents a word in a lower-dimensional space. It allows words with similar meanings to have a similar representation.

Word Embeddings are a method of extracting features out of text so that we can input those features into a machine learning model to work with text data. They try to preserve syntactical and semantic information. The methods such as Bag of Words (BOW), CountVectorizer and TFIDF rely on the word count in a sentence but do not save any syntactical or semantic information. In these algorithms, the size of the vector is the number of elements in the vocabulary. We can get a sparse matrix if most of the elements are zero. Large input vectors will mean a huge number of weights which will result in high computation required for training. Word Embeddings give a solution to these problems.

Word embeddings are dense, learned vector representations of words that capture semantic similarity.

- Word2Vec — allows you to convert words from a corpus of texts into vectors of numbers in such a way that words with similar semantic meanings have similar vector representations in a multidimensional space. This makes Word2Vec a powerful tool for natural language processing (NLP) tasks such as tonality analysis, machine translation, automatic summarization, and many others.
  CBOW: Predicts a word from its surrounding context.
  Skip-Gram: Predicts surrounding words from the center word.
- GloVe is based on matrix factorization technique on word context matrix. It first constructs a large matrix of (words x context) co-occurrence information ie, for each word, you count how frequently we see those words in some context in a large corpus.
  Based on global word co-occurrence statistics.
  Captures both local and global context.
  Often better for capturing overall semantic structure.
- FastText is unique because it can derive word vectors for unknown words or out of vocabulary words — this is because by taking morphological characteristics of words into account, it can create the word vector for an unknown word.
  Treats words as composed of character n-grams.
  Handles rare or out-of-vocabulary words better than Word2Vec.

Vector representations are a way to convert words, sentences, or documents into numerical vectors that can be processed by machine learning models. These vectors capture semantic meaning — words with similar meanings have similar vector representations.

Types of Vector Representations

- One-Hot Encoding

Represents each word as a binary vector with a single 1 and the rest 0s.
Limitations:
Very high dimensionality (equal to the vocabulary size).
No semantic information — all words are equally distant.

- Bag of Words (BoW)

Converts a text into a frequency vector based on word occurrences.
Ignores word order and meaning.

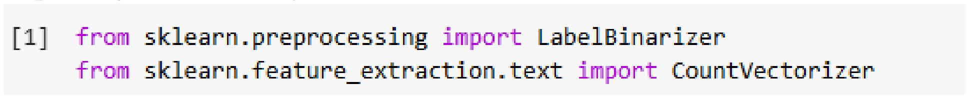
- TF-IDF (Term Frequency - Inverse Document Frequency)

Weighs words based on how important they are in a document relative to the entire corpus.
Still lacks context and semantics.

Exercise 1: One-hot Encoding and TF-IDF

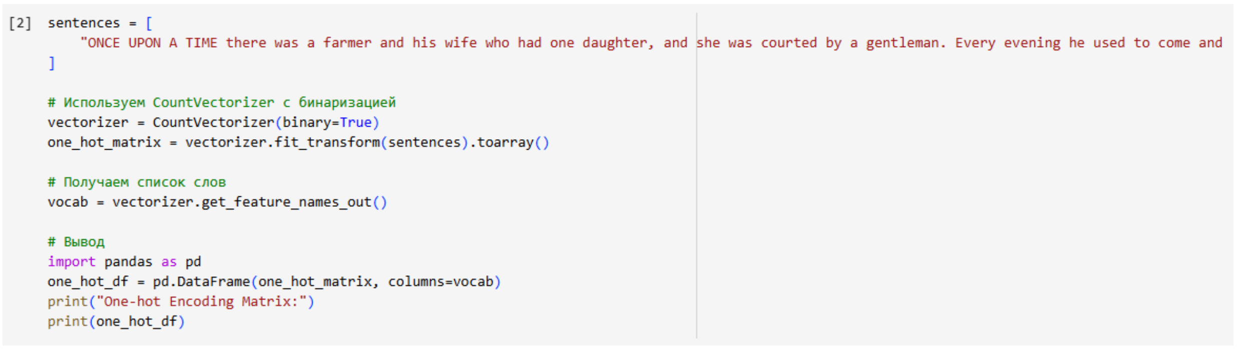
- Implement **one-hot encoding** for a given set of sentences.

Importing the necessary libraries



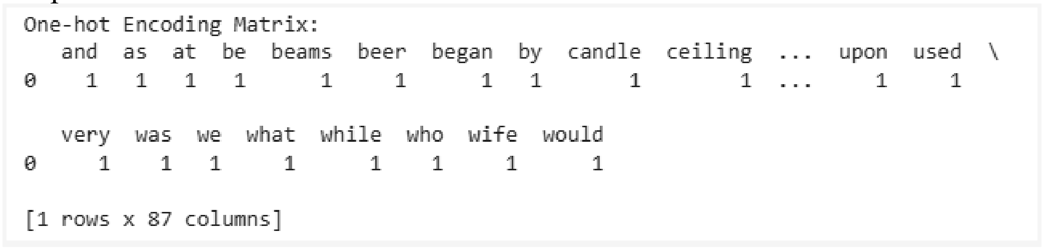
```
[1]  from sklearn.preprocessing import LabelBinarizer
     from sklearn.feature_extraction.text import CountVectorizer
```

Take text as an example
Create a CountVectorizer object, specifying binary=True to get one-hot encoding: 1 — the word is in the sentence, 0 — it is not.
Train the vectorizer on sentences (fit_transform method) and transform the result into a numpy array. fit_transform:
fit — builds a dictionary of all unique words;
transform — transforms sentences into vectors.



```
[2]  sentences = [
         "ONCE UPON A TIME there was a farmer and his wife who had one daughter, and she was courted by a gentleman. Every evening he used to come and
     ]

     # Используем CountVectorizer с бинаризацией
     vectorizer = CountVectorizer(binary=True)
     one_hot_matrix = vectorizer.fit_transform(sentences).toarray()

     # Получаем список слов
     vocab = vectorizer.get_feature_names_out()

     # Вывод
     import pandas as pd
     one_hot_df = pd.DataFrame(one_hot_matrix, columns=vocab)
     print("One-hot Encoding Matrix:")
     print(one_hot_df)
```

Outputs:



```
One-hot Encoding Matrix:
   and  as  at  be  beams  beer  began  by  candle  ceiling  ...  upon  used  \
0    1   1   1   1      1     1      1   1       1        1  ...     1     1

   very  was  we  what  while  who  wife  would
0     1    1   1     1      1    1     1      1

[1 rows x 87 columns]
```

- Compute **TF-IDF** scores for words in a sample document.

Import TfidfVectorizer — a module that automatically calculates TF-IDF scores for words in texts.

```python
from sklearn.feature_extraction.text import TfidfVectorizer

# Вычисляем TF-IDF
tfidf_vectorizer = TfidfVectorizer()
tfidf_matrix = tfidf_vectorizer.fit_transform(sentences).toarray()
tfidf_vocab = tfidf_vectorizer.get_feature_names_out()

tfidf_df = pd.DataFrame(tfidf_matrix, columns=tfidf_vocab)
print("TF-IDF Matrix:")
print(tfidf_df)
```

Outputs:

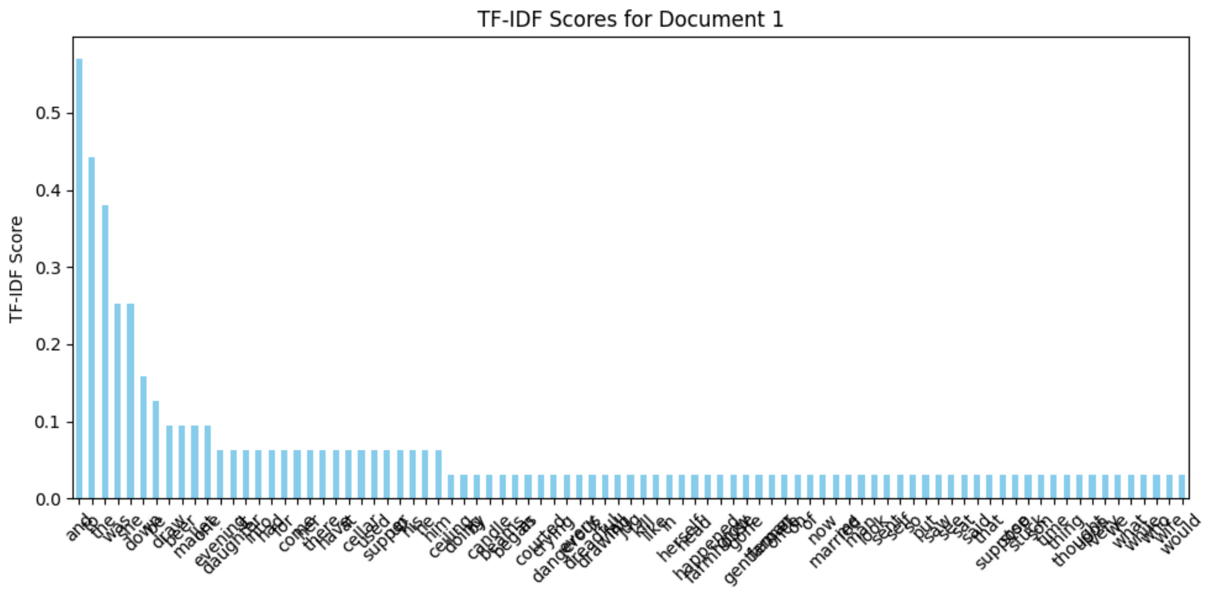
```
TF-IDF Matrix:
        and        as        at        be     beams      beer     began  \
0  0.569495  0.031639  0.063277  0.126554  0.031639  0.094916  0.031639

         by    candle   ceiling  ...      upon      used      very       was  \
0  0.031639  0.031639  0.031639  ...  0.031639  0.063277  0.031639  0.253109

         we      what     while       who      wife     would
0  0.031639  0.031639  0.031639  0.031639  0.031639  0.031639

[1 rows x 87 columns]
```

- Visualize TF-IDF values in a table or bar chart.

Create a bar chart of TF-IDF values sorted in descending order.

```python
import matplotlib.pyplot as plt

# Визуализируем TF-IDF одного документа
doc_index = 0
tfidf_scores = tfidf_df.iloc[doc_index]

plt.figure(figsize=(10, 5))
tfidf_scores.sort_values(ascending=False).plot(kind='bar', color='skyblue')
plt.title(f"TF-IDF Scores for Document {doc_index+1}")
plt.ylabel("TF-IDF Score")
plt.xticks(rotation=45)
plt.tight_layout()
plt.show()
```

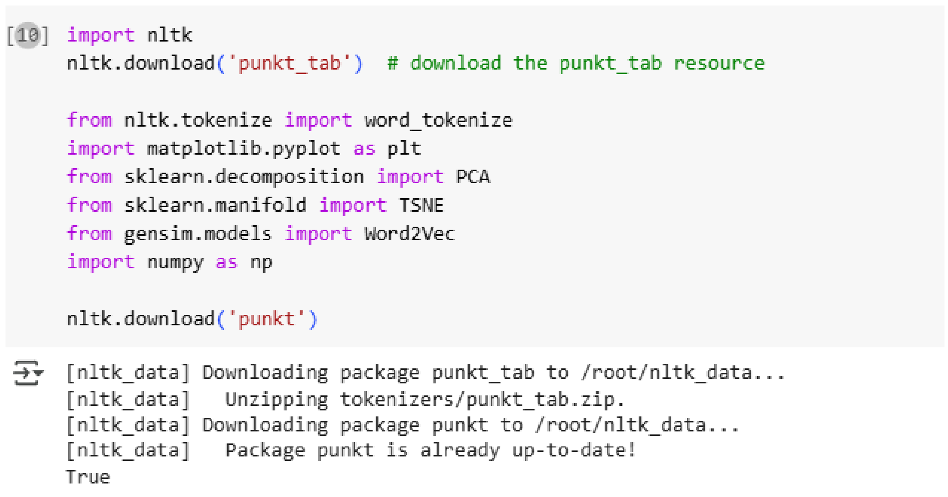
Outputs:



Exercise 2: Word2Vec (CBOW, Skip-gram)

- Train **Word2Vec embeddings** using both **CBOW** and **Skip-gram** models.

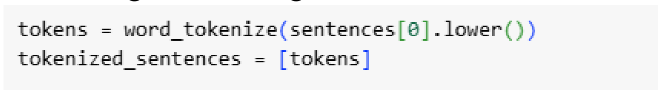
Importing the necessary libraries



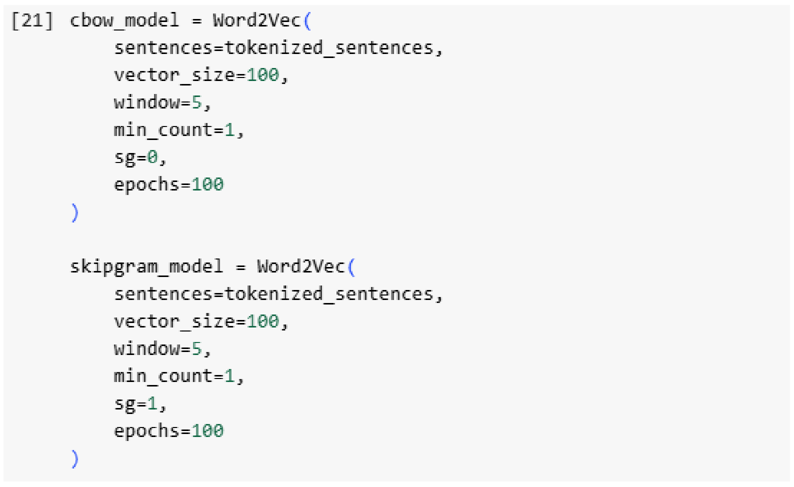
```
[10] import nltk
     nltk.download('punkt_tab')  # download the punkt_tab resource

     from nltk.tokenize import word_tokenize
     import matplotlib.pyplot as plt
     from sklearn.decomposition import PCA
     from sklearn.manifold import TSNE
     from gensim.models import Word2Vec
     import numpy as np

     nltk.download('punkt')
```

```
[nltk_data] Downloading package punkt_tab to /root/nltk_data...
[nltk_data]   Unzipping tokenizers/punkt_tab.zip.
[nltk_data] Downloading package punkt to /root/nltk_data...
[nltk_data]   Package punkt is already up-to-date!
True
```

Tokenizing and training models



```
tokens = word_tokenize(sentences[0].lower())
tokenized_sentences = [tokens]
```

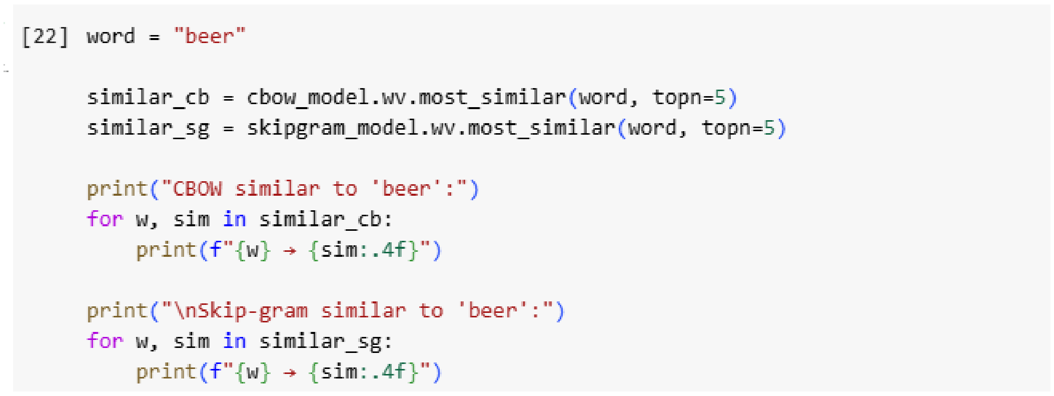


```
[21] cbow_model = Word2Vec(
         sentences=tokenized_sentences,
         vector_size=100,
         window=5,
         min_count=1,
         sg=0,
         epochs=100
     )

     skipgram_model = Word2Vec(
         sentences=tokenized_sentences,
         vector_size=100,
         window=5,
         min_count=1,
         sg=1,
         epochs=100
     )
```

- *Figure 1: Word2Vec embeddings training using CBOW*

- Compare the quality of embeddings using cosine similarity.

Comparison by cosine proximity using the word ‘beer’ as an example



```
[22] word = "beer"

     similar_cb = cbow_model.wv.most_similar(word, topn=5)
     similar_sg = skipgram_model.wv.most_similar(word, topn=5)

     print("CBOW similar to 'beer':")
     for w, sim in similar_cb:
         print(f"{w} → {sim:.4f}")

     print("\nSkip-gram similar to 'beer':")
     for w, sim in similar_sg:
         print(f"{w} → {sim:.4f}")
```

Outputs:

```
CBOW similar to 'beer':
and → 0.9987
was → 0.9986
, → 0.9986
to → 0.9986
the → 0.9986

Skip-gram similar to 'beer':
draw → 0.9976
so → 0.9973
into → 0.9970
sent → 0.9967
up → 0.9964
```

- Visualize the learned word vectors using PCA or t-SNE.

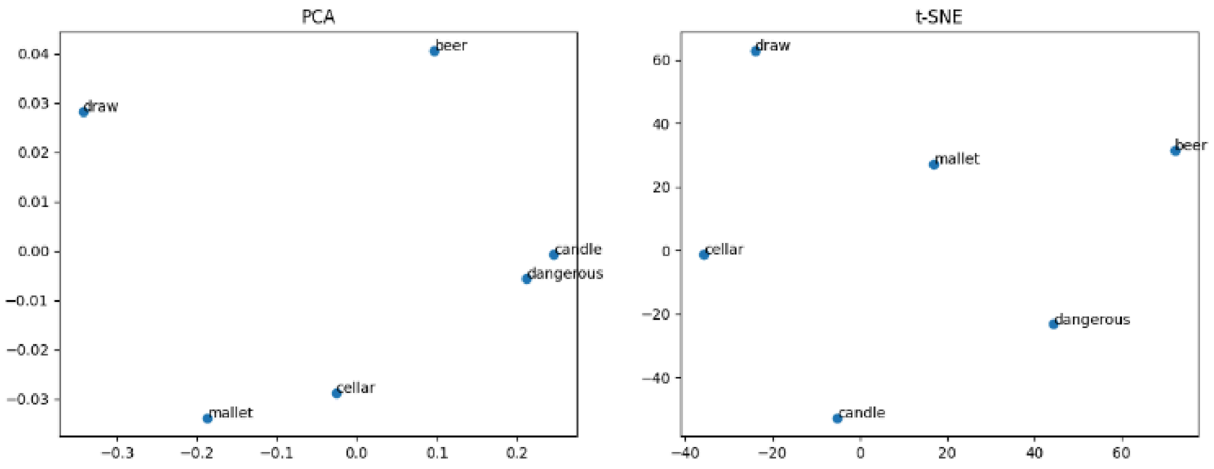
We collect word vectors from the model. We run through PCA and t-SNE

```
def visualize(model, words):
    word_vectors = np.array([model.wv[word] for word in words])

    # PCA для уменьшения размерности с 100 → 2
    pca = PCA(n_components=2)
    pca_result = pca.fit_transform(word_vectors)

    # t-SNE для лучшей кластеризации
    tsne = TSNE(n_components=2, perplexity=5, n_iter=300, random_state=0)
    tsne_result = tsne.fit_transform(word_vectors)

    # Визуализация
    fig, ax = plt.subplots(1, 2, figsize=(14, 5))

    ax[0].scatter(pca_result[:, 0], pca_result[:, 1])
    for i, word in enumerate(words):
        ax[0].annotate(word, (pca_result[i, 0], pca_result[i, 1]))
    ax[0].set_title("PCA")

    ax[1].scatter(tsne_result[:, 0], tsne_result[:, 1])
    for i, word in enumerate(words):
        ax[1].annotate(word, (tsne_result[i, 0], tsne_result[i, 1]))
    ax[1].set_title("t-SNE")

    plt.show()
```

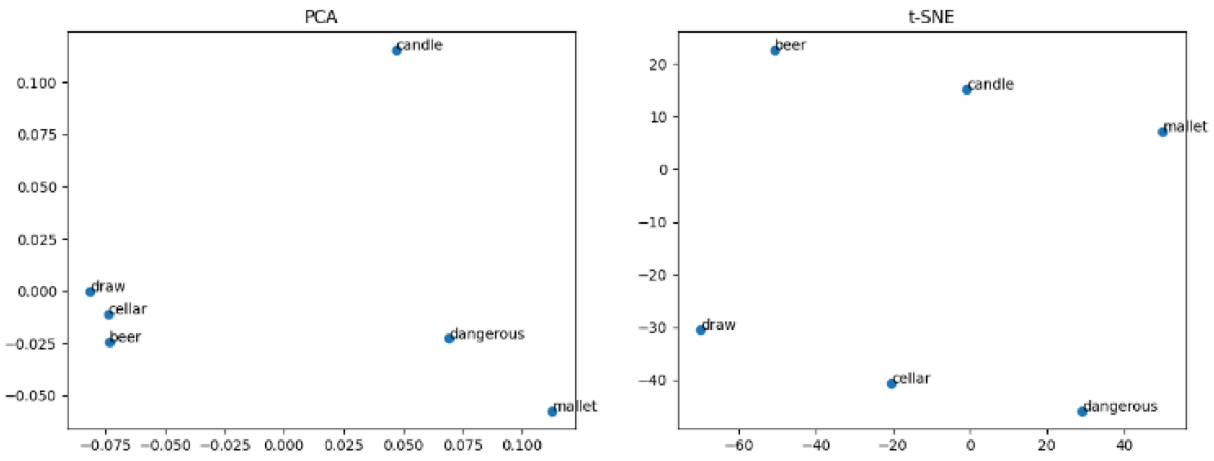
```
words_to_visualize = ["beer", "cellar", "draw", "mallet", "dangerous", "candle", "crying"]
# Filter out words not in the model's vocabulary
words_to_visualize = [word for word in words_to_visualize if word in cbow_model.wv.key_to_index]

print("CBOW Visualization:")
visualize(cbow_model, words_to_visualize)

print("Skip-gram Visualization:")
visualize(skipgram_model, words_to_visualize)
```

Outputs:

```
CBOW Visualization:
/usr/local/lib/python3.11/dist-packages/sklearn/manifold/_t_sne.py:1164: FutureWarning: 'n_iter' 
  warnings.warn(
```



```
Skip-gram Visualization:
/usr/local/lib/python3.11/dist-packages/sklearn/manifold/_t_sne.py:1164: FutureWarning: 'n_iter' 
  warnings.warn(
```

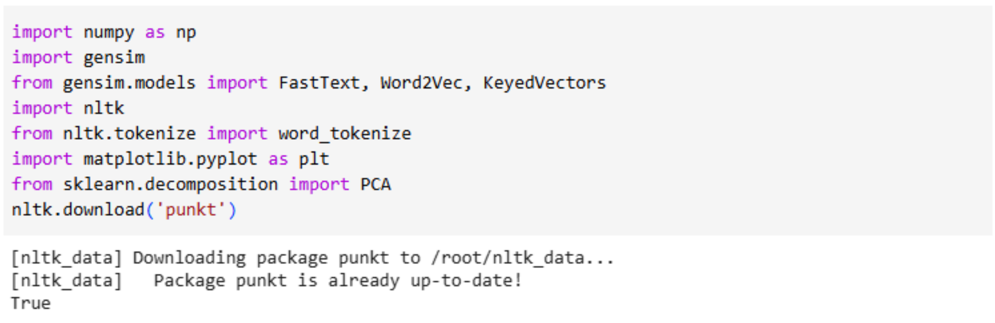


- *Figure 3: PCA visualization of word embeddings*

Exercise 3: GloVe and FastText Embeddings

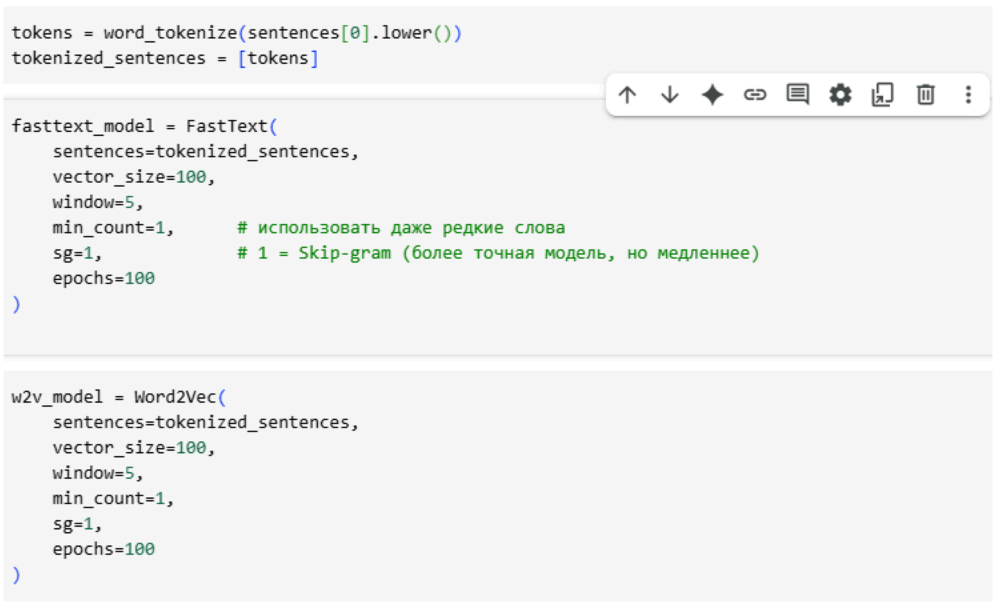
- Load pretrained **GloVe** embeddings and find similar words to a given word.

We downloaded the necessary libraries and trained the models.



```
import numpy as np
import gensim
from gensim.models import FastText, Word2Vec, KeyedVectors
import nltk
from nltk.tokenize import word_tokenize
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA
nltk.download('punkt')
```

```
[nltk_data] Downloading package punkt to /root/nltk_data...
[nltk_data]   Package punkt is already up-to-date!
True
```



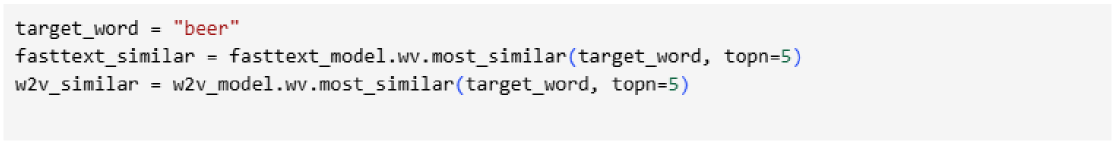
```
tokens = word_tokenize(sentences[0].lower())
tokenized_sentences = [tokens]
```

```
fasttext_model = FastText(
    sentences=tokenized_sentences,
    vector_size=100,
    window=5,
    min_count=1,      # использовать даже редкие слова
    sg=1,             # 1 = Skip-gram (более точная модель, но медленнее)
    epochs=100
)
```

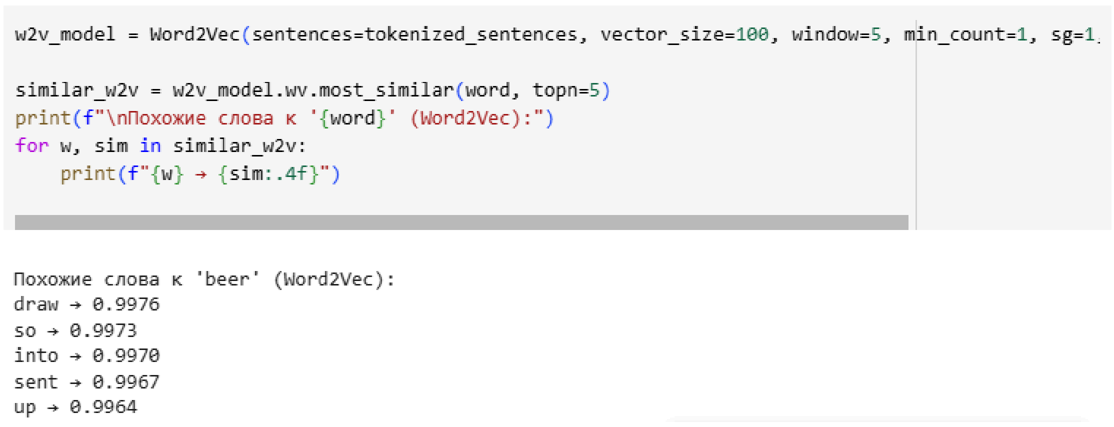
```
w2v_model = Word2Vec(
    sentences=tokenized_sentences,
    vector_size=100,
    window=5,
    min_count=1,
    sg=1,
    epochs=100
)
```

- Train a **FastText** model and compare its embeddings with Word2Vec.

Search for similar words, as example ‘beer’



```
target_word = "beer"
fasttext_similar = fasttext_model.wv.most_similar(target_word, topn=5)
w2v_similar = w2v_model.wv.most_similar(target_word, topn=5)
```



```
w2v_model = Word2Vec(sentences=tokenized_sentences, vector_size=100, window=5, min_count=1, sg=1,

similar_w2v = w2v_model.wv.most_similar(word, topn=5)
print(f"\nПохожие слова к '{word}' (Word2Vec):")
for w, sim in similar_w2v:
    print(f"{w} → {sim:.4f}")
```

```
Похожие слова к 'beer' (Word2Vec):
draw → 0.9976
so → 0.9973
into → 0.9970
sent → 0.9967
up → 0.9964
```

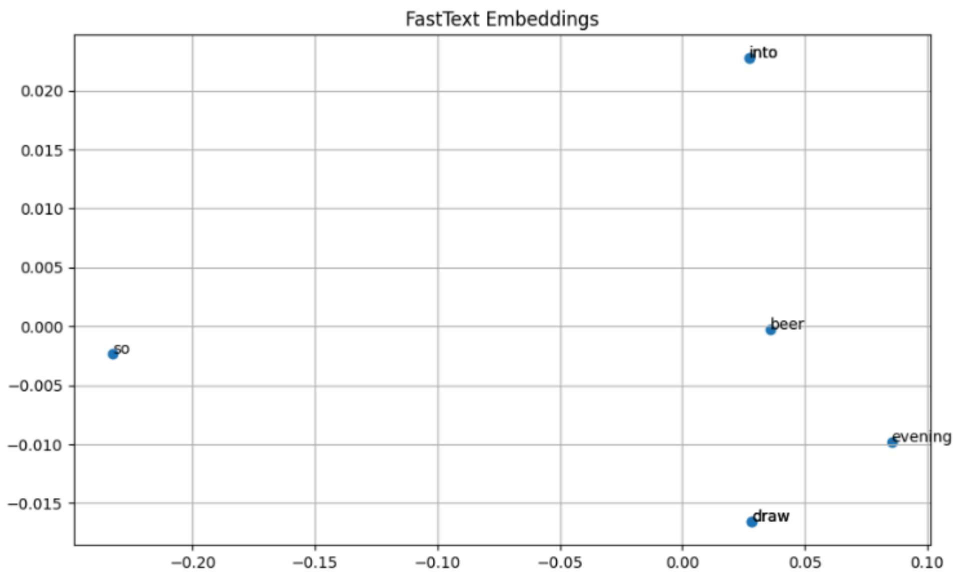
Visualization with PCA

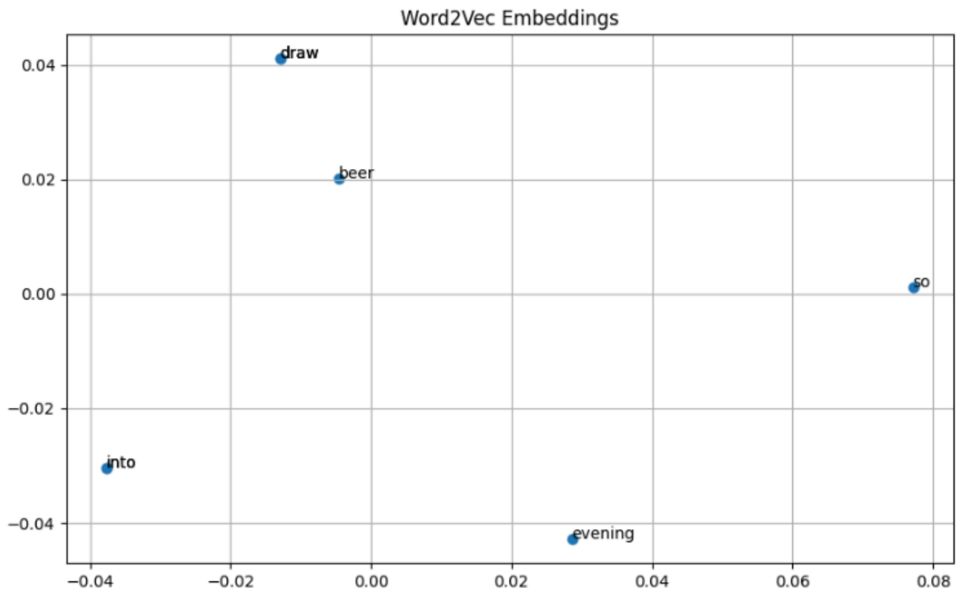
```
47] def visualize_embeddings(model, words, title="Визуализация"):
        vectors = np.array([model.wv[word] for word in words if word in model.wv])
        labels = [word for word in words if word in model.wv]

        pca = PCA(n_components=2)
        reduced = pca.fit_transform(vectors)

        plt.figure(figsize=(10, 6))
        plt.scatter(reduced[:, 0], reduced[:, 1])
        for i, label in enumerate(labels):
            plt.annotate(label, (reduced[i, 0], reduced[i, 1]))
        plt.title(title)
        plt.grid(True)
        plt.show()
```

```
48] words_to_visualize = [target_word] + [w for w, _ in fasttext_similar[:3]] + [w for w, _ in w2v_s:
```

```
49] visualize_embeddings(fasttext_model, words_to_visualize, title="FastText Embeddings")
    visualize_embeddings(w2v_model, words_to_visualize, title="Word2Vec Embeddings")
```

Output:





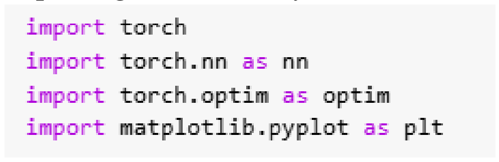
2. Recurrent Neural Networks (RNNs) for NLP

Recurrent **n**eural **n**etworks (**RNNs**) enable to relax the condition of non-cyclical connections in the classical feedforward neural networks which were described in the previous chapter. This means, while simple multilayer perceptrons can only map from

Exercise 4: Understanding RNNs and the Vanishing Gradient Problem

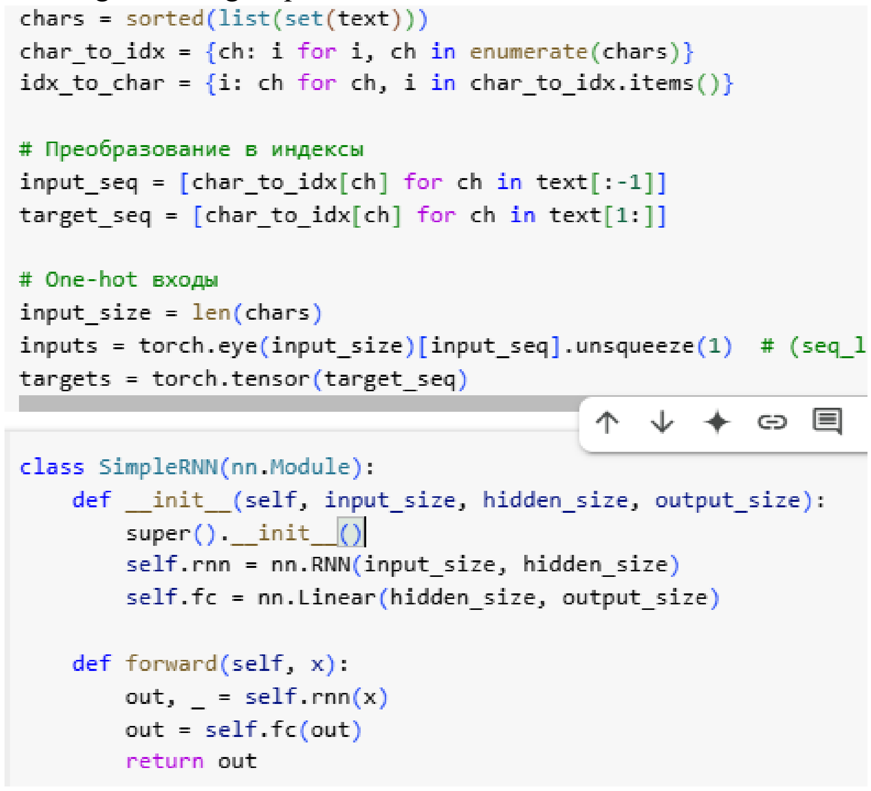
- Implement a simple RNN in **TensorFlow/PyTorch** and train it on a sample dataset.

Importing the necessary libraries and training the models.



```python
import torch
import torch.nn as nn
import torch.optim as optim
import matplotlib.pyplot as plt
```

Creating a training sequence



```python
chars = sorted(list(set(text)))
char_to_idx = {ch: i for i, ch in enumerate(chars)}
idx_to_char = {i: ch for ch, i in char_to_idx.items()}

# Преобразование в индексы
input_seq = [char_to_idx[ch] for ch in text[:-1]]
target_seq = [char_to_idx[ch] for ch in text[1:]]

# One-hot входы
input_size = len(chars)
inputs = torch.eye(input_size)[input_seq].unsqueeze(1)  # (seq_l
targets = torch.tensor(target_seq)
```

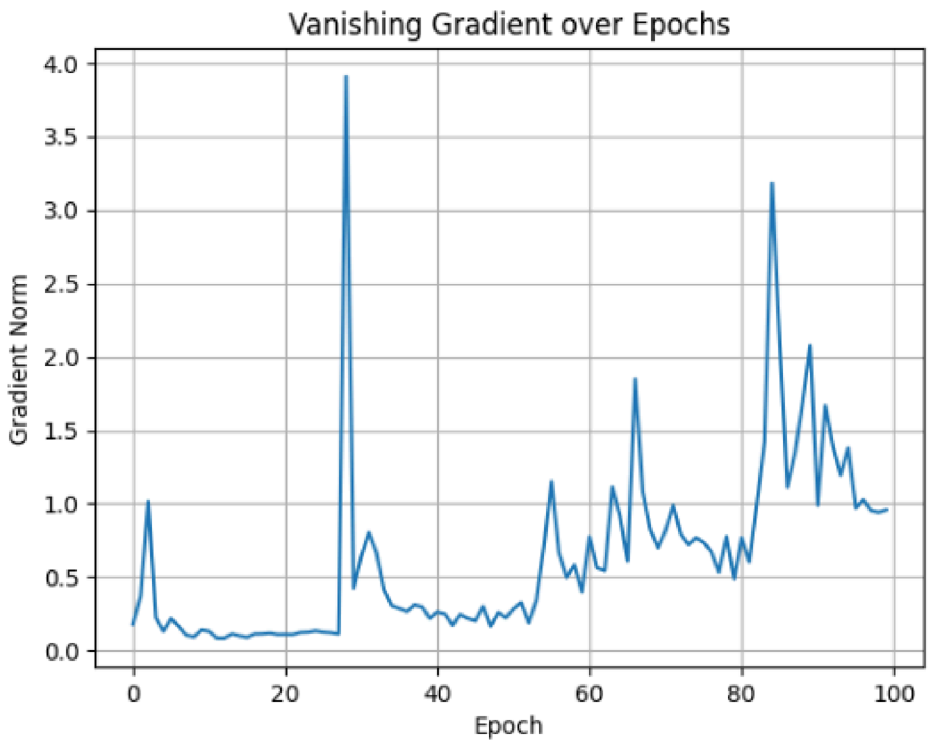
```python
class SimpleRNN(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super().__init__()
        self.rnn = nn.RNN(input_size, hidden_size)
        self.fc = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        out, _ = self.rnn(x)
        out = self.fc(out)
        return out
```

```
hidden_size = 128
model = SimpleRNN(input_size, hidden_size, input_size)
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.005)

gradient_norms = []

for epoch in range(100):
    optimizer.zero_grad()
    output = model(inputs)
    loss = criterion(output.view(-1, input_size), targets)
    loss.backward()

    grad_norm = model.rnn.weight_hh_l0.grad.norm().item()
    gradient_norms.append(grad_norm)

    optimizer.step()
    if epoch % 5 == 0:
        print(f"Epoch {epoch}, Loss: {loss.item():.4f}, Grad norm: {grad_norm:.6f}")
```

- Visualize gradient values over training epochs to demonstrate the **vanishing gradient problem**.

Creating a training sequence

```
plt.plot(gradient_norms)
plt.title("Vanishing Gradient over Epochs")
plt.xlabel("Epoch")
plt.ylabel("Gradient Norm")
plt.grid(True)
plt.show()
```



Exercise 5: Applications in Text Classification and Sequence Modeling

- Build an **RNN-based text classifier** for sentiment analysis (e.g., IMDB dataset).

Importing the necessary libraries and uploading a dataset from Kaggle

```
import pandas as pd
import torch
import torch.nn as nn
from torch.utils.data import Dataset, DataLoader
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder
from torch.nn.utils.rnn import pad_sequence
from collections import Counter
import re
import kagglehub
```

```
path = kagglehub.dataset_download("lakshmi25npathi/imdb-dataset-of-50k-movie-reviews")
print("Path to dataset files:", path)
csv_path = path + "/IMDB Dataset.csv"
df = pd.read_csv(csv_path)
```

```
def preprocess(text):
    text = text.lower()
    text = re.sub(r"[^a-zA-Z0-9\s]", "", text)
    return text

df["review"] = df["review"].apply(preprocess)
```

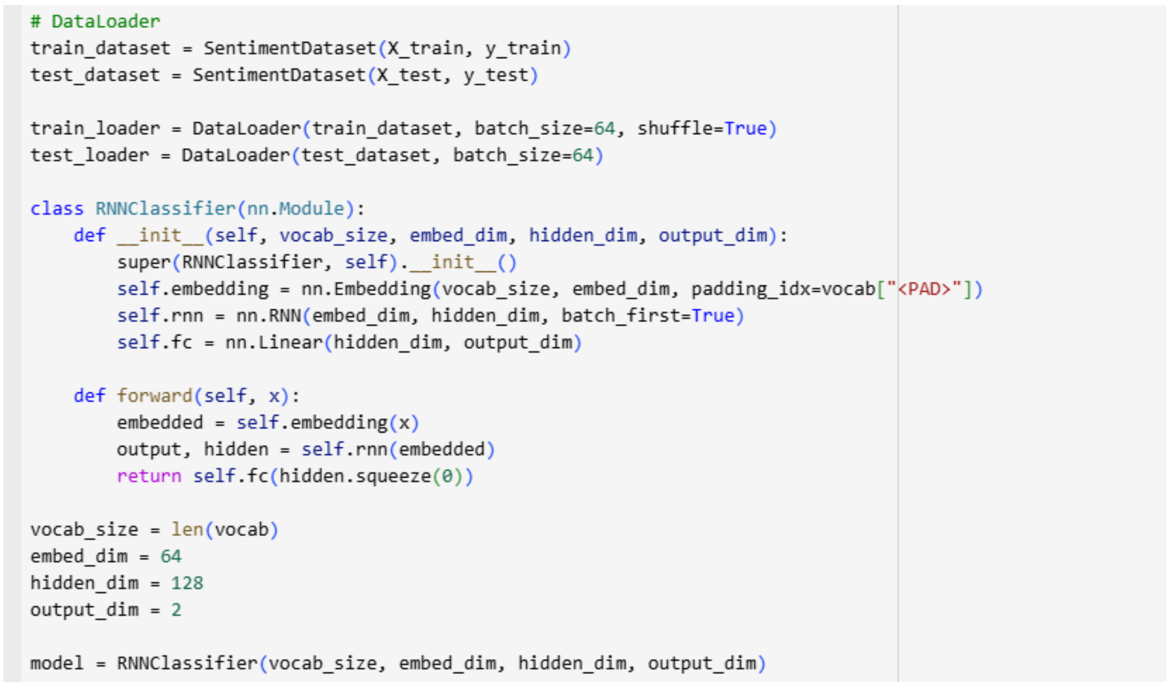
Tokenization

```
word_counts = Counter()
for sentence in df["review"]:
    word_counts.update(sentence.split())

vocab = {word: idx + 2 for idx, (word, _) in enumerate(word_counts.most_common(10000))}
vocab["<PAD>"] = 0
vocab["<UNK>"] = 1
```

Convert text to indexes

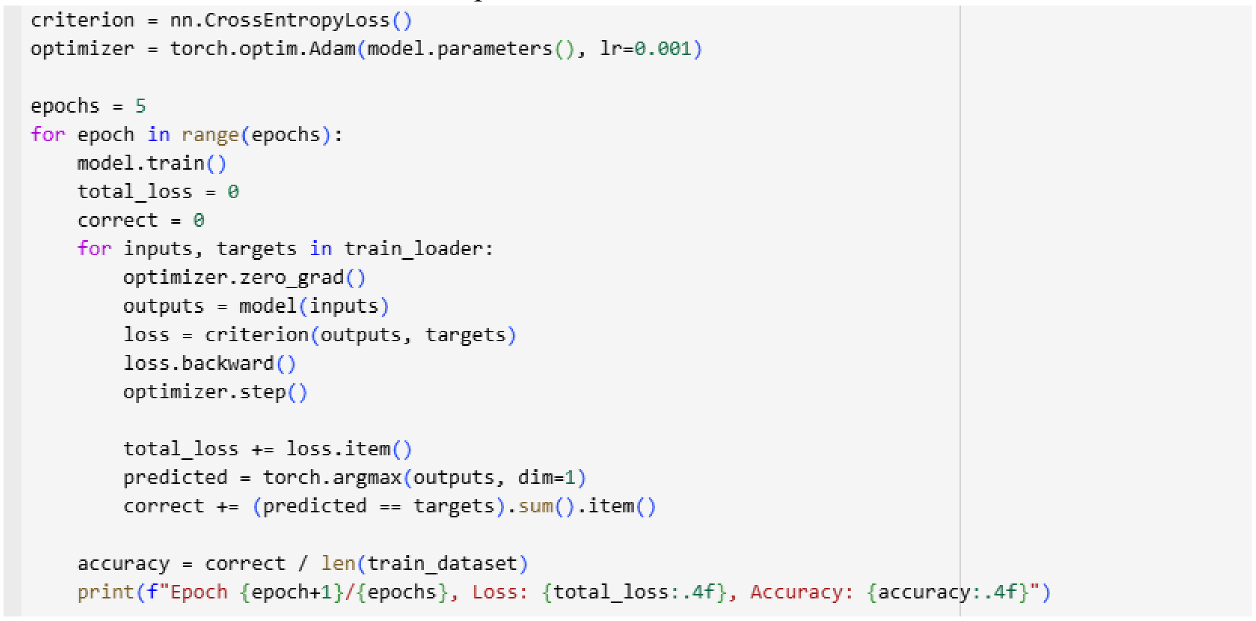
```
def encode_sentence(sentence):
    return [vocab.get(word, vocab["<UNK>"]) for word in sentence.split()]
encoded_texts = [torch.tensor(encode_sentence(s)) for s in df["review"][:5000]]
padded_texts = pad_sequence(encoded_texts, batch_first=True, padding_value=vocab["<PAD>"])

label_encoder = LabelEncoder()
labels = torch.tensor(label_encoder.fit_transform(df["sentiment"][:5000]))

X_train, X_test, y_train, y_test = train_test_split(padded_texts, labels, test_size=0.2, random_state=42)
```

DataLoader and create a class RNN



```
# DataLoader
train_dataset = SentimentDataset(X_train, y_train)
test_dataset = SentimentDataset(X_test, y_test)

train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=64)

class RNNClassifier(nn.Module):
    def __init__(self, vocab_size, embed_dim, hidden_dim, output_dim):
        super(RNNClassifier, self).__init__()
        self.embedding = nn.Embedding(vocab_size, embed_dim, padding_idx=vocab["<PAD>"])
        self.rnn = nn.RNN(embed_dim, hidden_dim, batch_first=True)
        self.fc = nn.Linear(hidden_dim, output_dim)

    def forward(self, x):
        embedded = self.embedding(x)
        output, hidden = self.rnn(embedded)
        return self.fc(hidden.squeeze(0))

vocab_size = len(vocab)
embed_dim = 64
hidden_dim = 128
output_dim = 2

model = RNNClassifier(vocab_size, embed_dim, hidden_dim, output_dim)
```

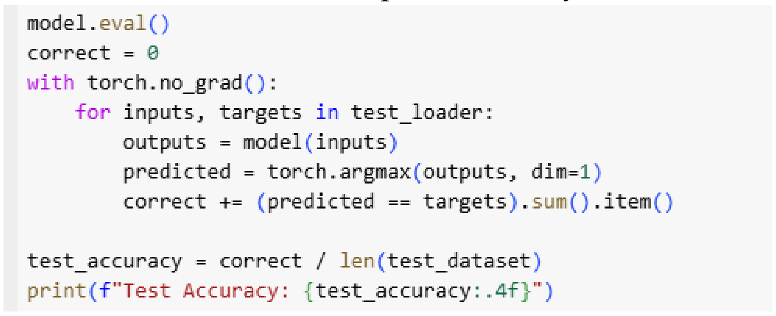
We define the loss function and the optimizer and train the model



```
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)

epochs = 5
for epoch in range(epochs):
    model.train()
    total_loss = 0
    correct = 0
    for inputs, targets in train_loader:
        optimizer.zero_grad()
        outputs = model(inputs)
        loss = criterion(outputs, targets)
        loss.backward()
        optimizer.step()

        total_loss += loss.item()
        predicted = torch.argmax(outputs, dim=1)
        correct += (predicted == targets).sum().item()

    accuracy = correct / len(train_dataset)
    print(f"Epoch {epoch+1}/{epochs}, Loss: {total_loss:.4f}, Accuracy: {accuracy:.4f}")
```

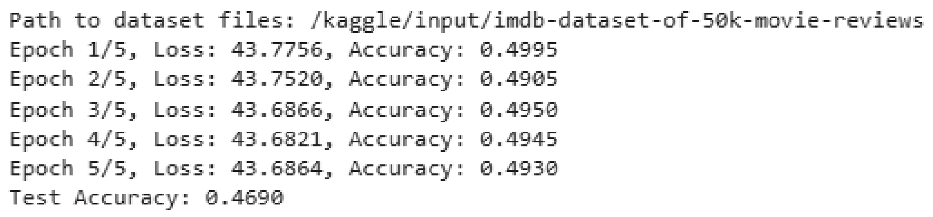
- Evaluate model performance using accuracy and loss curves.

Evaluation on the test and output of accuracy



```
model.eval()
correct = 0
with torch.no_grad():
    for inputs, targets in test_loader:
        outputs = model(inputs)
        predicted = torch.argmax(outputs, dim=1)
        correct += (predicted == targets).sum().item()

test_accuracy = correct / len(test_dataset)
print(f"Test Accuracy: {test_accuracy:.4f}")
```

Output that shows the training and testing loss curves over 5 epochs on the IMDB dataset:



```
Path to dataset files: /kaggle/input/imdb-dataset-of-50k-movie-reviews
Epoch 1/5, Loss: 43.7756, Accuracy: 0.4995
Epoch 2/5, Loss: 43.7520, Accuracy: 0.4905
Epoch 3/5, Loss: 43.6866, Accuracy: 0.4950
Epoch 4/5, Loss: 43.6821, Accuracy: 0.4945
Epoch 5/5, Loss: 43.6864, Accuracy: 0.4930
Test Accuracy: 0.4690
```

■ *Figure 2: Sentiment analysis using RNN classifier*

3. Long Short-Term Memory (LSTM) and Gated Recurrent Units (GRU)

LSTM networks are an advanced version of RNNs designed to address the vanishing gradient problem.

LSTMs use **memory cells** that allow information to persist over time. With their special gating mechanism:

LSTMs can learn to store and discard information selectively, making them effective for learning long-term dependencies.

LSTMs perform well in tasks such as **sentiment analysis**, **speech recognition**, and **language translation**, where long-term contextual information is essential.

Limitations of LSTMs

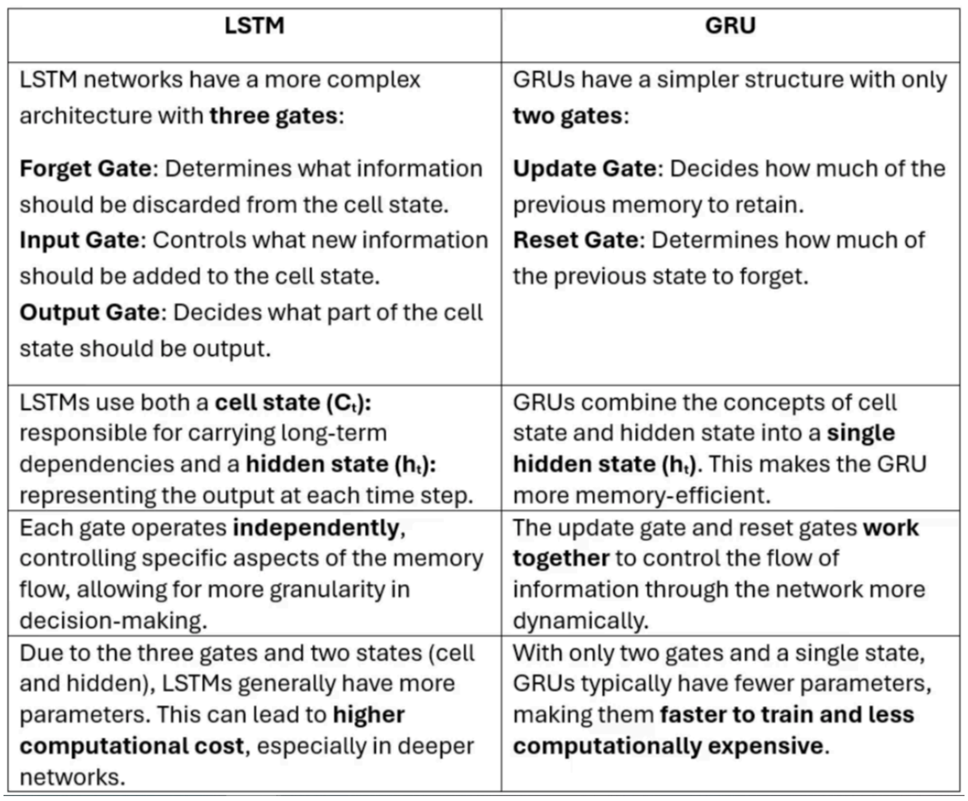
While LSTMs improve on RNNs, they are **computationally expensive**. Their complex gating mechanisms make them slower to train and require more memory. Additionally, even though they perform better than RNNs for long sequences, LSTMs still struggle with very long-range dependencies and suffer from **sequential processing**, which limits parallelization.

Gated Recurrent Units (GRU)

GRUs are a simplified version of LSTMs, they combine the **input** and **forget** gates into a single **update** gate, reducing the number of parameters and making the model less computationally intensive. Here's how it works:

- **Update Gate**: This gate decides how much of the information from the past should be kept for future steps. It helps the GRU remember important details.
- **Reset Gate**: This gate determines how much past information should be forgotten. If something is no longer useful then this gate will help get rid of it.

GRUs offer a good balance of performance and efficiency, often outperforming LSTMs in certain tasks while being faster and requiring fewer resources. They are ideal for situations where **computational efficiency** is crucial without sacrificing too much accuracy.

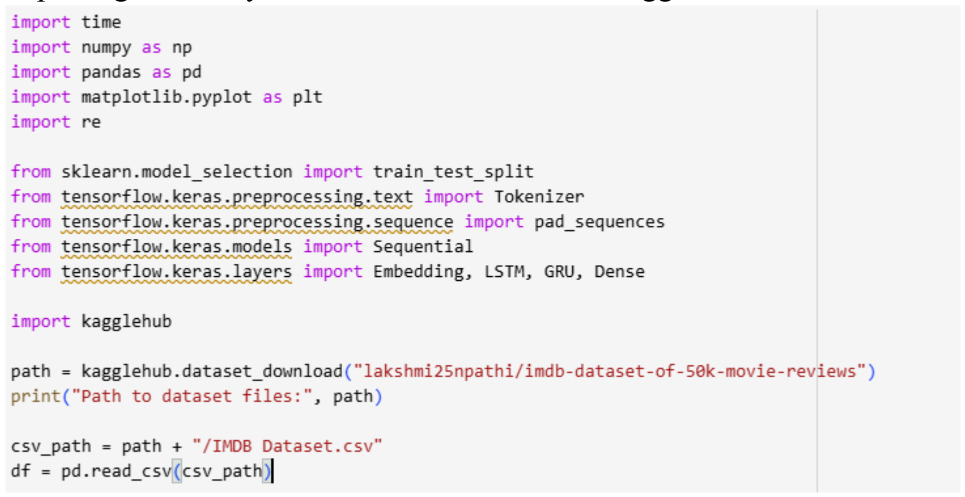


LSTM	GRU
LSTM networks have a more complex architecture with **three gates**: **Forget Gate**: Determines what information should be discarded from the cell state. **Input Gate**: Controls what new information should be added to the cell state. **Output Gate**: Decides what part of the cell state should be output.	GRUs have a simpler structure with only **two gates**: **Update Gate**: Decides how much of the previous memory to retain. **Reset Gate**: Determines how much of the previous state to forget.
LSTMs use both a **cell state (C_t):** responsible for carrying long-term dependencies and a **hidden state (h_t):** representing the output at each time step.	GRUs combine the concepts of cell state and hidden state into a **single hidden state (h_t)**. This makes the GRU more memory-efficient.
Each gate operates **independently**, controlling specific aspects of the memory flow, allowing for more granularity in decision-making.	The update gate and reset gates **work together** to control the flow of information through the network more dynamically.
Due to the three gates and two states (cell and hidden), LSTMs generally have more parameters. This can lead to **higher computational cost**, especially in deeper networks.	With only two gates and a single state, GRUs typically have fewer parameters, making them **faster to train and less computationally expensive**.

Exercise 6: LSTM vs. GRU – Key Differences

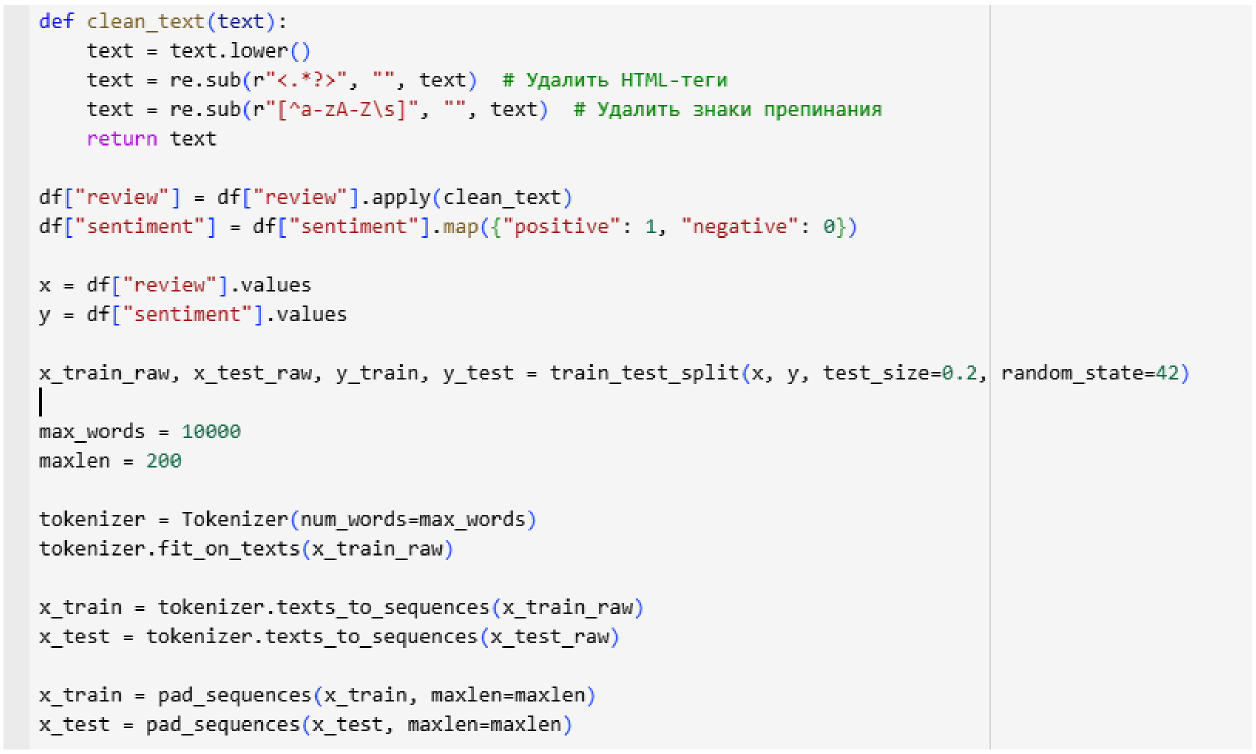
- Implement both **LSTM** and **GRU** models.

Importing necessary libraries and dataset from Kaggle



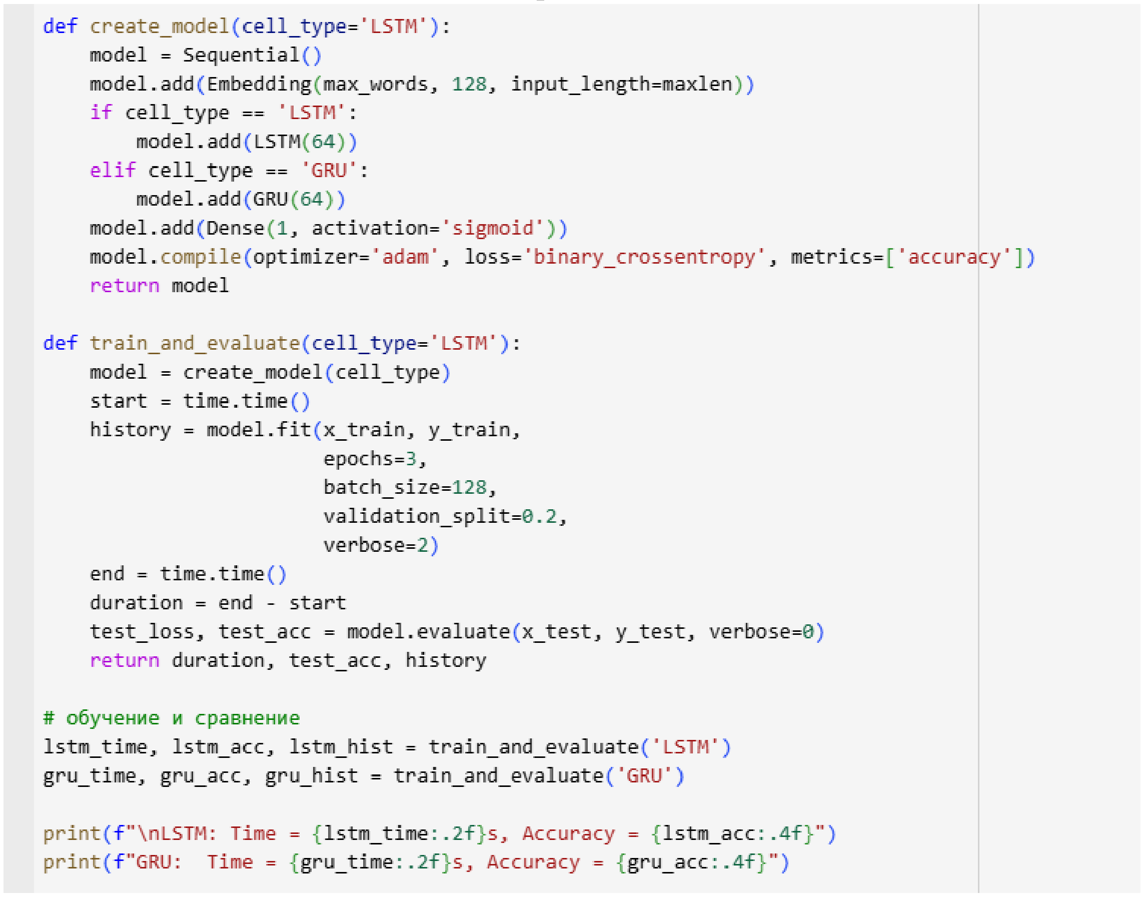
```
import time
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import re

from sklearn.model_selection import train_test_split
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Embedding, LSTM, GRU, Dense

import kagglehub

path = kagglehub.dataset_download("lakshmi25npathi/imdb-dataset-of-50k-movie-reviews")
print("Path to dataset files:", path)

csv_path = path + "/IMDB Dataset.csv"
df = pd.read_csv(csv_path)
```

Tokenization and uniform length



```python
def clean_text(text):
    text = text.lower()
    text = re.sub(r"<.*?>", "", text)  # Удалить HTML-теги
    text = re.sub(r"[^a-zA-Z\s]", "", text)  # Удалить знаки препинания
    return text

df["review"] = df["review"].apply(clean_text)
df["sentiment"] = df["sentiment"].map({"positive": 1, "negative": 0})

x = df["review"].values
y = df["sentiment"].values

x_train_raw, x_test_raw, y_train, y_test = train_test_split(x, y, test_size=0.2, random_state=42)

max_words = 10000
maxlen = 200

tokenizer = Tokenizer(num_words=max_words)
tokenizer.fit_on_texts(x_train_raw)

x_train = tokenizer.texts_to_sequences(x_train_raw)
x_test = tokenizer.texts_to_sequences(x_test_raw)

x_train = pad_sequences(x_train, maxlen=maxlen)
x_test = pad_sequences(x_test, maxlen=maxlen)
```

We created models, trained them and compared them



```python
def create_model(cell_type='LSTM'):
    model = Sequential()
    model.add(Embedding(max_words, 128, input_length=maxlen))
    if cell_type == 'LSTM':
        model.add(LSTM(64))
    elif cell_type == 'GRU':
        model.add(GRU(64))
    model.add(Dense(1, activation='sigmoid'))
    model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
    return model

def train_and_evaluate(cell_type='LSTM'):
    model = create_model(cell_type)
    start = time.time()
    history = model.fit(x_train, y_train,
                        epochs=3,
                        batch_size=128,
                        validation_split=0.2,
                        verbose=2)
    end = time.time()
    duration = end - start
    test_loss, test_acc = model.evaluate(x_test, y_test, verbose=0)
    return duration, test_acc, history

# обучение и сравнение
lstm_time, lstm_acc, lstm_hist = train_and_evaluate('LSTM')
gru_time, gru_acc, gru_hist = train_and_evaluate('GRU')

print(f"\nLSTM: Time = {lstm_time:.2f}s, Accuracy = {lstm_acc:.4f}")
print(f"GRU:  Time = {gru_time:.2f}s, Accuracy = {gru_acc:.4f}")
```

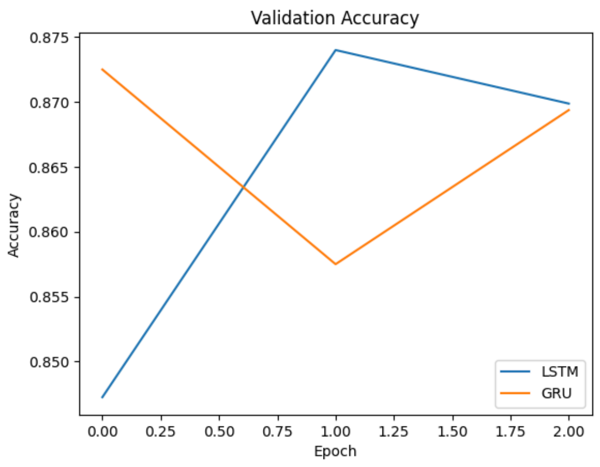
- Train them on the same dataset and compare training time and accuracy.

We display accuracy, time and visualize data on graphs

```
plt.plot(lstm_hist.history['val_accuracy'], label='LSTM')
plt.plot(gru_hist.history['val_accuracy'], label='GRU')
plt.title("Validation Accuracy")
plt.xlabel("Epoch")
plt.ylabel("Accuracy")
plt.legend()
plt.show()
```

From the output data, we can see that both models did an excellent job - the accuracy is almost the same, and above 87%, in terms of timing, LSTM is slightly faster.
Conclusion
If simplicity and efficiency are important, GRU is a great choice.
If long-term memory and working with a more complex context are important, you should choose LSTM.

```
Path to dataset files: /kaggle/input/imdb-dataset-of-50k-movie-reviews
Epoch 1/3
/usr/local/lib/python3.11/dist-packages/keras/src/layers/core/embedding.py:90: UserWarning: Argument `input_length` is deprecated. Just remove it.
  warnings.warn(
250/250 - 112s - 447ms/step - accuracy: 0.8089 - loss: 0.4085 - val_accuracy: 0.8472 - val_loss: 0.3527
Epoch 2/3
250/250 - 111s - 445ms/step - accuracy: 0.9043 - loss: 0.2439 - val_accuracy: 0.8740 - val_loss: 0.3009
Epoch 3/3
250/250 - 140s - 561ms/step - accuracy: 0.9258 - loss: 0.1977 - val_accuracy: 0.8699 - val_loss: 0.3172
Epoch 1/3
250/250 - 123s - 492ms/step - accuracy: 0.7765 - loss: 0.4521 - val_accuracy: 0.8725 - val_loss: 0.3117
Epoch 2/3
250/250 - 142s - 566ms/step - accuracy: 0.8954 - loss: 0.2624 - val_accuracy: 0.8575 - val_loss: 0.3316
Epoch 3/3
250/250 - 144s - 574ms/step - accuracy: 0.9218 - loss: 0.2051 - val_accuracy: 0.8694 - val_loss: 0.3217

LSTM: Time = 395.80s, Accuracy = 0.8723
GRU:  Time = 408.77s, Accuracy = 0.8740
```

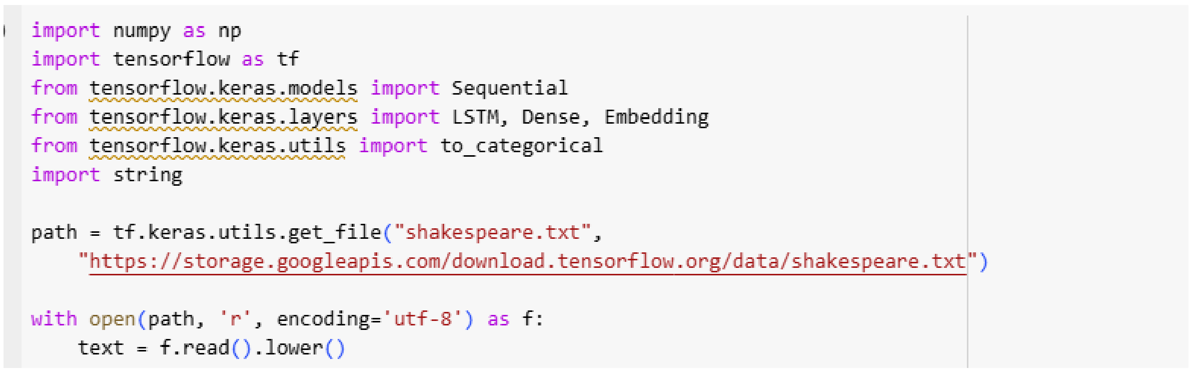


- *Figure 4: Training loss comparison between LSTM and GRU*

Exercise 7: Implementing LSTMs for Text Generation and Classification

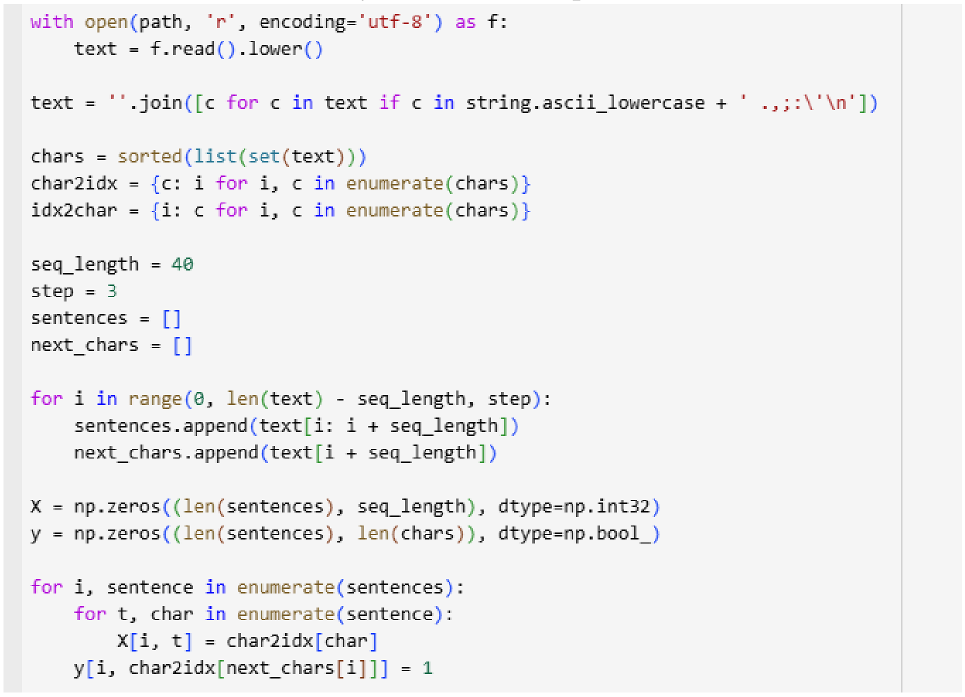
- Train an **LSTM-based text generation model** on a dataset (e.g., Shakespeare text).

Import necessary libraries and dataset Shakespeare text

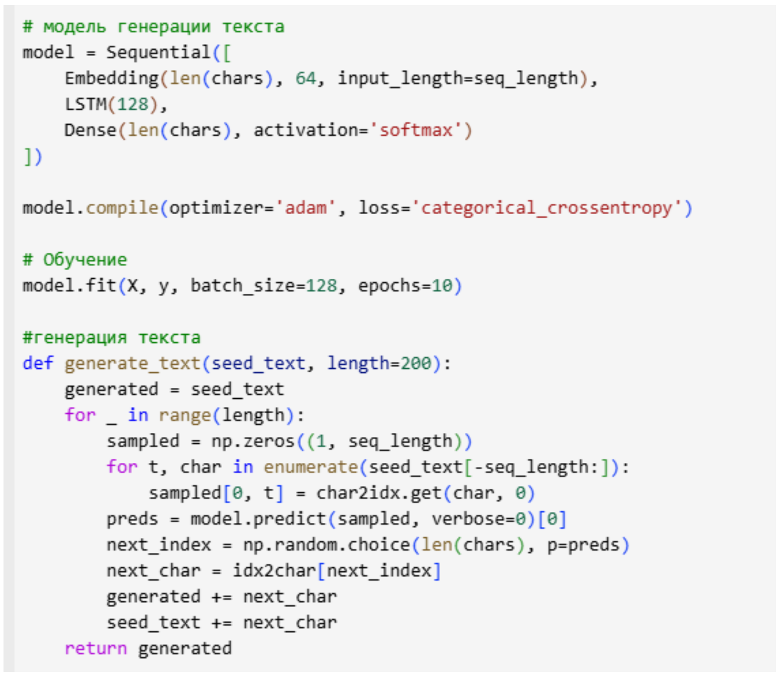


```python
import numpy as np
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense, Embedding
from tensorflow.keras.utils import to_categorical
import string

path = tf.keras.utils.get_file("shakespeare.txt",
    "https://storage.googleapis.com/download.tensorflow.org/data/shakespeare.txt")

with open(path, 'r', encoding='utf-8') as f:
    text = f.read().lower()
```

Next we remove unnecessary characters, set parameters and convert the number format



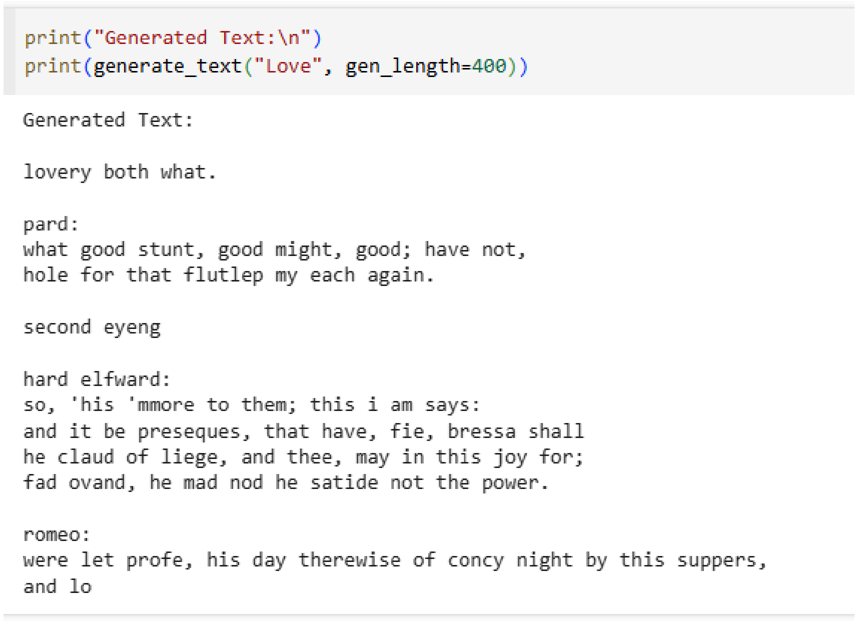
```python
with open(path, 'r', encoding='utf-8') as f:
    text = f.read().lower()

text = ''.join([c for c in text if c in string.ascii_lowercase + ' .,;:\'\n'])

chars = sorted(list(set(text)))
char2idx = {c: i for i, c in enumerate(chars)}
idx2char = {i: c for i, c in enumerate(chars)}

seq_length = 40
step = 3
sentences = []
next_chars = []

for i in range(0, len(text) - seq_length, step):
    sentences.append(text[i: i + seq_length])
    next_chars.append(text[i + seq_length])

X = np.zeros((len(sentences), seq_length), dtype=np.int32)
y = np.zeros((len(sentences), len(chars)), dtype=np.bool_)

for i, sentence in enumerate(sentences):
    for t, char in enumerate(sentence):
        X[i, t] = char2idx[char]
    y[i, char2idx[next_chars[i]]] = 1
```



```python
# модель генерации текста
model = Sequential([
    Embedding(len(chars), 64, input_length=seq_length),
    LSTM(128),
    Dense(len(chars), activation='softmax')
])

model.compile(optimizer='adam', loss='categorical_crossentropy')

# Обучение
model.fit(X, y, batch_size=128, epochs=10)

#генерация текста
def generate_text(seed_text, length=200):
    generated = seed_text
    for _ in range(length):
        sampled = np.zeros((1, seq_length))
        for t, char in enumerate(seed_text[-seq_length:]):
            sampled[0, t] = char2idx.get(char, 0)
        preds = model.predict(sampled, verbose=0)[0]
        next_index = np.random.choice(len(chars), p=preds)
        next_char = idx2char[next_index]
        generated += next_char
        seed_text += next_char
    return generated
```

- Generate text samples and analyze the results.

We enter any word or sentence and the model generates a test based on Shakespearean poems.



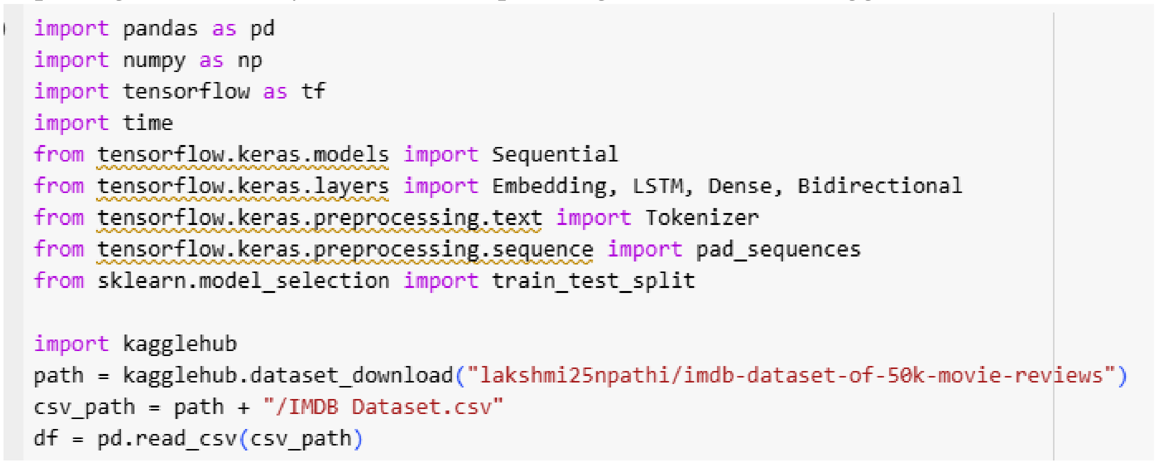
```python
print("Generated Text:\n")
print(generate_text("Love", gen_length=400))
```

```
Generated Text:

lovery both what.

pard:
what good stunt, good might, good; have not,
hole for that flutlep my each again.

second eyeng

hard elfward:
so, 'his 'mmore to them; this i am says:
and it be preseques, that have, fie, bressa shall
he claud of liege, and thee, may in this joy for;
fad ovand, he mad nod he satide not the power.

romeo:
were let profe, his day therewise of concy night by this suppers,
and lo
```

Exercise 8: Improving Performance with Bidirectional LSTMs

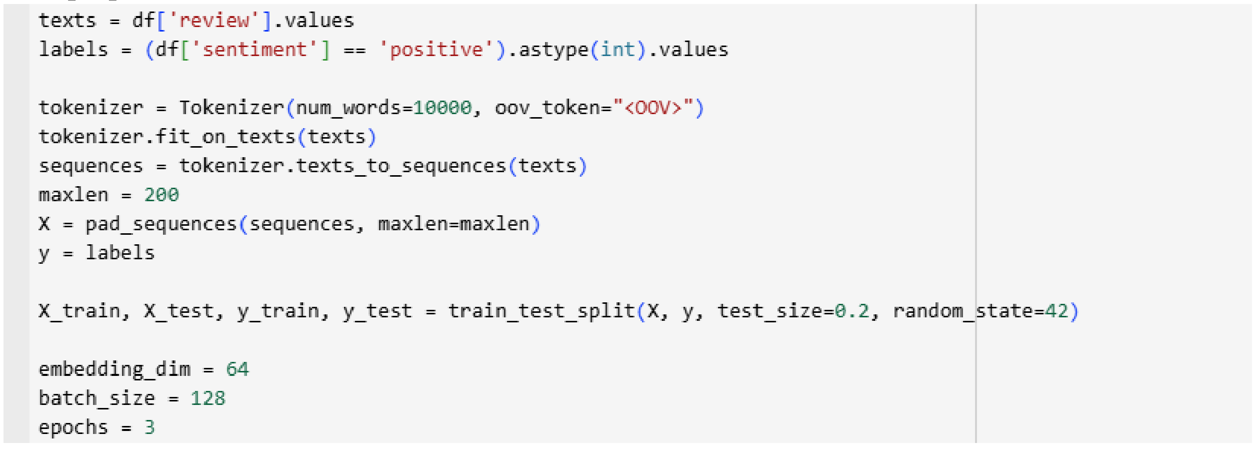
- Modify an existing LSTM model to use **Bidirectional LSTM**.

Importing the necessary libraries and uploading a dataset from Kaggle



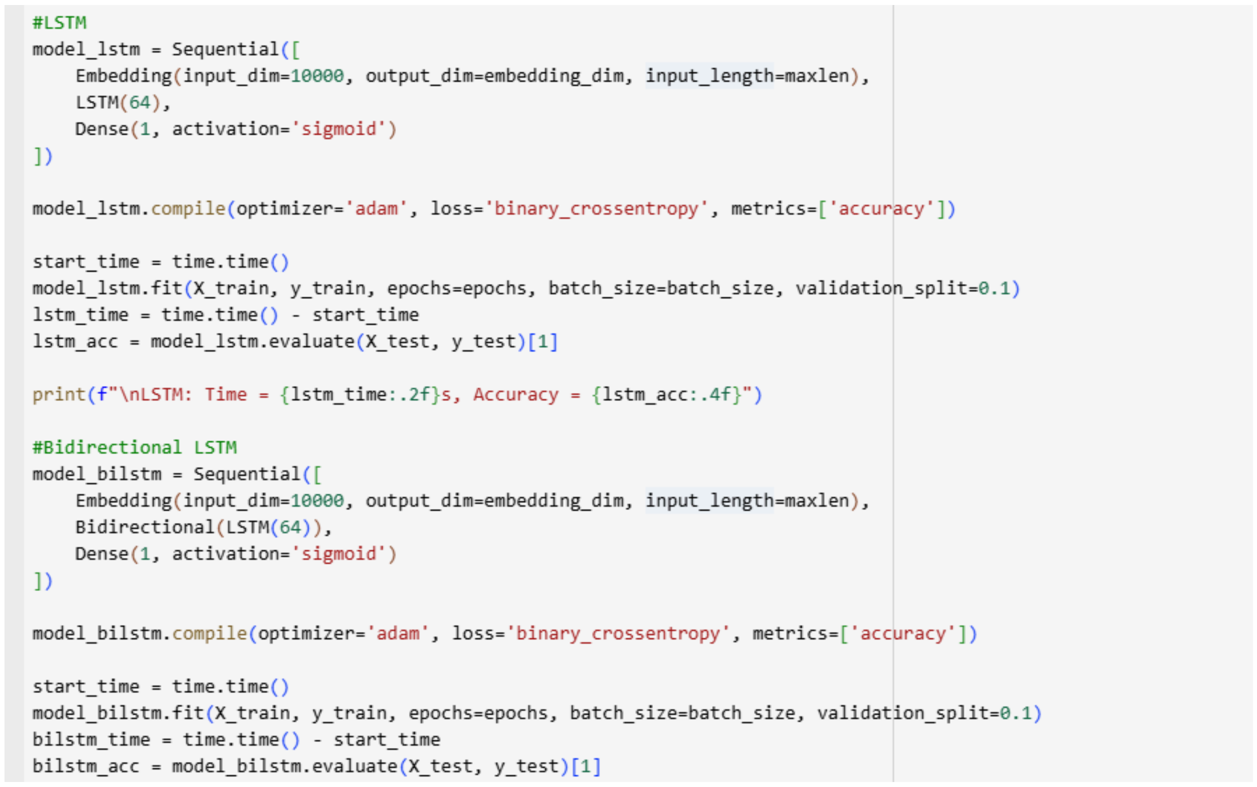
```python
import pandas as pd
import numpy as np
import tensorflow as tf
import time
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Embedding, LSTM, Dense, Bidirectional
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
from sklearn.model_selection import train_test_split

import kagglehub
path = kagglehub.dataset_download("lakshmi25npathi/imdb-dataset-of-50k-movie-reviews")
csv_path = path + "/IMDB Dataset.csv"
df = pd.read_csv(csv_path)
```

Data preparation and tokenization



```python
texts = df['review'].values
labels = (df['sentiment'] == 'positive').astype(int).values

tokenizer = Tokenizer(num_words=10000, oov_token="<OOV>")
tokenizer.fit_on_texts(texts)
sequences = tokenizer.texts_to_sequences(texts)
maxlen = 200
X = pad_sequences(sequences, maxlen=maxlen)
y = labels

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

embedding_dim = 64
batch_size = 128
epochs = 3
```

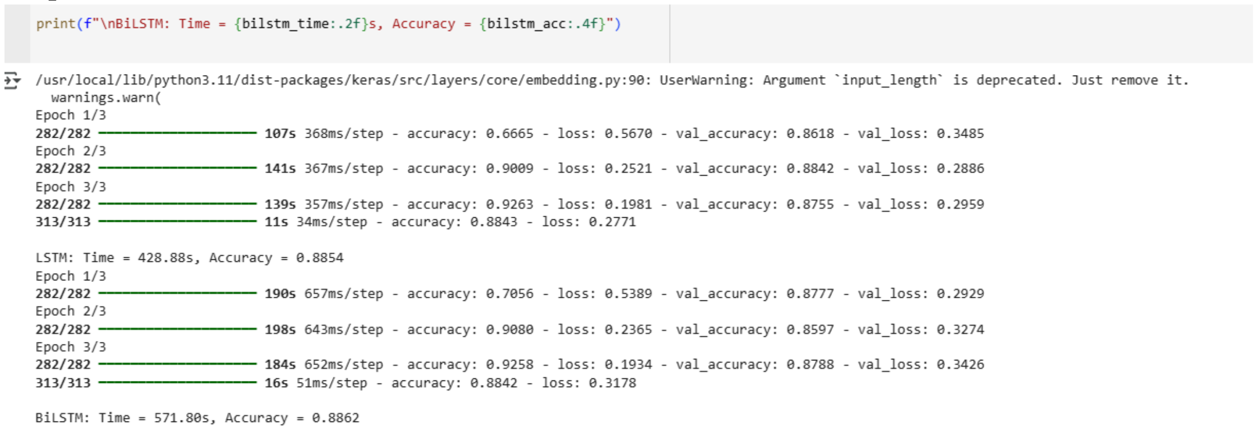
We create models LSTM and Bidirectional LSTM and train them on a dataset



```python
#LSTM
model_lstm = Sequential([
    Embedding(input_dim=10000, output_dim=embedding_dim, input_length=maxlen),
    LSTM(64),
    Dense(1, activation='sigmoid')
])

model_lstm.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])

start_time = time.time()
model_lstm.fit(X_train, y_train, epochs=epochs, batch_size=batch_size, validation_split=0.1)
lstm_time = time.time() - start_time
lstm_acc = model_lstm.evaluate(X_test, y_test)[1]

print(f"\nLSTM: Time = {lstm_time:.2f}s, Accuracy = {lstm_acc:.4f}")

#Bidirectional LSTM
model_bilstm = Sequential([
    Embedding(input_dim=10000, output_dim=embedding_dim, input_length=maxlen),
    Bidirectional(LSTM(64)),
    Dense(1, activation='sigmoid')
])

model_bilstm.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])

start_time = time.time()
model_bilstm.fit(X_train, y_train, epochs=epochs, batch_size=batch_size, validation_split=0.1)
bilstm_time = time.time() - start_time
bilstm_acc = model_bilstm.evaluate(X_test, y_test)[1]
```

- Compare results with standard LSTM in terms of accuracy and training efficiency.

To compare the models, we display the time and accuracy
Output:



```python
print(f"\nBiLSTM: Time = {bilstm_time:.2f}s, Accuracy = {bilstm_acc:.4f}")
```

```
/usr/local/lib/python3.11/dist-packages/keras/src/layers/core/embedding.py:90: UserWarning: Argument `input_length` is deprecated. Just remove it.
  warnings.warn(
Epoch 1/3
282/282 ━━━━━━━━━━━━━━━━━━━━ 107s 368ms/step - accuracy: 0.6665 - loss: 0.5670 - val_accuracy: 0.8618 - val_loss: 0.3485
Epoch 2/3
282/282 ━━━━━━━━━━━━━━━━━━━━ 141s 367ms/step - accuracy: 0.9009 - loss: 0.2521 - val_accuracy: 0.8842 - val_loss: 0.2886
Epoch 3/3
282/282 ━━━━━━━━━━━━━━━━━━━━ 139s 357ms/step - accuracy: 0.9263 - loss: 0.1981 - val_accuracy: 0.8755 - val_loss: 0.2959
313/313 ━━━━━━━━━━━━━━━━━━━━ 11s 34ms/step - accuracy: 0.8843 - loss: 0.2771

LSTM: Time = 428.88s, Accuracy = 0.8854
Epoch 1/3
282/282 ━━━━━━━━━━━━━━━━━━━━ 190s 657ms/step - accuracy: 0.7056 - loss: 0.5389 - val_accuracy: 0.8777 - val_loss: 0.2929
Epoch 2/3
282/282 ━━━━━━━━━━━━━━━━━━━━ 198s 643ms/step - accuracy: 0.9080 - loss: 0.2365 - val_accuracy: 0.8597 - val_loss: 0.3274
Epoch 3/3
282/282 ━━━━━━━━━━━━━━━━━━━━ 184s 652ms/step - accuracy: 0.9258 - loss: 0.1934 - val_accuracy: 0.8788 - val_loss: 0.3426
313/313 ━━━━━━━━━━━━━━━━━━━━ 16s 51ms/step - accuracy: 0.8842 - loss: 0.3178

BiLSTM: Time = 571.80s, Accuracy = 0.8862
```

4. Results and Discussion

When comparing different approaches to text representation and sequence modeling, it becomes clear that traditional methods like One-hot and TF-IDF are suitable starting points for simple tasks but fail to capture semantic and contextual relationships between words. In contrast, distributed representations such as Word2Vec and FastText provide a deeper understanding of language through dense vectors and context awareness. Moving to sequence models, the advantages of LSTM and GRU over basic RNNs are evident—they handle long-term dependencies more effectively and offer more stable training. GRU tends to be faster and more resource-efficient, while LSTM can be slightly more accurate in tasks involving long-range context.

For a visual comparison, I created a table where we can see the accuracy and time of the models.

Model	**Accuracy**	**Training Time (s)**
LSTM	~87%	395.80
GRU	~87%	408.77
Bidirectional LSTM	~88%	571.80

5. Conclusion

In conclusion, this project demonstrated the evolution of natural language processing techniques from basic text representations to advanced sequence modeling architectures. Key findings include:

- TF-IDF offers interpretable word importance, useful for simple models.
- Word2Vec and FastText provide rich semantic embeddings, essential for deep NLP tasks.
- FastText is superior in handling subwords and rare words.
- RNNs, while foundational, suffer from vanishing gradients and memory limitations.
- LSTM and GRU significantly improve text modeling performance—GRU being computationally lighter.
- Bidirectional LSTM (not shown) further enhances performance in many applications by incorporating context from both directions.

These experiments highlight the importance of choosing the right representation and model architecture based on task complexity and resource constraints.

5. References

1. Word Embeddings in NLP | GeeksforGeeks
2. Chapter 4 Recurrent neural networks and their applications in NLP | Modern Approaches in Natural Language Processing
3. Long Short Term Memory(LSTM) and Gated Recurrent Units(GRU) | by Parveen Khurana | Medium
4. Long Short Term Memory(LSTM) and Gated Recurrent Units(GRU) | by Parveen Khurana | Medium