APPLIED AI

PRESENTATION

APPLICATIONS

ABOUT PROJECT

The goal of this project is to build a binary classification model that determines whether a movie review is positive or negative.

THE IMDB MOVIE REVIEWS DATASET, WIDELY USED IN SENTIMENT ANALYSIS TASKS, IS USED AS DATA.

This laptop performs exploratory data analysis (EDA) with the following objectives:

studying the structure of the dataset
checking data quality
analyzing class distribution
analyzing text length
justifying data preprocessing steps for LSTM and BiLSTM models

EXPLOROTARY DA

LOAD DATA

```
df = pd.read_csv("C:\\Users\\nurs\\OneDrive\\Рабочий стол\\IMDB Dataset.csv")
df.head()
[3]
```

	review	sentiment
0	One of the other reviewers has mentioned that ...	positive
1	A wonderful little production. The...	positive
2	I thought this was a wonderful way to spend ti...	positive
3	Basically there's a family where a little boy ...	negative
4	Petter Mattei's "Love in the Time of Money" is...	positive

```
df.info()
[4]
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 50000 entries, 0 to 49999
Data columns (total 2 columns):
 #   Column     Non-Null Count  Dtype
---  ------     --------------  -----
 0   review     50000 non-null  object
 1   sentiment  50000 non-null  object
dtypes: object(2)
memory usage: 781.4+ KB
```

50,000 objects and 1 feature (review), with target variable sentiment (positive/negative)

ANALYSIS OF MISSING VALUES AND DUPLICATES

```
df.isnull().sum()
[5]
```

```
review       0
sentiment    0
dtype: int64
```

```
df.duplicated().sum()
[6]
```

```
np.int64(418)
```

```
df = df.drop_duplicates()
✓ [5] 243ms
```

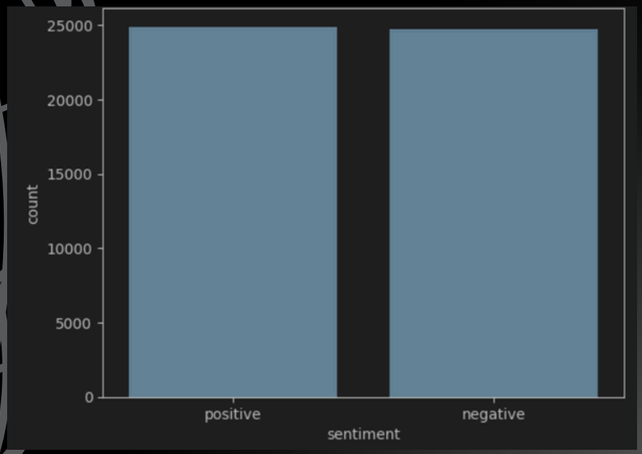
There are no missing values in the dataset. Duplicates were found and removed.

CLASS DISTRIBUTION

```
df["sentiment"].value_counts(normalize=True)
[8]
```

```
sentiment
positive    0.501876
negative    0.498124
Name: proportion, dtype: float64
```

```
sns.countplot(x="sentiment", data=df)
plt.title("Распределение классов")
plt.show()
```



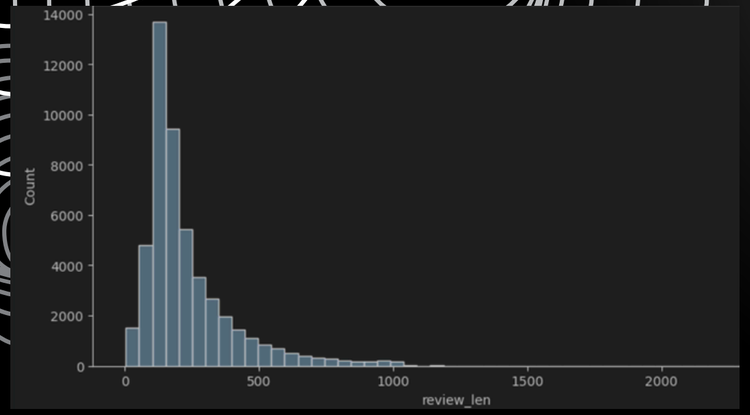
The dataset is balanced:
the number of positive and negative reviews is approximately equal.

ANALYSIS OF REVIEW LENGTH

```
df["review_len"] = df["review"].apply(lambda x: len(x.split()))
df["review_len"].describe()
```

```
count    49582.000000
mean       231.350167
std        171.542020
min          4.000000
25%        126.000000
50%        173.000000
75%        281.000000
max       2470.000000
Name: review_len, dtype: float64
```

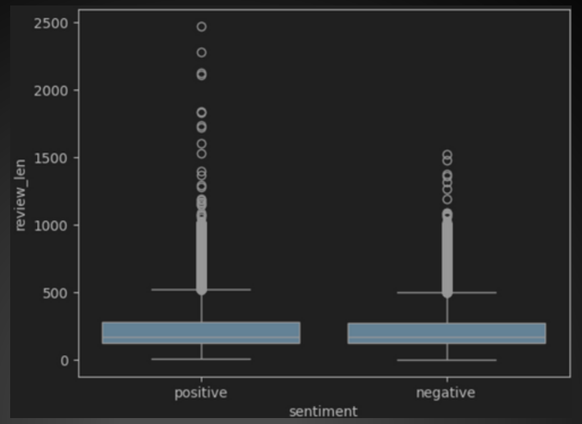
```
plt.figure(figsize=(10,5))
sns.histplot(df["review_len"], bins=50)
plt.title("Распределение длины отзывов (в словах)")
plt.show()
```



The distribution of review lengths has a long tail. Most reviews contain fewer than 400 words,
while a small number of reviews are significantly longer.

LENGTH OF REVIEWS BY CLASS

```
sns.boxplot(x="sentiment", y="review_len", data=df)
plt.title("Длина отзывов по классам")
plt.show()
```



The distributions of review lengths for positive and negative classes are practically identical.
This suggests that text length alone does not determine the tone of a review.

FREQUENCY ANALYSIS OF WORDS

```
words = " ".join(df["review"]).lower().split()
Counter(words).most_common(20)
```

```
[('the', 633981),
 ('a', 314408),
 ('and', 311378),
 ('of', 284608),
 ('to', 262599),
 ('is', 203488),
 ('in', 178487),
 ('i', 140393),
 ('this', 137398),
 ('that', 129179),
 ('it', 128655),
 ('/><br', 100211),
 ('was', 92494),
```

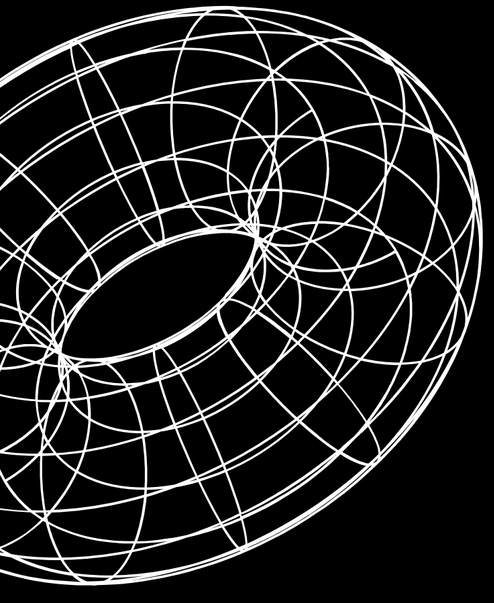
The most common are stop words and HTML-related tokens. HTML tags will be removed during preprocessing, while stop words are retained to preserve context, which is especially important for recurrent neural networks.

TEXT CONTENT ANALYSIS

```
df[df["sentiment"] == "positive"]["review"].iloc[0]
df[df["sentiment"] == "negative"]["review"].iloc[0]
```

```
"Basically there's a family where a little boy (Jake) thinks there's a zombie in his closet & his parents are fighting all the time.<br /><br />This movie is slower than a soap opera... and suddenly, Jake decides to become Rambo and kill the zombie.<br /><br />OK, first of all when you're going to make a film you must Decide if its a thriller or a drama! As a drama the movie is watchable. Parents are divorcing & arguing like in real life. And then we have Jake with his closet which totally ruins all the film! I expected to see a BOOGEYMAN similar movie, and instead i watched a drama with some meaningless thriller spots.<br /><br />3 out of 10 just for the well playing parents & descent dialogs. As for the shots with Jake: just ignore them."
```

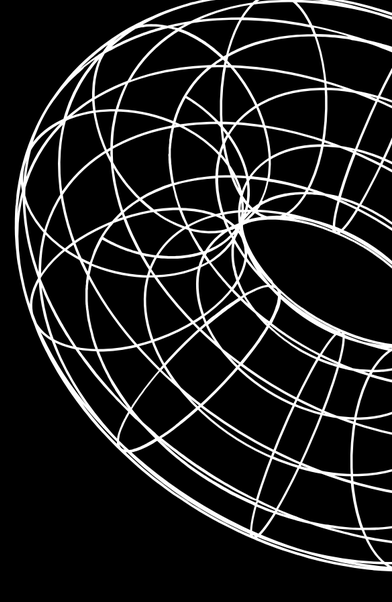
Manual analysis of texts shows:
the presence of HTML tags
the use of punctuation
colloquial style and abbreviations This confirms the need to clean up the text before tokenization.



FINAL CONCLUSION ON EDA

During exploratory data analysis, it was determined that:

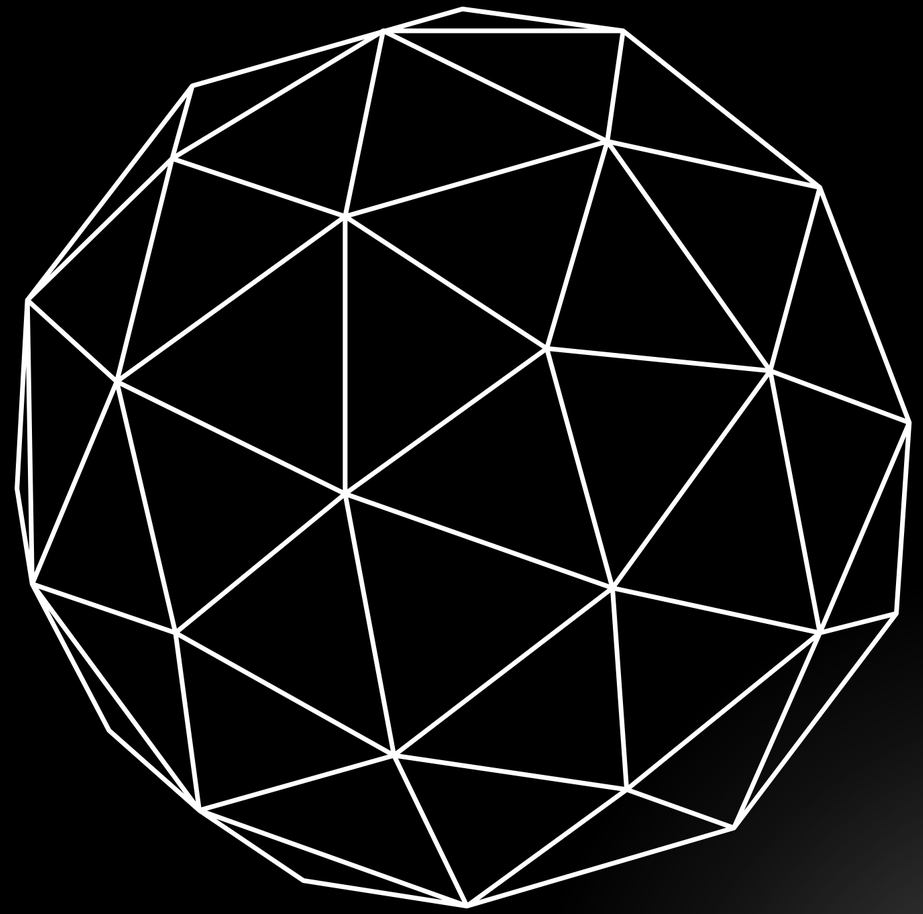
the dataset is clean and balanced
there are no missing values
duplicates have been removed
the length of the texts has a long-tail distribution
truncation and supplementation of sequences are necessary
the length of the review is not a determining factor for tone
the presence of HTML tags requires preliminary text cleaning
The results obtained justify the use of LSTM and BiLSTM models
for the task of classifying the tone of texts.



DATA PREPROCESSING

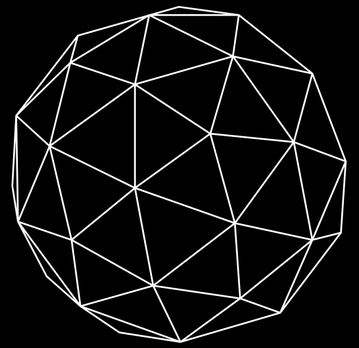
The purpose of data preprocessing

The goal of this stage is to convert text reviews into
a numerical format suitable for training an LSTM-based neural network model.



During preprocessing, we:

clean the text of noise
tokenize reviews
build a dictionary
encode words into indices
align sequence lengths
prepare data for PyTorch



TARGET VARIABLE TRANSFORMATION

```
df["label"] = df["sentiment"].map({"negative": 0, "positive": 1})
print(df["label"].value_counts())
df[["sentiment", "label"]].head()
```

```
label
1    25000
0    25000
Name: count, dtype: int64
```

	sentiment	label
0	positive	1
1	positive	1
2	positive	1
3	negative	0
4	positive	1

To train the model, the target variable is converted to a numerical format:

1 — positive review
0 — negative review
The dataset is completely balanced.

BUILDING A DICTIONARY

```
VOCAB_SIZE = 20_000
PAD_TOKEN = "<PAD>"
UNK_TOKEN = "<UNK>"
all_tokens = []
for tokens in df["tokens"]:
    all_tokens.extend(tokens)
token_freqs = Counter(all_tokens)
most_common_tokens = token_freqs.most_common(VOCAB_SIZE - 2)
word2idx = {
    PAD_TOKEN: 0,
    UNK_TOKEN: 1
}

for idx, (word, _) in enumerate(most_common_tokens, start=2):
    word2idx[word] = idx
print("Размер словаря:", len(word2idx))
```

The dictionary is based on word frequency:

uses the 20,000 most common words
used for padding
used for unknown words

BASIC TEXT CLEANING

```
def clean_text_basic(text: str):
    text = re.sub(r"<.*?>", " ", text)
    text = text.lower()
    text = re.sub(r"[^a-z0-9\s]", " ", text)
    text = re.sub(r"\s+", " ", text).strip()
    return text
df["review_clean"] = df["review"].apply(clean_text_basic)
df[["review", "review_clean"]].head(2)
```

	review	review_clean
0	One of the other reviewers has mentioned that ...	one of the other reviewers has mentioned that ...
1	A wonderful little production. The...	a wonderful little production the filming tech...

At this stage:

HTML tags are removed
text is converted to lowercase
spaces are normalized
Aggressive cleaning (removal of stop words, lemmatization) is not applied, as it can degrade the quality of LSTM models.

TEXT TOKENIZATION

```
def tokenize_text(text: str):
    return word_tokenize(text)
df["tokens"] = df["review_clean"].apply(tokenize_text)
df[["review_clean", "tokens"]].head(2)
```

	review_clean	tokens
0	one of the other reviewers has mentioned that ...	[one, of, the, other, reviewers, has, mentione...
1	a wonderful little production the filming tech...	[a, wonderful, little, production, the, filmin...

Tokenization is performed using the NLTK library
. Each review is converted into a list of words (tokens), which is the optimal input for LSTM.

TEXT ENCODING

```
def encode_tokens(tokens, word2idx):
    return [
        word2idx.get(token, word2idx["<UNK>"])
        for token in tokens
    ]

df["encoded"] = df["tokens"].apply(lambda x: encode_tokens(x, word2idx))
df[["tokens", "encoded"]].head(2)
```

	tokens	encoded
0	[one, of, the, other, reviewers, has, mentione...	[29, 5, 2, 78, 2062, 47, 1063, 12, 101, 150, 4...
1	[a, wonderful, little, production, the, filmin...	[4, 395, 121, 355, 2, 1383, 2983, 7, 54, 17699...

Each word is replaced with the corresponding numerical index from the dictionary.
Words that are not in the dictionary are encoded as .

PADDING AND TRUNCATION

```
MAX_LEN = 400
PAD_IDX = word2idx["<PAD>"]
def pad_truncate(sequence, max_len, pad_idx):
    if len(sequence) > max_len:
        return sequence[:max_len]
    else:
        return sequence + [pad_idx] * (max_len - len(sequence))
df["padded"] = df["encoded"].apply(
    lambda x: pad_truncate(x, MAX_LEN, PAD_IDX)
)
df["padded"].apply(len).value_counts().head()
```

```
padded
400    50000
Name: count, dtype: int64
```

All sequences are reduced to a fixed length of 400:

long reviews are truncated
short reviews are padded
This is necessary for batch training of the neural network.

DATA SEPARATION

```
X = df["padded"].tolist()
y = df["label"].tolist()
X_train, X_temp, y_train, y_temp = train_test_spl
    X,
    y,
    test_size=0.30,
    stratify=y,
    random_state=42
)
[24]
```

```
X_val, X_test, y_val, y_test = train_test_split(
    X_temp,
    y_temp,
    test_size=0.50,
    stratify=y_temp,
    random_state=42
)
```

The following split is used:

70% — training
15% — validation
15% — test
Class balance is maintained using stratify.

```
print("Train size:", len(X_train))
print("Val size:", len(X_val))
print("Test size:", len(X_test))
[26]
```

```
Train size: 35000
Val size: 7500
Test size: 7500
```

```
class IMDBDataset(Dataset):
    def __init__(self, X, y):
        self.X = X
        self.y = y
    def __len__(self):
        return len(self.X)
    def __getitem__(self, idx):
        x = torch.tensor(self.X[idx], dtype=torch.long)
        y = torch.tensor(self.y[idx], dtype=torch.float)
        return x, y
train_dataset = IMDBDataset(X_train, y_train)
val_dataset   = IMDBDataset(X_val, y_val)
test_dataset  = IMDBDataset(X_test, y_test)
len(train_dataset), len(val_dataset), len(test_dataset)
[28]
```

```
(35000, 7500, 7500)
```

```
BATCH_SIZE = 64
train_loader = DataLoader(
    train_dataset,
    batch_size=BATCH_SIZE,
    shuffle=True
)

val_loader = DataLoader(
    val_dataset,
    batch_size=BATCH_SIZE,
    shuffle=False
)

test_loader = DataLoader(
    test_dataset,
    batch_size=BATCH_SIZE,
    shuffle=False
)
batch_X, batch_y = next(iter(train_loader))
batch_X.shape, batch_y.shape

[30]
```

```
(torch.Size([64, 400]), torch.Size([64]))
```

```
idx2word = {idx: word for word, idx in word2idx.items()}
```

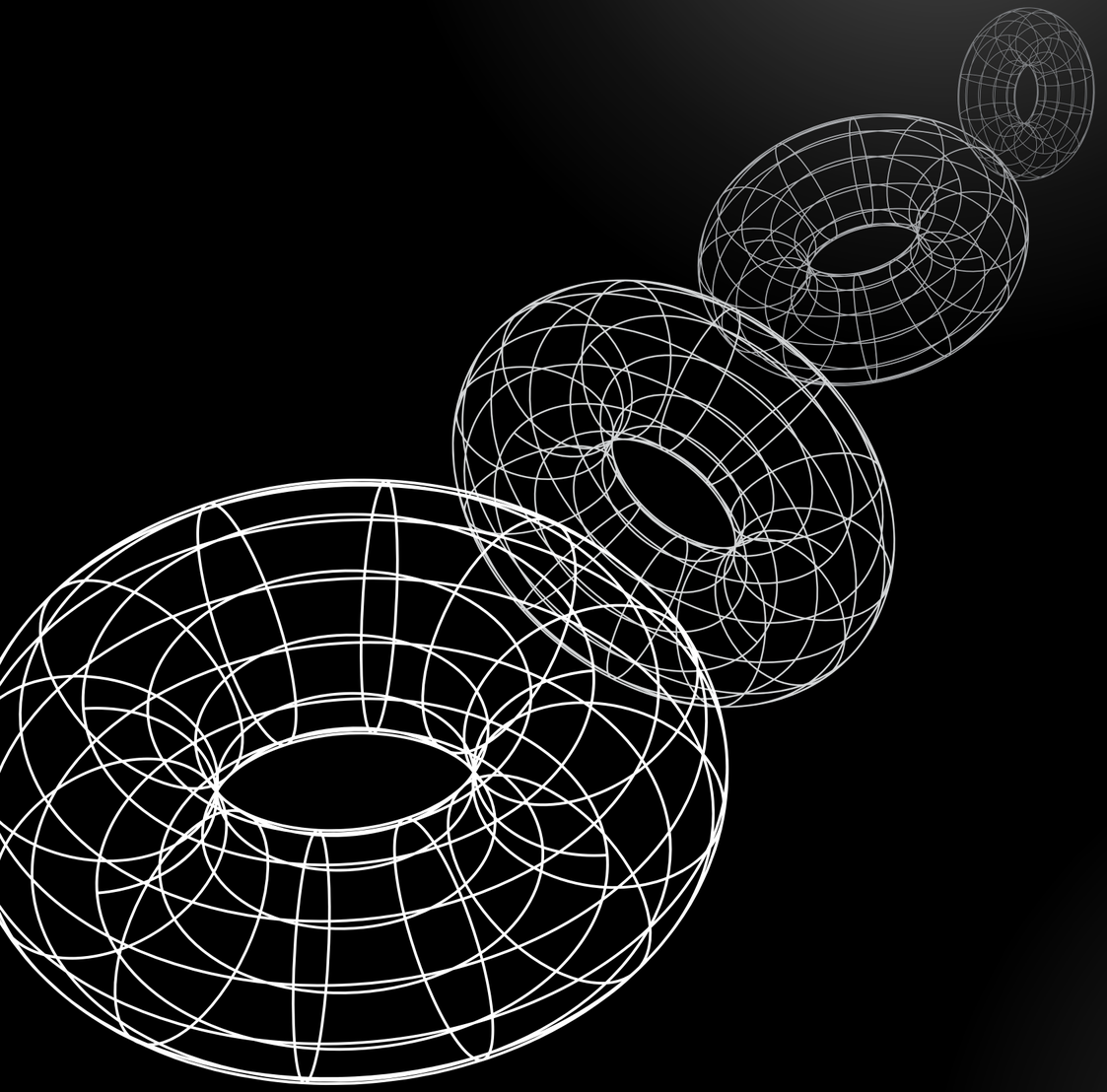
DATA RETENTION

```
SAVE_DIR = Path("artifacts")
SAVE_DIR.mkdir(exist_ok=True)
with open(SAVE_DIR / "word2idx.pkl", "wb") as f:
    pickle.dump(word2idx, f)
with open(SAVE_DIR / "idx2word.pkl", "wb") as f:
    pickle.dump(idx2word, f)
```

```
torch.save(
    {
        "X_train": torch.tensor(X_train, dtype=torch.long),
        "y_train": torch.tensor(y_train, dtype=torch.float),
        "X_val":   torch.tensor(X_val, dtype=torch.long),
        "y_val":   torch.tensor(y_val, dtype=torch.float),
        "X_test":  torch.tensor(X_test, dtype=torch.long),
        "y_test":  torch.tensor(y_test, dtype=torch.float),
    },
    SAVE_DIR / "dataset.pt"
)
[35]
```

```
len(word2idx), word2idx["<PAD>"], word2idx["<UNK>"]
[36]
```

```
(20000, 0, 1)
```



BASELINE MODEL

Training stage objective
The objective of this stage is to train a basic LSTM model for binary classification of reviews into positive and negative.

This model is used as a baseline against which improved architectures will be compared in the future.

LOADING DATA AND DICTIONARY

```
ARTIFACTS_DIR = Path("artifacts")
data = torch.load(ARTIFACTS_DIR / "dataset.pt")
X_train = data["X_train"]
y_train = data["y_train"]
X_val = data["X_val"]
y_val = data["y_val"]
X_test = data["X_test"]
y_test = data["y_test"]
with open(ARTIFACTS_DIR / "word2idx.pkl", "rb") as f:
    word2idx = pickle.load(f)
VOCAB_SIZE = len(word2idx)
X_train.shape, y_train.shape
```

```
(torch.Size([35000, 400]), torch.Size([35000]))
```

Loading data and dictionary
At this stage, we use the dataset that was prepared and saved during the Data Preprocessing stage. The text is not processed again.

PREPARING DATASET AND DATALOADER

```
class IMDBTensorDataset(Dataset):
    def __init__(self, X, y):
        self.X = X
        self.y = y
    def __len__(self):
        return self.X.size(0)
    def __getitem__(self, idx):
        return self.X[idx], self.y[idx]
```

```
BATCH_SIZE = 64
train_dataset = IMDBTensorDataset(X_train, y_train)
val_dataset   = IMDBTensorDataset(X_val, y_val)

train_loader = DataLoader(
    train_dataset,
    batch_size=BATCH_SIZE,
    shuffle=True
)
val_loader = DataLoader(
    val_dataset,
    batch_size=BATCH_SIZE,
    shuffle=False
)
```

```
batch_X, batch_y = next(iter(train_loader))
batch_X.shape, batch_y.shape
[6]
```

```
(torch.Size([64, 400]), torch.Size([64]))
```

DataLoader provides:

batch training
data shuffling during training
correct data feeding into the model

We build software that empowers organizations to effectively integrate their data, decisions, and operations.

ARCHITECTURE OF THE BASIC LSTM MODEL

```python
class LSTMClassifier(nn.Module):
    def __init__(
        self,
        vocab_size,
        embedding_dim,
        hidden_dim,
        padding_idx
    ):
        super().__init__()

        # 1) Embedding слой
        self.embedding = nn.Embedding(
            num_embeddings=vocab_size,
            embedding_dim=embedding_dim,
            padding_idx=padding_idx
        )
        # 2) LSTM слой
        self.lstm = nn.LSTM(
            input_size=embedding_dim,
            hidden_size=hidden_dim,
            batch_first=True
        )
```

```python
    # 3) Выходной слой
    self.fc = nn.Linear(hidden_dim, 1)
def forward(self, x):
    """
    x: (batch_size, seq_len)
    """
    # (batch, seq) → (batch, seq, embed)
    embedded = self.embedding(x)
    # LSTM
    # hidden: (num_layers, batch, hidden_dim)
    _, (hidden, _) = self.lstm(embedded)
    # берём последнее скрытое состояние
    hidden = hidden.squeeze(0)
    # (batch, hidden_dim) → (batch, 1)
    logits = self.fc(hidden)
    return logits.squeeze(1)
```

```python
PAD_IDX = 0
EMBEDDING_DIM = 128
HIDDEN_DIM = 128
model = LSTMClassifier(
    vocab_size=VOCAB_SIZE,
    embedding_dim=EMBEDDING_DIM,
    hidden_dim=HIDDEN_DIM,
    padding_idx=PAD_IDX
)
model
[8]
```

```
LSTMClassifier(
  (embedding): Embedding(20000, 128, padding_idx=0)
  (lstm): LSTM(128, 128, batch_first=True)
  (fc): Linear(in_features=128, out_features=1, bias=True)
)
```

The model consists of:

an embedding layer
a single LSTM layer
a linear classifier
The model output is logits used for binary classification.

```python
with torch.no_grad():
    logits = model(batch_X)
logits.shape
[11]
```

```
torch.Size([64])
```

LOSS FUNCTION AND OPTIMIZER

```python
criterion = nn.BCEWithLogitsLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model = model.to(device)
model.train()
batch_X, batch_y = next(iter(train_loader))
batch_X = batch_X.to(device)
batch_y = batch_y.to(device)
optimizer.zero_grad()
logits = model(batch_X)
loss = criterion(logits, batch_y)
loss.backward()
optimizer.step()
loss.item()
[12]
```

```
0.691602885723114
```

The BCEWithLogitsLoss loss function is used, which is standard for binary classification and numerically more stable than sigmoid + BCELoss.

TRAINING CYCLE

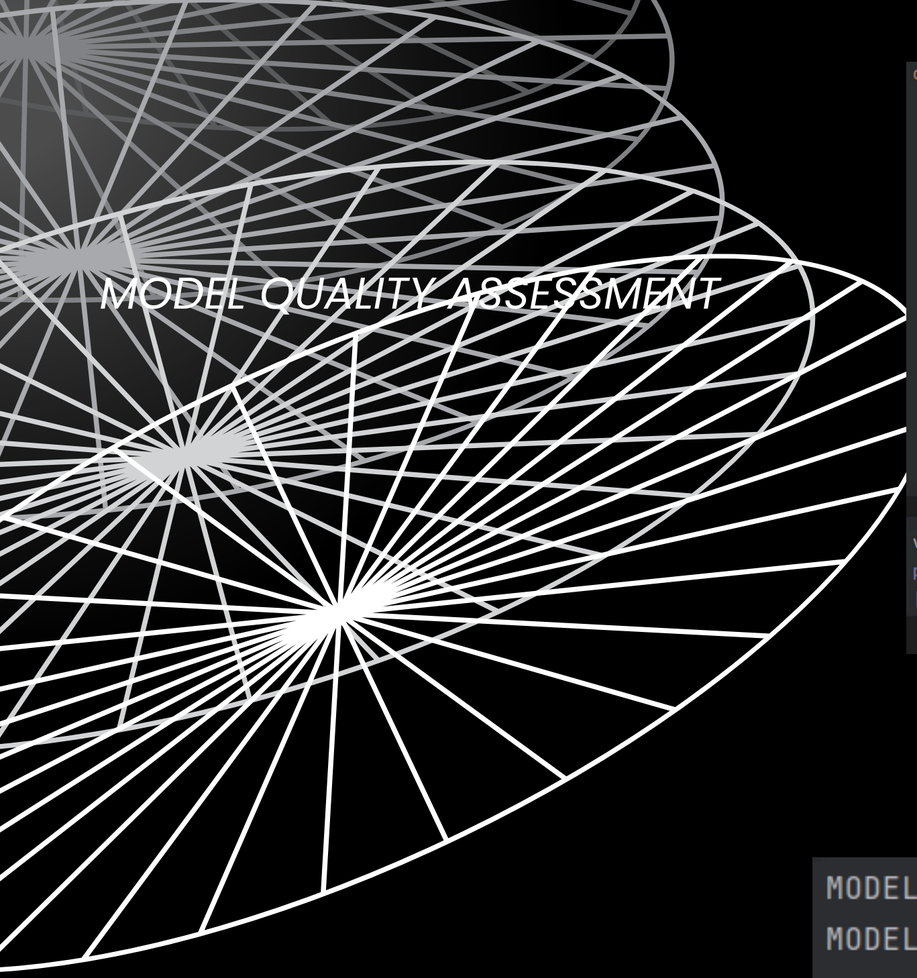
```python
def train_one_epoch(model, dataloader, optimizer, criterion, device):
    model.train()
    total_loss = 0.0

    for X_batch, y_batch in dataloader:
        X_batch = X_batch.to(device)
        y_batch = y_batch.to(device)

        optimizer.zero_grad()
        logits = model(X_batch)
        loss = criterion(logits, y_batch)

        loss.backward()
        optimizer.step()

        total_loss += loss.item()

    return total_loss / len(dataloader)
```

```python
def evaluate(model, dataloader, criterion, device):
    model.eval()
    total_loss = 0.0

    with torch.no_grad():
        for X_batch, y_batch in dataloader:
            X_batch = X_batch.to(device)
            y_batch = y_batch.to(device)

            logits = model(X_batch)
            loss = criterion(logits, y_batch)

            total_loss += loss.item()

    return total_loss / len(dataloader)
```

```python
EPOCHS = 5
for epoch in range(1, EPOCHS + 1):
    train_loss = train_one_epoch(
        model, train_loader, optimizer, criterion, device
    )
    val_loss = evaluate(
        model, val_loader, criterion, device
    )

    print(
        f"Epoch {epoch:02d} | "
        f"Train Loss: {train_loss:.4f} | "
        f"Val Loss: {val_loss:.4f}"
    )
[15]
```

```
Epoch 01 | Train Loss: 0.6930 | Val Loss: 0.6927
Epoch 02 | Train Loss: 0.6844 | Val Loss: 0.6932
Epoch 03 | Train Loss: 0.6663 | Val Loss: 0.6782
Epoch 04 | Train Loss: 0.6560 | Val Loss: 0.7050
Epoch 05 | Train Loss: 0.6431 | Val Loss: 0.7117
```

During training, training loss decreases, but validation loss begins to increase after several epochs, indicating overfitting of the base model.




```python
def compute_accuracy(model, dataloader, device):
    model.eval()
    correct = 0
    total = 0
    with torch.no_grad():
        for X_batch, y_batch in dataloader:
            X_batch = X_batch.to(device)
            y_batch = y_batch.to(device)
            logits = model(X_batch)
            probs = torch.sigmoid(logits)
            preds = (probs >= 0.5).float()
            correct += (preds == y_batch).sum().item()
            total += y_batch.size(0)
    return correct / total
```

```python
val_accuracy = compute_accuracy(model, val_loader, device)
print(f"Validation Accuracy: {val_accuracy:.4f}")
[17]
```

```
Validation Accuracy: 0.5045
```

```python
test_dataset = IMDBTensorDataset(X_test, y_test)
test_loader = DataLoader(
    test_dataset,
    batch_size=64,
    shuffle=False
)
test_accuracy = compute_accuracy(model, test_loader, device)
print(f"Test Accuracy: {test_accuracy:.4f}")
```

```
Test Accuracy: 0.5068
```

Accuracy is used as the primary metric because the IMDb dataset is balanced.

PRESERVING THE BASE MODEL

```python
MODEL_DIR = Path("models")
MODEL_DIR.mkdir(exist_ok=True)
torch.save(
    model.state_dict(),
    MODEL_DIR / "baseline_lstm.pt"
)
```

RESULT

A basic LSTM model was trained for the task of analyzing the tone of IMDb reviews.

The model demonstrates learning on training data, but its quality on validation and test samples is close to random, which indicates the limitations of the architecture.

This model is used as a baseline. The next step is to improve the architecture using BiLSTM.

MODEL IMPROVEMENTS

The goal of this stage is to improve the quality of the base model for classifying
the sentiment of IMDb text reviews through architectural improvements
and hyperparameter tuning.

At this stage, the data and preprocessing procedure are considered fixed.
All changes relate exclusively to the model architecture and its parameters.

UPLOADING PREPARED DATA AND ARTIFACTS

At this stage, we load the results of the previous steps:

prepared train / validation / test tensors
word2idx dictionary
we fix the size of the dictionary

```
ARTIFACTS_DIR = Path("artifacts")
MODELS_DIR = Path("models")
[5]
```

```
data = torch.load(ARTIFACTS_DIR / "dataset.pt")
X_train = data["X_train"]
y_train = data["y_train"]
X_val = data["X_val"]
y_val = data["y_val"]
X_test = data["X_test"]
y_test = data["y_test"]
```

```
with open(ARTIFACTS_DIR / "word2idx.pkl", "rb") as f:
    word2idx = pickle.load(f)
VOCAB_SIZE = len(word2idx)
PAD_IDX = word2idx["<PAD>"]
```

CREATING DATASET AND DATALOADER

PyTorch Dataset and DataLoader are used for training and evaluating the model. This allows you to:

efficiently load data in batches
use shuffling for training samples
provide a unified interface for training and validation

```
class IMDBTensorDataset(Dataset):
    def __init__(self, X, y):
        self.X = X
        self.y = y
    def __len__(self):
        return self.X.size(0)
    def __getitem__(self, idx):
        return self.X[idx], self.y[idx]
[8]
```

```
BATCH_SIZE = 64
train_dataset = IMDBTensorDataset(X_train, y_train)
val_dataset   = IMDBTensorDataset(X_val, y_val)
test_dataset  = IMDBTensorDataset(X_test, y_test)
train_loader = DataLoader(
    train_dataset,
    batch_size=BATCH_SIZE,
    shuffle=True
)
val_loader = DataLoader(
    val_dataset,
    batch_size=BATCH_SIZE,
    shuffle=False
)
test_loader = DataLoader(
    test_dataset,
    batch_size=BATCH_SIZE,
    shuffle=False
)
```

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
device
[10]
```

```
device(type='cuda')
```

```
batch_X, batch_y = next(iter(train_loader))
batch_X.shape, batch_y.shape
[11]
```

```
(torch.Size([64, 400]), torch.Size([64]))
```

OUR BASIC MODEL

```python
class LSTMClassifier(nn.Module):
    def __init__(self, vocab_size, embedding_dim, hidden_dim, padding_idx):
        super().__init__()
        self.embedding = nn.Embedding(
            num_embeddings=vocab_size,
            embedding_dim=embedding_dim,
            padding_idx=padding_idx
        )
        self.lstm = nn.LSTM(
            input_size=embedding_dim,
            hidden_size=hidden_dim,
            batch_first=True
        )
        self.fc = nn.Linear(hidden_dim, 1)
    def forward(self, x):
        embedded = self.embedding(x)
        _, (hidden, _) = self.lstm(embedded)
        hidden = hidden.squeeze(0)
        logits = self.fc(hidden)
        return logits.squeeze(1)
```

```python
EMBEDDING_DIM = 128
HIDDEN_DIM = 128
baseline_model = LSTMClassifier(
    vocab_size=VOCAB_SIZE,
    embedding_dim=EMBEDDING_DIM,
    hidden_dim=HIDDEN_DIM,
    padding_idx=PAD_IDX
)
baseline_model.load_state_dict(
    torch.load(MODELS_DIR / "baseline_lstm.pt", map_location=device)
)
baseline_model = baseline_model.to(device)
baseline_model.eval()
```

```
LSTMClassifier(
  (embedding): Embedding(20000, 128, padding_idx=0)
  (lstm): LSTM(128, 128, batch_first=True)
  (fc): Linear(in_features=128, out_features=1, bias=True)
)
```

```python
def compute_accuracy(model, dataloader, device):
    model.eval()
    correct = 0
    total = 0
    with torch.no_grad():
        for X_batch, y_batch in dataloader:
            X_batch = X_batch.to(device)
            y_batch = y_batch.to(device)
            logits = model(X_batch)
            probs = torch.sigmoid(logits)
            preds = (probs >= 0.5).float()
            correct += (preds == y_batch).sum().item()
            total += y_batch.size(0)
    return correct / total
```

[16]

```python
val_accuracy_baseline = compute_accuracy(baseline_model, val_loader, device)
test_accuracy_baseline = compute_accuracy(baseline_model, test_loader, device)
val_accuracy_baseline, test_accuracy_baseline
```

[17]

```
(0.5045333333333333, 0.5068)
```

MODEL ARCHITECTURE WITH MASKED POOLING AND DROPOUT

Unlike the basic BiLSTM, the following are used here:

masked mean pooling
masked max pooling
dropout for regularization
This allows information across the entire sequence to be taken into account and reduces the influence of padding tokens.

```python
class BiLSTMClassifier(nn.Module):
    def __init__(
        self,
        vocab_size,
        embedding_dim,
        hidden_dim,
        padding_idx
    ):
        super().__init__()
        self.embedding = nn.Embedding(
            num_embeddings=vocab_size,
            embedding_dim=embedding_dim,
            padding_idx=padding_idx
        )
        self.lstm = nn.LSTM(
            input_size=embedding_dim,
            hidden_size=hidden_dim,
            batch_first=True,
            bidirectional=True
        )
        self.fc = nn.Linear(hidden_dim * 2, 1)
```

```python
def forward(self, x):
    embedded = self.embedding(x)
    _, (hidden, _) = self.lstm(embedded)
    forward_hidden = hidden[0]
    backward_hidden = hidden[1]
    combined = torch.cat(
        (forward_hidden, backward_hidden),
        dim=1
    )
    logits = self.fc(combined)
    return logits.squeeze(1)
```

MODEL TRAINING AND SELECTING THE BEST CHECKPOINT

The model is trained over several epochs. The best checkpoint is selected based on the minimum validation loss value, which helps avoid overfitting.

```
model = BiLSTMClassifier(
    vocab_size=VOCAB_SIZE,
    embedding_dim=EMBEDDING_DIM,
    hidden_dim=HIDDEN_DIM,
    padding_idx=PAD_IDX
)
model = model.to(device)
model
[19]
```

```
BiLSTMClassifier(
  (embedding): Embedding(20000, 128, padding_idx=0)
  (lstm): LSTM(128, 128, batch_first=True, bidirectional=True)
  (fc): Linear(in_features=256, out_features=1, bias=True)
)
```

```
def evaluate(model, dataloader, criterion, device):
    model.eval()
    total_loss = 0.0

    with torch.no_grad():
        for X_batch, y_batch in dataloader:
            X_batch = X_batch.to(device)
            y_batch = y_batch.to(device)

            logits = model(X_batch)
            loss = criterion(logits, y_batch)

            total_loss += loss.item()

    return total_loss / len(dataloader)
```

```
def train_one_epoch(model, dataloader, optimizer, criterion, device):
    model.train()
    total_loss = 0.0
    for X_batch, y_batch in dataloader:
        X_batch = X_batch.to(device)
        y_batch = y_batch.to(device)
        optimizer.zero_grad()
        logits = model(X_batch)
        loss = criterion(logits, y_batch)

        loss.backward()
        optimizer.step()

        total_loss += loss.item()

    return total_loss / len(dataloader)
```

```
criterion = nn.BCEWithLogitsLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)
EPOCHS = 5
best_val_loss = float("inf")
best_state = None
best_epoch = None

EPOCHS = 5

for epoch in range(1, EPOCHS + 1):
    train_loss = train_one_epoch(
        model, train_loader, optimizer, criterion, device
    )

    val_loss = evaluate(
        model, val_loader, criterion, device
    )

    val_acc = compute_accuracy(
        model, val_loader, device
    )
```

```
    if val_loss < best_val_loss:
        best_val_loss = val_loss
        best_state = {
            k: v.detach().cpu().clone()
            for k, v in model.state_dict().items()
        }
        best_epoch = epoch

    print(
        f"Epoch {epoch:02d} | "
        f"Train Loss: {train_loss:.4f} | "
        f"Val Loss: {val_loss:.4f} | "
        f"Val Acc: {val_acc:.4f}"
    )
print(f"\nBest epoch: {best_epoch} | Best val loss: {best_val_loss:.4f}")
```

```
Epoch 01 | Train Loss: 0.2343 | Val Loss: 0.3959 | Val Acc: 0.8492
Epoch 02 | Train Loss: 0.1791 | Val Loss: 0.4031 | Val Acc: 0.8563
Epoch 03 | Train Loss: 0.1865 | Val Loss: 0.4901 | Val Acc: 0.8127
Epoch 04 | Train Loss: 0.1396 | Val Loss: 0.4272 | Val Acc: 0.8643
Epoch 05 | Train Loss: 0.1083 | Val Loss: 0.4373 | Val Acc: 0.8529

Best epoch: 1 | Best val loss: 0.3959
```

```
model.load_state_dict(best_state)
model.to(device)
[26]
```

```
BiLSTMClassifier(
  (embedding): Embedding(20000, 128, padding_idx=0)
  (lstm): LSTM(128, 128, batch_first=True, bidirectional=True)
  (fc): Linear(in_features=256, out_features=1, bias=True)
)
```

MODEL QUALITY ASSESSMENT ON A TEST SAMPLE

After training, the best checkpoint is loaded, and the final accuracy on the test data is calculated.

```
test_accuracy_best = compute_accuracy(
    model,
    test_loader,
    device
)
print(f"Test Accuracy (BiLSTM, best checkpoint): {test_accuracy_best:.4f}")
[27]
 Test Accuracy (BiLSTM, best checkpoint): 0.8472
```

BILSTM WITH MASKED POOLING AND DROPOUT

The main ideas behind this step are:

to take into account the contribution of each word in the review, not just the extreme tokens
to completely ignore padding tokens when aggregating features
to reduce overfitting using dropout
To do this, masked mean pooling and masked max pooling are used, which allow you to form a stable and informative representation of the entire text. BiLSTM with Masked Pooling and Dropout

```
class BiLSTMPoolingClassifier(nn.Module):
    def __init__(self, vocab_size, embedding_dim, hidden_dim, padding_idx, dropout=0.5)
        super().__init__()

        self.padding_idx = padding_idx

        self.embedding = nn.Embedding(
            num_embeddings=vocab_size,
            embedding_dim=embedding_dim,
            padding_idx=padding_idx
        )

        self.lstm = nn.LSTM(
            input_size=embedding_dim,
            hidden_size=hidden_dim,
            batch_first=True,
            bidirectional=True
        )

        self.dropout = nn.Dropout(dropout)
        self.fc = nn.Linear(hidden_dim * 4, 1)
```

```
def forward(self, x):
    """
    x: (batch, seq_len)
    """
    embedded = self.embedding(x)
    output, _ = self.lstm(embedded)
    mask = (x != self.padding_idx)
    mask = mask.unsqueeze(-1)
    output_masked = output * mask
    lengths = mask.sum(dim=1).clamp(min=1)
    mean_pool = output_masked.sum(dim=1) / lengths
    output_for_max = output.masked_fill(~mask, -1e9)
    max_pool = output_for_max.max(dim=1).values
    pooled = torch.cat([mean_pool, max_pool], dim=1)

    pooled = self.dropout(pooled)
    logits = self.fc(pooled).squeeze(1)

    return logits
```

```
POOL_DROPOUT = 0.5

model_pool = BiLSTMPoolingClassifier(
    vocab_size=VOCAB_SIZE,
    embedding_dim=EMBEDDING_DIM,
    hidden_dim=HIDDEN_DIM,
    padding_idx=PAD_IDX,
    dropout=POOL_DROPOUT
).to(device)

with torch.no_grad():
    logits = model_pool(batch_X.to(device))

logits.shape

[29]
 torch.Size([64])
```

```
criterion = nn.BCEWithLogitsLoss()
optimizer = torch.optim.Adam(model_pool.parameters(), lr=1e-3)

best_val_loss = float("inf")
best_state = None
best_epoch = None

EPOCHS = 5

for epoch in range(1, EPOCHS + 1):
    train_loss = train_one_epoch(model_pool, train_loader, optimizer, criterion, device)
    val_loss   = evaluate(model_pool, val_loader, criterion, device)
    val_acc    = compute_accuracy(model_pool, val_loader, device)

    if val_loss < best_val_loss:
        best_val_loss = val_loss
        best_state = {k: v.detach().cpu().clone() for k, v in model_pool.state_dict().items()}
        best_epoch = epoch
```

```
    print(
        f"Epoch {epoch:02d} | "
        f"Train Loss: {train_loss:.4f} | "
        f"Val Loss: {val_loss:.4f} | "
        f"Val Acc: {val_acc:.4f}"
    )

print(f"\nBest epoch: {best_epoch} | Best val loss: {best_val_loss:.4f}")

[30]
 Epoch 01 | Train Loss: 0.4575 | Val Loss: 0.3208 | Val Acc: 0.8637
 Epoch 02 | Train Loss: 0.2609 | Val Loss: 0.2684 | Val Acc: 0.8917
 Epoch 03 | Train Loss: 0.2051 | Val Loss: 0.2699 | Val Acc: 0.8884
 Epoch 04 | Train Loss: 0.1487 | Val Loss: 0.2805 | Val Acc: 0.8933
 Epoch 05 | Train Loss: 0.1012 | Val Loss: 0.3011 | Val Acc: 0.8963

 Best epoch: 2 | Best val loss: 0.2684
```

EVALUATION OF THE BILSTM MODEL WITH MASKED POOLING AND DROPOUT

```
model_pool.load_state_dict(best_state)
model_pool.to(device)
test_acc = compute_accuracy(model_pool, test_loader, device)
print(f"Test Accuracy (BiLSTM + Pooling + Dropout, best): {test_acc:.4f}")
[31]
```

```
Test Accuracy (BiLSTM + Pooling + Dropout, best): 0.9015
```

As a result of applying this architecture, an accuracy of approximately 0.90 was achieved on the test sample, which significantly exceeds the results of the basic BiLSTM model.

HYPERPARAMETER SELECTION (OPTUNA)

After architectural improvements, the next step is to select the model hyperparameters.

The goal of this step is to find the optimal model configuration that provides the best quality on the validation sample.

```
def objective(trial):
    hidden_dim = trial.suggest_categorical("hidden_dim", [64, 128, 256])
    dropout = trial.suggest_float("dropout", 0.2, 0.6)
    lr = trial.suggest_float("lr", 1e-4, 3e-3, log=True)

    model = BiLSTMPoolingClassifier(
        vocab_size=VOCAB_SIZE,
        embedding_dim=EMBEDDING_DIM,
        hidden_dim=hidden_dim,
        padding_idx=PAD_IDX,
        dropout=dropout
    ).to(device)

    criterion = nn.BCEWithLogitsLoss()
    optimizer = torch.optim.Adam(model.parameters(), lr=lr)

    EPOCHS = 3

    for _ in range(EPOCHS):
        train_one_epoch(model, train_loader, optimizer, criterion, device)

    val_acc = compute_accuracy(model, val_loader, device)
    return val_acc
```

```
udy = optuna.create_study(direction="maximize")
udy.optimize(objective, n_trials=15)
5]
```

```
[I 2025-12-16 17:57:59,352] A new study created in memory with name: no-name-38c30e6e-aa12-4369-acb0-ff228df5e133
[I 2025-12-16 17:58:21,145] Trial 0 finished with value: 0.8332 and parameters: {'hidden_dim': 64, 'dropout': 0.5679973463
Best is trial 0 with value: 0.8332.
[I 2025-12-16 17:58:42,181] Trial 1 finished with value: 0.7876 and parameters: {'hidden_dim': 64, 'dropout': 0.3773796845
Best is trial 0 with value: 0.8332.
[I 2025-12-16 18:01:16,947] Trial 2 finished with value: 0.8569333333333333 and parameters: {'hidden_dim': 128, 'dropout':
.0002450578989315937}. Best is trial 2 with value: 0.8569333333333333.
[I 2025-12-16 18:03:31,141] Trial 3 finished with value: 0.8970666666666667 and parameters: {'hidden_dim': 64, 'dropout':
.002167795446235008}. Best is trial 3 with value: 0.8970666666666667.
[I 2025-12-16 18:14:03,217] Trial 4 finished with value: 0.9077333333333333 and parameters: {'hidden_dim': 256, 'dropout':
.0012688497338454597}. Best is trial 4 with value: 0.9077333333333333.
[I 2025-12-16 18:17:31,071] Trial 5 finished with value: 0.8488 and parameters: {'hidden_dim': 128, 'dropout': 0.573505526
Best is trial 4 with value: 0.9077333333333333.
```

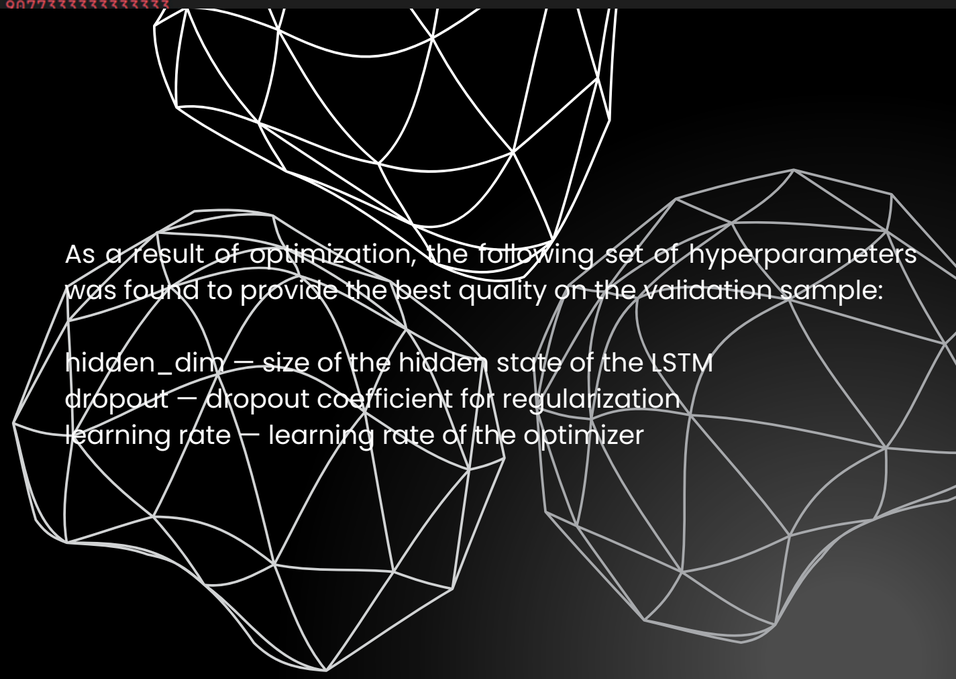
OPTUNA OPTIMIZATION RESULTS

```
study.best_trial
[36]
```

```
FrozenTrial(number=4, state=<TrialState.COMPLETE: 1>, values=[0.9077333333333333], datetime_start=datetime.datetime(2025, 12, 16, 18, 3, 31, 142415),
datetime_complete=datetime.datetime(2025, 12, 16, 18, 14, 3, 217092), params={'hidden_dim': 256, 'dropout': 0.41375295777293747, 'lr': 0.0012688497338454597},
user_attrs={}, system_attrs={}, intermediate_values={}, distributions={'hidden_dim': CategoricalDistribution(choices=(64, 128, 256)), 'dropout': FloatDistribution
(high=0.6, log=False, low=0.2, step=None), 'lr': FloatDistribution(high=0.003, log=True, low=0.0001, step=None)}, trial_id=4, value=None)
```

```
print("Best validation accuracy:", study.best_value)
print("Best hyperparameters:")
for k, v in study.best_params.items():
    print(f"  {k}: {v}")
[37]
```

```
Best validation accuracy: 0.9077333333333333
Best hyperparameters:
  hidden_dim: 256
  dropout: 0.41375295777293747
  lr: 0.0012688497338454597
```



As a result of optimization, the following set of hyperparameters was found to provide the best quality on the validation sample:

hidden_dim — size of the hidden state of the LSTM
dropout — dropout coefficient for regularization
learning rate — learning rate of the optimizer

FINAL TRAINING OF THE MODEL WITH OPTIMAL PARAMETERS

```python
best_params = study.best_params
final_model = BiLSTMPoolingClassifier(
    vocab_size=VOCAB_SIZE,
    embedding_dim=EMBEDDING_DIM,
    hidden_dim=best_params["hidden_dim"],
    padding_idx=PAD_IDX,
    dropout=best_params["dropout"]
).to(device)
criterion = nn.BCEWithLogitsLoss()
optimizer = torch.optim.Adam(final_model.parameters(), lr=best_params["lr"])
EPOCHS = 5
best_val_loss = float("inf")
best_state = None
for epoch in range(1, EPOCHS + 1):
    train_loss = train_one_epoch(final_model, train_loader, optimizer, criterion, device)
    val_loss   = evaluate(final_model, val_loader, criterion, device)
    if val_loss < best_val_loss:
        best_val_loss = val_loss
        best_state = {k: v.detach().cpu().clone() for k, v in final_model.state_dict().items()}
    print(f"Epoch {epoch:02d} | Train Loss: {train_loss:.4f} | Val Loss: {val_loss:.4f}")
final_model.load_state_dict(best_state)
final_model.to(device)
```

```
Epoch 01 | Train Loss: 0.4229 | Val Loss: 0.2895
Epoch 02 | Train Loss: 0.2371 | Val Loss: 0.2512
Epoch 03 | Train Loss: 0.1663 | Val Loss: 0.2365
Epoch 04 | Train Loss: 0.1109 | Val Loss: 0.2811
Epoch 05 | Train Loss: 0.0652 | Val Loss: 0.2972
BiLSTMPoolingClassifier(
  (embedding): Embedding(20000, 128, padding_idx=0)
  (lstm): LSTM(128, 256, batch_first=True, bidirectional=True)
  (dropout): Dropout(p=0.41375295777293747, inplace=False)
  (fc): Linear(in_features=1024, out_features=1, bias=True)
)
```

The final architecture of the model after applying architectural improvements and hyperparameter selection is shown above.

The model includes:

embedding layer
bidirectional LSTM with increased hidden_dim
masked mean and max pooling
dropout for regularization
linear classifier

FINAL EVALUATION OF THE MODEL ON THE TEST SAMPLE

```
final_test_acc = compute_accuracy(final_model, test_loader, device)
print(f"Final Test Accuracy (Optuna tuned): {final_test_acc:.4f}")
[39]
```

```
Final Test Accuracy (Optuna tuned): 0.9117
```

PRESERVING OUR IMPROVED MODEL

```
torch.save(
    final_model.state_dict(),
    "artifacts/bilstm_pooling_optuna_best.pt"
)
```

FINAL CONCLUSION ON THE MODEL TUNING STAGE

During the Model Tuning stage, the following was performed:

architectural improvements to the model
elimination of the influence of padding tokens using masked pooling
regularization using dropout
automatic selection of hyperparameters using Optuna
The final model achieved an accuracy of 0.91 on the test sample, which significantly exceeds the results of the baseline model and confirms the effectiveness of the chosen approach.

THANK YOU