Fake-News Classifier

Yocheved Ofstein, Shay Gali, Shalom Ofstein

January 2025

Abstract

This project aims to develop a machine learning model that classifies news articles as either "true" or "fake," focusing on identifying misleading or false information. Framed as a binary classification task, the model relies on textual content for the classification. A variety of models, from simple baseline classifiers to advanced deep learning techniques, were implemented and evaluated. Performance was assessed using classification metrics such as accuracy, precision, recall, and F1-score. The results show that complex models significantly outperform the baseline, emphasizing the value of advanced architectures in tackling the challenge of fake news detection.
The complete implementation and codebase are available on GitHub: Fake-News Clasifier.

1 Introduction

1.1 Motivation

The goal of this project is to develop a machine learning model that classifies news articles as either true or fake. By analyzing textual features, the model aims to predict the veracity of news content, framing the problem as a binary classification task. The objective is to differentiate between accurate news and misleading or false information.

1.2 Dataset Description

The dataset used in this project consists of labeled news articles categorized as "true news" and "fake news." It includes two sources, the first "true and fake news" dataset[1], and the second "WELFake" dataset [2]. Although both datasets contain additional features such as title, subject category, and publication date, the model primarily uses the cleaned and processed text content for classification. The text data is preprocessed through tokenization and stop word removal. For each model, a different numerical representation was applied.
The dataset is split into 70% for training, 15% for testing, and 15% reserved for validation to ensure effective model tuning.

1.3 Model Frameworks

The models employed in this study range from simple approaches, such as a majority class classifier (serving as the baseline), to more sophisticated techniques like logistic regression, fully connected neural networks, and advanced deep learning models. Each model's performance is evaluated using standard classification metrics, including accuracy, precision, recall, and F1-score, to compare their effectiveness in distinguishing between true and fake news.

2 Baseline Model

A baseline model serves as a simple starting point for analysis, providing a reference for evaluating the performance of more complex models. For this project, the baseline was implemented using the Majority Class Classifier approach. This method predicts the majority class in the dataset for all test samples, offering a straightforward benchmark for model performance.

2.1 Majority Class Classifier Approach

The Majority Class Classifier was implemented by calculating the distribution of "true news" and "fake news" articles in the training set. The class with the higher frequency, "true news" in this case, was selected as the majority class. As a result, the baseline model predicted "true news" for all test samples.

2.2 Results

The baseline model was evaluated using three key classification metrics: accuracy, precision, and recall. These metrics help provide a comprehensive understanding of the model's performance:

- **Accuracy**: The baseline model achieved an accuracy of approximately 51.46%, indicating that slightly more than half of the test samples belonged to the majority class.
- **Precision**: The precision for the "true news" class was 51.46%, as all predictions were for this class.
- **Recall**: The recall for the "true news" class was 100%, as all "true news" samples were correctly identified. However, the recall for the "fake news" class was 0%, as the baseline model made no predictions for this class.

The performance of the Majority Class Classifier highlights its limitations, particularly in addressing class imbalance and its inability to correctly identify instances of the minority class.

3 Logistic Regression Model

To further explore the classification task, logistic regression was implemented as a more structured approach. The following subsections outline the preprocessing steps, model training, and evaluation.

3.1 Model Training

The logistic regression model was trained using the preprocessed text data from the training set. The training process involved two key stages: preprocessing (feature extraction) and model fitting.

3.1.1 Preprocessing

In the preprocessing phase, the raw text data was transformed into numerical representations suitable for input to the machine learning model. This was done using the TF-IDF vectorizer, which captures the importance of words in the context of the entire dataset. The vectorizer was configured to select up to 5,000 features and eliminate common stopwords, ensuring the model focused on more informative and relevant terms.

3.1.2 Training

After preprocessing, the logistic regression model was trained on the transformed data. The training utilized a maximum of 1,000 iterations to ensure convergence. The logistic regression algorithm learned to assign weights to the features, allowing it to predict the probability of an instance belonging to a specific class based on the transformed input data.

3.2 Initial Attempts and Corrections

The initial implementation of the Logistic Regression model achieved an unusually high accuracy of 100%, which was later found to be caused by an unintended bias in the dataset. Upon reviewing the data, it was discovered that the term "Reuters" appears exclusively in "true news" articles early in the text, leading the model to rely on this term for classification instead of learning meaningful features.

To address this, the preprocessing step was updated to remove "Reuters" from the text data, and in addition, look for more class-exclusive words. After re-training the model, the accuracy decreased to a more realistic level, highlighting the importance of eliminating biases in the dataset.

Additionally, the initial use of CountVectorizer resulted in a validation accuracy of 90%. Switching to TfidfVectorizer, which provided a more effective feature representation, improved the accuracy to 92.9%.

3.3 Results

After the model was trained using the updated feature extraction method (TfidfVectorizer), the validation results were evaluated. The following metrics were computed for the model's performance:

- **Accuracy**: 0.929
- **Precision**: 0.93 for both "true" and "fake" classes.
- **Recall**: 0.91 for "true" and 0.94 for "fake" classes.
- **F1-Score**: 0.92 for "true" and 0.94 for "fake" classes.

3.4 Comparison with Baseline

Unlike the baseline model, which only predicted the majority class ("true news"), the logistic regression model effectively captured patterns in the text data and provided accurate predictions for both "true" and "fake" news articles. The performance metrics of the logistic regression model are summarized in Table 1.

Metric	Baseline Model	Logistic Regression
Accuracy	54.67%	92.9%
Precision (True News)	54.67%	93%
Recall (True News)	100%	94%
F1-Score (True News)	70.69%	94%
Precision (Fake News)	N/A	93%
Recall (Fake News)	0%	91%
F1-Score (Fake News)	N/A	92%

Table 1: Comparison of Model Performance Metrics.

4 Fully Connected Neural Network

Building upon the improvements seen with the logistic regression model, the Fully Connected Neural Network (FCNN) was implemented to explore the potential of even more complex models for this classification task. The FCNN's ability to learn non-linear patterns makes it a promising candidate for capturing subtle relationships in the text data that simpler models might miss.

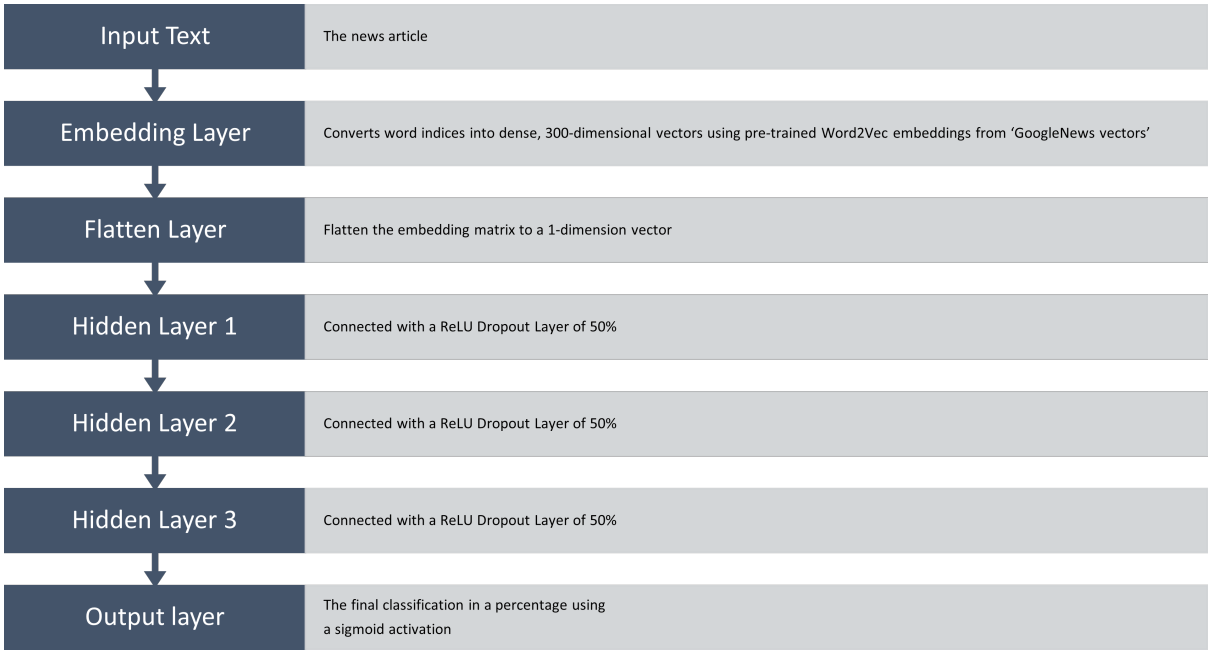
4.1 Model Architecture

The FCNN architecture consists of several key components designed to effectively process and classify the text data:

- **Embedding Layer**: Utilizes pre-trained Word2Vec [3] embeddings (300 dimensions) to convert tokenized words into dense vector representations. The embeddings are frozen during training to maintain the pre-trained semantic relationships.
- **Input Processing**: The embedded sequences are flattened into a single vector of size (embedding dim * max sequence length) to serve as input to the dense layers.
- **Hidden Layers**: The network includes three fully connected layers with decreasing dimensions:
  - First hidden layer: 512 units
  - Second hidden layer: 256 units
  - Third hidden layer: 128 units
- **Regularization Features**:
  - Layer Normalization after each hidden layer to stabilize training
  - ReLU activation functions to introduce non-linearity

  - Dropout layers (50% for input, 20% for hidden layers) to prevent overfitting

- **Output Layer**: Single unit with sigmoid activation for binary classification.



4.2 Model Training

The training process incorporates several advanced techniques to ensure robust learning and prevent overfitting:

- **Loss Function**: Binary Cross-Entropy loss with class weights to handle potential class imbalance The formula for BCEWithLogitsLoss with a pos_weight term:

$$Loss = -\frac{1}{N}\sum_{i=1}^{N}\left[y_i \cdot \log\left(\sigma(z_i)\right) \cdot \text{pos\_weight} + (1 - y_i) \cdot \log\left(1 - \sigma(z_i)\right)\right]$$

  Where:

  - N: The number of samples in the batch.
  - y_i: The ground truth label for the i-th sample ($y_i \in \{0, 1\}$).
  - z_i: The raw logits (unnormalized scores) predicted by the model for the i-th sample.
  - $\sigma(z_i)$: The sigmoid activation function applied to z_i, defined as $\sigma(z_i) = \frac{1}{1+e^{-z_i}}$.
  - pos_weight: A weighting factor applied to the positive class ($y_i = 1$) to handle class imbalance.

- **Optimizer**: AdamW optimizer with weight decay (0.1) for better regularization

- **Learning Rate Strategy**:

  - Initial learning rate: 1e-4 (0.0001)
  - Linear warmup over first 100 steps
  - ReduceLROnPlateau scheduler for dynamic adjustment

- **Training Parameters**:

  - Batch size: 32
  - Maximum epochs: 15
  - Early stopping with patience of 4 epochs
  - Gradient clipping at norm 1.0

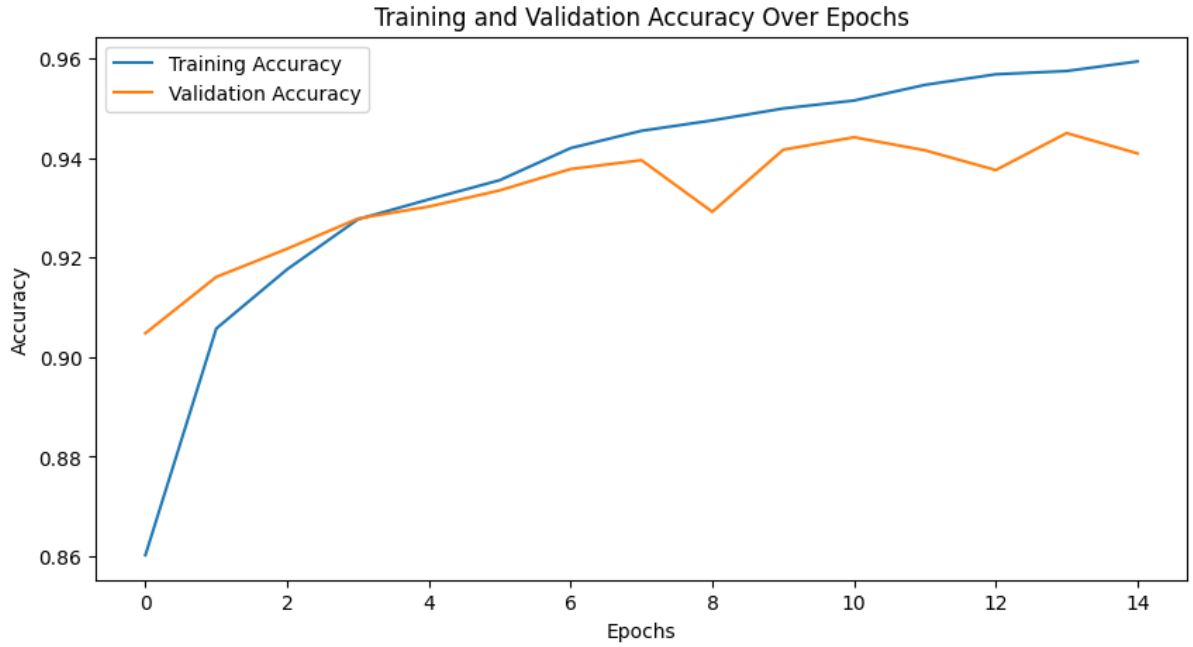
4.3 Initial Attempts and Corrections

During the initial implementation, several challenges were encountered and addressed to improve the model's performance:

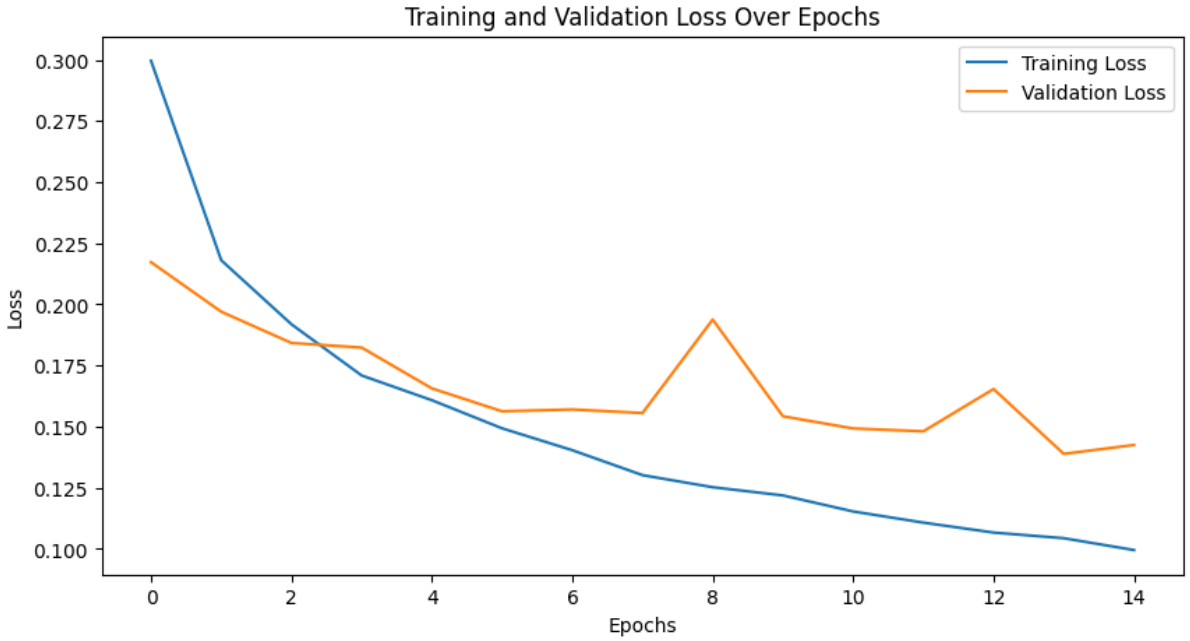
- **Gradient Handling**: Implemented gradient clipping to prevent exploding gradients during training.
- **Batch Processing**: Added shape compatibility checks to handle variable-sized batches correctly.
- **Memory Management**: Optimized the embedding layer by freezing weights to reduce memory usage and training time.

4.4 Training Visualization

4.4.1 Accuracy



4.4.2 Loss



4.5 Results

- **Accuracy**: 0.9436
- **Precision**: 0.9495
- **Recall**: 0.9473
- **F1-Score**: 0.9484

4.6 Comparison with Previous Models

To provide a clearer understanding of the FCNN's performance relative to the baseline and logistic regression models, a comparison of key evaluation metrics is presented in Table 2.

Metric	Baseline Model	Logistic Regression	FCNN
Accuracy	54.67%	92.9%	94%
Precision (True News)	54.67%	93%	95%
Recall (True News)	100%	94%	95%
F1-Score (True News)	70.69%	94%	95%
Precision (Fake News)	N/A	93%	94%
Recall (Fake News)	0%	91%	94%
F1-Score (Fake News)	N/A	92%	94%

Table 2: Comparison of Model Performance Metrics

4.7 Implementation Details

4.7.1 Pre-trained Word2Vec Embeddings[3]

```
# Load pre-trained Word2Vec model
word2vec = KeyedVectors.load_word2vec_format("../GoogleNews-vectors-negative300.bin", binary=True)
# Create a vocabulary
embedding_dim = 300
```

```
vocab = {"<PAD>": 0, "<UNK>": 1}  # Special tokens
# Initialize <PAD> and <UNK>
embedding_matrix = [np.zeros(embedding_dim), np.random.uniform(-0.01, 0.01, embedding_dim)]
# Build vocabulary from Word2Vec
for text in X_train:
    for word in word_tokenize(text.lower()):
        if word not in vocab and word in word2vec:
            vocab[word] = len(vocab)
            embedding_matrix.append(word2vec[word])

embedding_matrix = np.array(embedding_matrix)
vocab_size = len(vocab)

# Tokenize and convert text to sequences
def text_to_sequence(text, vocab, max_len):
    sequence = [vocab.get(word, vocab["<UNK>"]) for word in word_tokenize(text.lower())]
    if len(sequence) < max_len:
        sequence.extend([vocab["<PAD>"]] * (max_len - len(sequence)))
    return sequence[:max_len]

# Apply tokenization
max_len = 1000
X_train_seq = [text_to_sequence(text, vocab, max_len) for text in X_train]
X_val_seq = [text_to_sequence(text, vocab, max_len) for text in X_val]
X_test_seq = [text_to_sequence(text, vocab, max_len) for text in X_test]
```

4.7.2 Model Definition

```
class FCNNModel(nn.Module):
    def __init__(self, embedding_matrix, hidden_dims=[512, 256, 128], output_dim=1):
        super(FCNNModel, self).__init__()

        # Embedding Layer with frozen weights
        self.embedding = nn.Embedding.from_pretrained(
            torch.tensor(embedding_matrix, dtype=torch.float32),
            freeze=True,
            padding_idx=0
        )

        # Calculate input dimension
        input_dim = embedding_matrix.shape[1] * max_len

        # Create list to hold all layers
        layers = []

        # Input layer
        layers.append(nn.Linear(input_dim, hidden_dims[0]))
        layers.append(nn.LayerNorm(hidden_dims[0]))
        layers.append(nn.ReLU())
        layers.append(nn.Dropout(0.5))

        # Hidden layers
        for i in range(len(hidden_dims)-1):
            layers.append(nn.Linear(hidden_dims[i], hidden_dims[i+1]))
            layers.append(nn.LayerNorm(hidden_dims[i+1]))
            layers.append(nn.ReLU())
            layers.append(nn.Dropout(0.2))
```

```
        # Output layer
        layers.append(nn.Linear(hidden_dims[-1], output_dim))

        # Combine all layers
        self.model = nn.Sequential(*layers)

    def forward(self, x):
        # Get embeddings and flatten
        embedded = self.embedding(x)
        flattened = embedded.view(embedded.size(0), -1)

        # Forward pass through all layers
        return self.model(flattened)
```

4.7.3 Training Loop

```
def train_model(model, train_loader, val_loader, epochs=15, learning_rate=1e-4):
    num_pos = sum(y_train == 1)
    num_neg = sum(y_train == 0)
    pos_weight = torch.tensor([num_neg / num_pos]).to(device)
    criterion = nn.BCEWithLogitsLoss(pos_weight=pos_weight)

    optimizer = torch.optim.AdamW(
        model.parameters(),
        lr=learning_rate,
        weight_decay=0.1
    )

    scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(
        optimizer,
        mode='max',
        factor=0.5,
        patience=2,
        verbose=True
    )

    best_model_state = None
    best_val_loss = float('inf')
    patience = 4

    num_warmup_steps = 100
    def get_lr(step):
        if step < num_warmup_steps:
            return learning_rate * (step / num_warmup_steps)
        return learning_rate

    for epoch in range(epochs):
        # Training phase
        model.train()
        total_loss = 0
        correct = 0
        total = 0

        for i, (texts, labels) in enumerate(train_loader):
            current_lr = get_lr(epoch * len(train_loader) + i)
            for param_group in optimizer.param_groups:
                param_group['lr'] = current_lr
```

```
        texts = texts.to(device)
        # Ensure labels are float and proper shape
        labels = labels.float().to(device)

        optimizer.zero_grad()

        # Forward pass and ensure output shape matches labels
        outputs = model(texts).squeeze(-1)

        # Ensure shapes match
        if len(outputs.shape) == 0:
            outputs = outputs.unsqueeze(0)
        if len(labels.shape) == 0:
            labels = labels.unsqueeze(0)

        loss = criterion(outputs, labels)
        loss.backward()

        torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)
        optimizer.step()

        predicted = torch.round(torch.sigmoid(outputs))
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

        total_loss += loss.item()

        if i % 100 == 0:
            print(f'Epoch: {epoch}, Batch: {i}, Loss: {loss.item():.4f}')

    train_acc = correct / total
    avg_loss = total_loss / len(train_loader)
    loss_training_values.append(avg_loss)

    # Validation phase
    val_loss, val_acc = evaluate_model(model, val_loader, criterion)
    loss_validation_values.append(val_loss)

    print(f'Epoch: {epoch}')
    print(f'Average Loss: {avg_loss:.4f}')
    print(f'Training Accuracy: {train_acc:.4f}')
    print(f'Validation Loss: {val_loss:.4f}')
    print(f'Validation Accuracy: {val_acc:.4f}')

    acc_training_values.append(train_acc)
    acc_validation_values.append(val_acc)

    scheduler.step(val_acc)

    if val_loss < best_val_loss:
        best_val_loss = val_loss
        best_model_state = model.state_dict().copy()
        patience_counter = 0
        torch.save(best_model_state, 'best_fcnn_model.pth')
    else:
        patience_counter += 1

    if patience_counter >= patience:
```

```
            print('Early stopping triggered')
            # Restore best model
            model.load_state_dict(best_model_state)
            break

    return loss_training_values, loss_validation_values,
            acc_training_values, acc_validation_values
```

5 Advanced Model

This section outlines an advanced model leveraging Long Short-Term Memory (LSTM) networks, incorporating Bidirectional LSTM layers, attention mechanisms, and dropout regularization. The architecture, training methodology, and evaluation strategy employed to assess the model's performance are described in the following sections.

5.1 Model Architecture

The proposed model consists of the following key components:

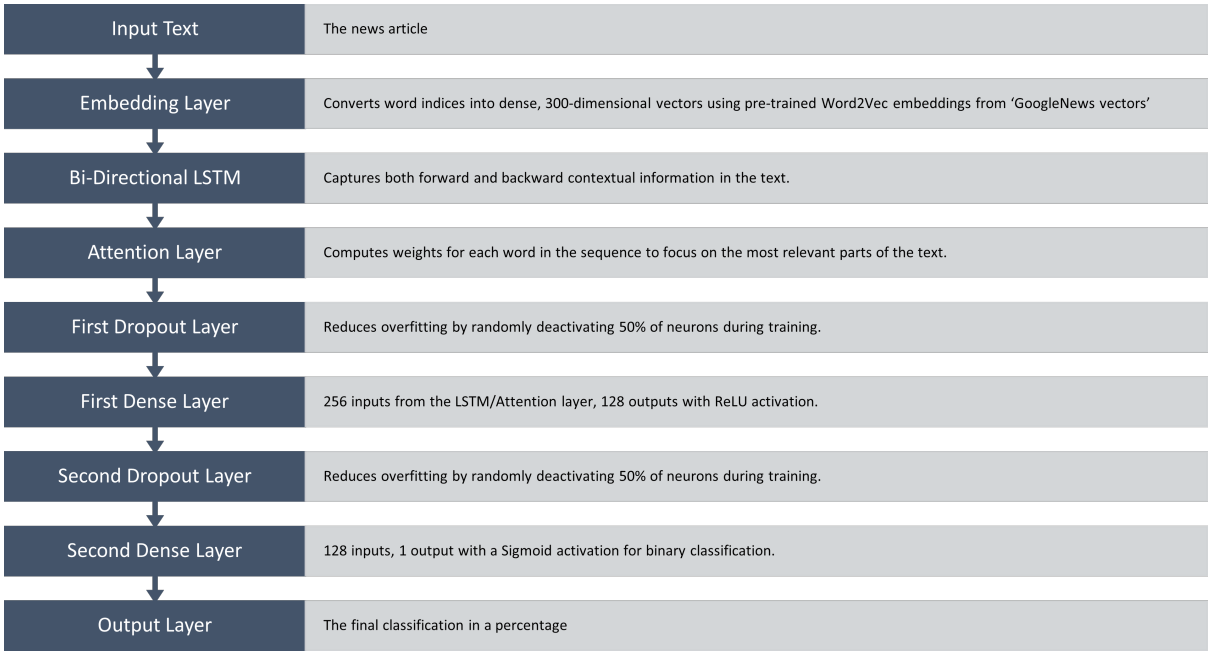


Figure 1: The Advanced Model Architecture

5.1.1 Embedding Layer

Purpose: Converts word indices into dense, 300-dimensional vectors using pre-trained Word2Vec embeddings[3].

Why Needed: Raw text cannot be directly used as input to machine learning models because they require numerical data. Embeddings provide a way to represent words as dense numerical vectors, capturing semantic relationships (e.g., "king" and "queen" have similar vectors).

Trainable: Yes, allowing fine-tuning of embeddings during training.

Input Shape: [Batch Size, Sequence Length].

Output Shape: [Batch Size, Sequence Length, Embedding Dimension (300)].

5.1.2 Bidirectional LSTM Layer

Purpose: Captures both forward and backward contextual information in the text.

Why Needed: Language is inherently sequential, and understanding a word often depends on its context (both before and after). Bidirectional LSTMs process the sequence in both directions, making it possible to consider future and past words simultaneously.

Hidden Units: 128 units in each direction, resulting in a total of 256 outputs.

Input Shape: [Batch Size, Sequence Length, Embedding Dimension (300)]

Output Shape: [Batch Size,256]

The following diagram presents the functionality of the Bi-Directional LSTM Layer:

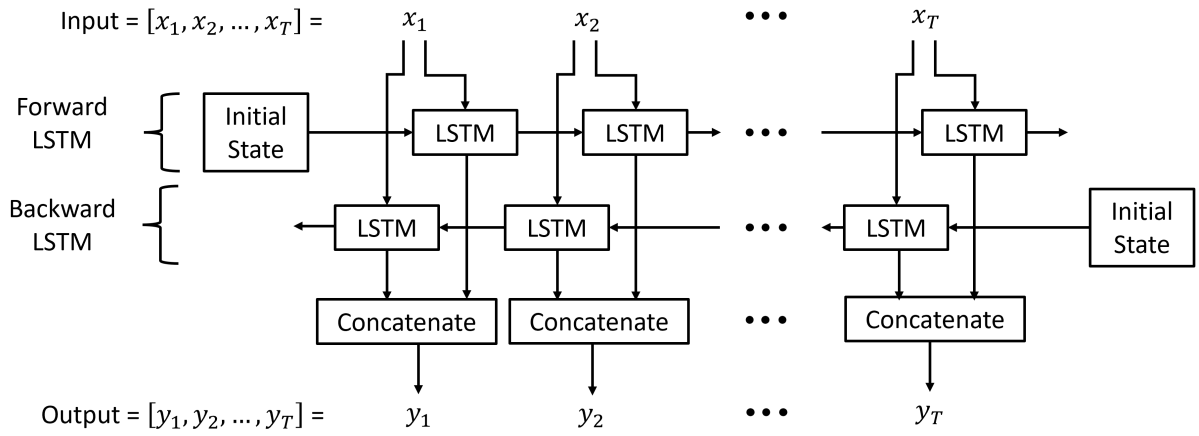


Figure 2: Bi-LSTM

5.1.3 Attention Mechanism

Purpose: Computes weights for each word in the sequence to focus on the most relevant parts of the text.

Why Needed: Not all words in a sentence contribute equally to the meaning. For example, in the sentence "The weather is beautiful, but it might rain later," the word "rain" is more important for predicting the sentiment. Attention allows the model to assign higher importance to relevant words.

Implementation: A linear layer computes attention scores, normalized using the softmax function.

Input Shape: [Batch Size,256]

Output Shape: [Batch Size,256]

The following diagram presents the Attention Layer Mechanism :

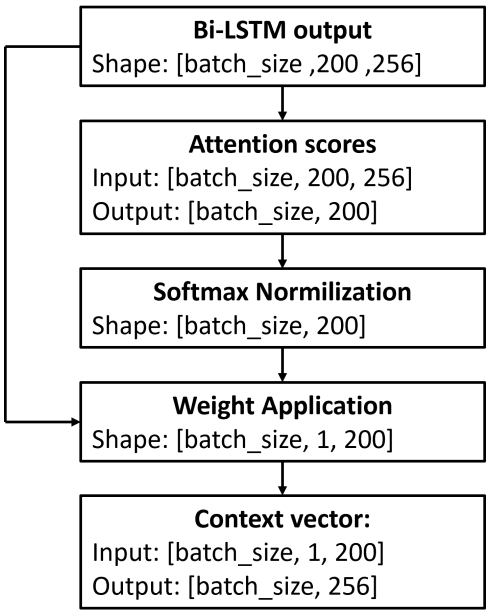


Figure 3: Attention Layer Mechanism

5.1.4 Dropout Regularization

Purpose: Reduces overfitting by randomly deactivating 50% of neurons during training.

Why Needed: Without dropout, the model may become too specialized to the training data and fail to generalize to unseen data. Dropout forces the network to learn more robust features by introducing randomness.

Input Shape: Matches the input shape of the layer where it is applied.

Output Shape: Matches the output shape of the layer where it is applied.

5.1.5 Fully Connected Layers

Structure:

- First Layer: 256 inputs from the LSTM/Attention layer, 128 outputs with ReLU activation.
- Second Layer: 128 inputs, 1 output with a Sigmoid activation for binary classification.

Purpose: Maps the high-dimensional features from the LSTM/Attention layers into a single output

Why Needed: After processing the sequential data, fully connected layers act as a classifier.

The following diagram presents the Dense Layers and the Dropout Layers as implemented in the Model:

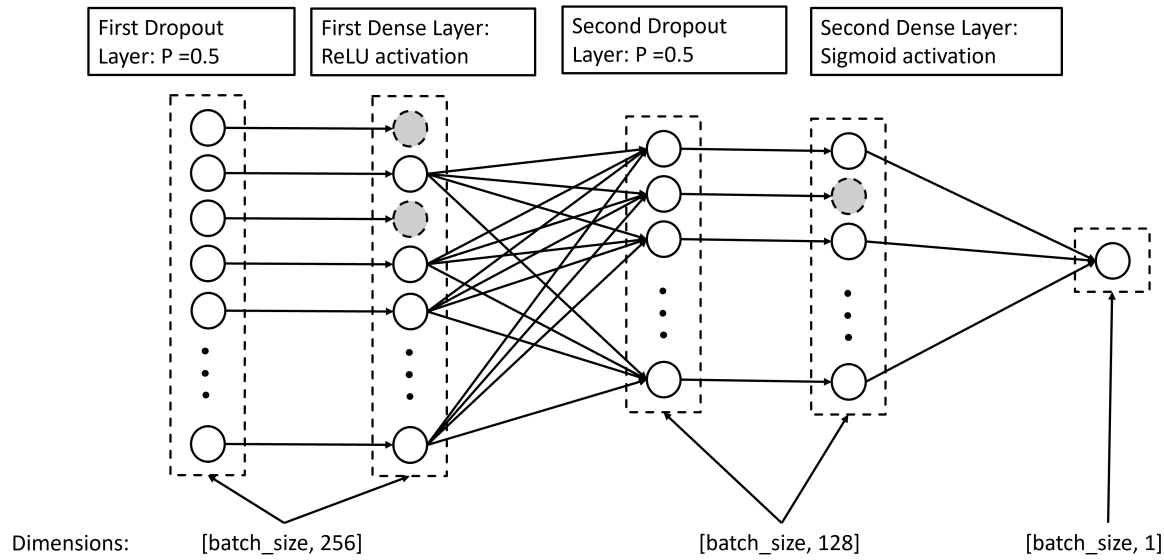


Figure 4: The Dense and Dropout Layers

5.2 Training Methodology

5.2.1 Loss Function

The Binary Cross-Entropy Loss (BCELoss) is utilized to evaluate the loss during training. BCELoss measures the divergence between predicted probabilities and true binary labels, making it an appropriate choice for binary classification tasks.

The formula for BCELoss:

$$Loss = -\frac{1}{N}\sum_{(i=1)}^{N}[(y_i)\cdot\log(p_i) + (1-y_i\ )\cdot\log(1-p_i)]$$

5.2.2 Optimizer

Adam Optimizer: Adaptive optimization algorithm with a learning rate of 0.001.

5.2.3 Training Process

- **Batch Size:** A batch size of 32 samples was used during training, balancing computational efficiency and model performance.
- **Epochs:** The model was trained for 15 epochs, iterating over the entire training dataset.
- **Validation:** After each epoch, the model's performance was evaluated on a separate validation set to monitor its generalization ability and detect overfitting.

5.3 Initial Attempts and Corrections

5.3.1 Version 1: Initial Attempt

The first version focused on building a LSTM-based text classification model using Word2Vec embeddings. The model consisted of an embedding layer, a bidirectional LSTM, and a fully connected output layer. The training loop was straightforward, using accuracy as the primary evaluation metric. However, the model lacked advanced features such as learning rate scheduling, early stopping, and detailed performance tracking, which limited its ability to prevent overfitting and monitor model behavior during training.

5.3.2 Version 2: Enhancements to Training and Evaluation

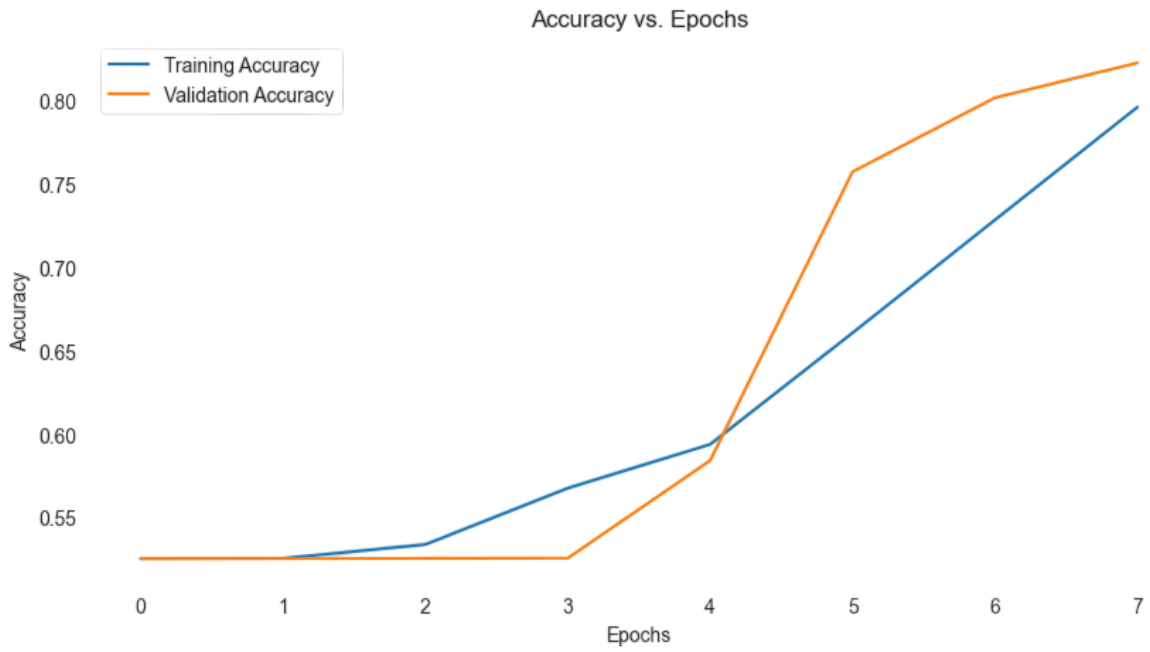
In the second version, several improvements were made to address the limitations of the first attempt. Learning rate scheduling was added to adjust the learning rate during training for better convergence, and early stopping was introduced to prevent overfitting. Evaluation was enhanced by adding precision, recall, F1 score, and a confusion matrix to provide a deeper understanding of model performance. Additionally, training and validation metrics were logged and visualized for better tracking of the model's progress. However, tokenization was still done manually, and preprocessing remained a bit inefficient.

5.3.3 Version 3: Tokenization and Sequence Length Adjustments

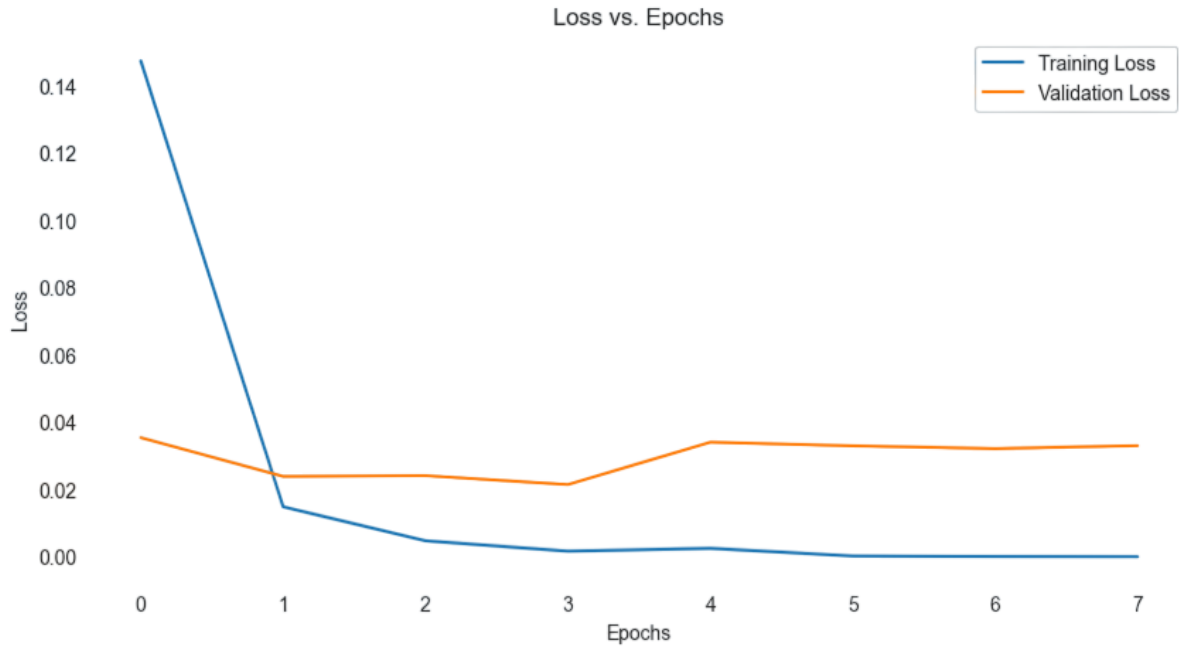
The third version introduced the use of NLTK for tokenization, replacing the manual approach from previous versions. This improved the consistency and reliability of the preprocessing step. Moreover, the maximum sequence length was increased from 200 to 1000 words to accommodate longer texts, which allowed the model to process more complex inputs. However, this change led to higher memory usage and slower training times due to the increased sequence length.

5.4 Training Visualization

5.4.1 Accuracy



5.4.2 Loss



5.5 Results

The model's performance on the test set was evaluated using several key metrics:

- **Accuracy**: 0.9619
- **Precision**: 0.9488
- **Recall**: 0.9833
- **F1-Score**: 0.9658

5.6 Comparison with Previous Models

To assess the improvements made by the current model, its performance is compared with previous models evaluated in similar tasks. The following metrics are presented in Table 3.

Metric	Baseline	Logistic Regression	FCNN	Advanced Model
Accuracy	54.67%	92.9%	94%	96%
Precision (True News)	54.67%	93%	95%	95%
Recall (True News)	100%	94%	95%	98%
F1-Score (True News)	70.69%	94%	95%	97%
Precision (Fake News)	N/A	93%	94%	98%
Recall (Fake News)	0%	91%	94%	94%
F1-Score (Fake News)	N/A	92%	94%	96%

Table 3: Comparison of Model Performance Metrics

5.7 Implementation Details

5.7.1 Model Definition

```
class AdvancedLSTMClassifier(nn.Module):
    def __init__(self, embedding_matrix, hidden_dim, output_dim, attention=True):
```

```
        super(AdvancedLSTMClassifier, self).__init__()
        # Embedding layer initialized with pre-trained Word2Vec embeddings
        self.embedding = nn.Embedding.from_pretrained(torch.tensor(embedding_matrix,
        dtype=torch.float32), freeze=False)

        # Bidirectional LSTM
        self.lstm = nn.LSTM(input_size=embedding_matrix.shape[1], hidden_size=hidden_dim,
                            batch_first=True, bidirectional=True)

        # Attention mechanism (optional)
        self.attention = attention
        if attention:
            self.attention_weights = nn.Linear(2 * hidden_dim, 1)  # Compute attention scores

        # Dropout for regularization
        self.dropout = nn.Dropout(0.5)

        # Fully connected layers
        self.fc1 = nn.Linear(2 * hidden_dim, 128)  # First dense layer
        self.fc2 = nn.Linear(128, output_dim)  # Second dense layer
        self.sigmoid = nn.Sigmoid()  # Output layer for binary classification

    def forward(self, x):
        # Step 1: Embedding layer
        x = self.embedding(x)

        # Step 2: Bidirectional LSTM
        lstm_out, _ = self.lstm(x)  # lstm_out shape: [batch_size, seq_len, 2 * hidden_dim]

        # Step 3: Attention mechanism (if enabled)
        if self.attention:
            attention_scores = self.attention_weights(lstm_out).squeeze(-1)
            attention_weights = torch.softmax(attention_scores, dim=1)
            x = torch.bmm(attention_weights.unsqueeze(1), lstm_out).squeeze(1)
        else:
            x = lstm_out[:, -1, :]  # Use the last hidden state if no attention

        # Step 4: Fully connected layers
        x = self.dropout(x)  # Apply dropout
        x = torch.relu(self.fc1(x))  # First dense layer with ReLU activation
        x = self.dropout(x)  # Apply dropout
        x = self.fc2(x)  # Second dense layer

        # Step 5: Output layer
        return self.sigmoid(x)
```

5.7.2 Training Loop

```
def train_model(model, train_loader, val_loader, epochs=15, learning_rate=0.0005):

    metrics_history = {
        'train_loss': [], 'val_loss': [],
        'train_acc': [], 'val_acc': [],
        'train_precision': [], 'val_precision': [],
        'train_recall': [], 'val_recall': [],
        'train_f1': [], 'val_f1': [],
        'learning_rates': []
    }
```

```
criterion = nn.BCELoss()
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)

scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(
    optimizer, mode='max', factor=0.5, patience=2, min_lr=1e-6
)

best_model_state = None
best_val_loss = float('inf')
patience = 4
patience_counter = 0

num_warmup_steps = 100
def get_lr(step):
    if step < num_warmup_steps:
        return learning_rate * (step / num_warmup_steps)
    return learning_rate

for epoch in range(epochs):
    print(f"\nEpoch {epoch+1}/{epochs}")
    print("-" * 50)

    # Training phase
    model.train()
    train_predictions = []
    train_true_labels = []
    total_train_loss = 0

    for i, (texts, labels) in enumerate(train_loader):
        current_lr = get_lr(epoch * len(train_loader) + i)
        for param_group in optimizer.param_groups:
            param_group['lr'] = current_lr

        texts = texts.to(device)
        labels = labels.float().to(device)

        optimizer.zero_grad()
        outputs = model(texts).squeeze(1)

        loss = criterion(outputs, labels)
        loss.backward()

        torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)
        optimizer.step()

        total_train_loss += loss.item()

        # Store predictions and true labels
        predicted = torch.round(outputs.detach())
        train_predictions.extend(predicted.cpu().numpy())
        train_true_labels.extend(labels.cpu().detach().numpy())

        if (i + 1) % 50 == 0:
            print(f"Batch {i+1}/{len(train_loader)}: Loss: {loss.item():.4f}")

    # Calculate training metrics
    train_acc = accuracy_score(train_true_labels, train_predictions)
```

```
        train_precision = precision_score(train_true_labels, train_predictions)
        train_recall = recall_score(train_true_labels, train_predictions)
        train_f1 = f1_score(train_true_labels, train_predictions)
        avg_train_loss = total_train_loss / len(train_loader)

        # Validation phase
        val_loss, val_metrics = evaluate_model(model, val_loader, criterion)

        # Store metrics
        metrics_history['train_loss'].append(avg_train_loss)
        metrics_history['val_loss'].append(val_loss)
        metrics_history['train_acc'].append(train_acc)
        metrics_history['val_acc'].append(val_metrics['accuracy'])
        metrics_history['train_precision'].append(train_precision)
        metrics_history['val_precision'].append(val_metrics['precision'])
        metrics_history['train_recall'].append(train_recall)
        metrics_history['val_recall'].append(val_metrics['recall'])
        metrics_history['train_f1'].append(train_f1)
        metrics_history['val_f1'].append(val_metrics['f1'])
        metrics_history['learning_rates'].append(current_lr)

        # Print epoch metrics
        print(f"\nEpoch {epoch+1} Results:")
        print(f"Training Loss: {avg_train_loss:.4f}")
        print(f"Validation Loss: {val_loss:.4f}")
        print(f"Training Metrics: Acc={train_acc:.4f},
                Prec={train_precision:.4f}, Rec={train_recall:.4f}, F1={train_f1:.4f}")
        print(f"Validation Metrics: Acc={val_metrics['accuracy']:.4f},
            Prec={val_metrics['precision']:.4f},
            Rec={val_metrics['recall']:.4f}, F1={val_metrics['f1']:.4f}")
        print(f"Learning Rate: {current_lr}")

        # Early stopping check
        if val_loss < best_val_loss:
            best_val_loss = val_loss
            best_model_state = model.state_dict().copy()
            patience_counter = 0
            torch.save(best_model_state, 'best_fcnn_model.pth')
            print(" Saved new best model")
        else:
            patience_counter += 1
            if patience_counter >= patience:
                print("\nEarly stopping triggered!")
                model.load_state_dict(best_model_state)
                break

        scheduler.step(val_metrics['accuracy'])

    return metrics_history
```

6 Conclusion

This project developed a fake-news classifier and evaluated a range of machine learning models, from baseline approaches to advanced architectures, to examine how model complexity influences performance. The results highlight the importance of leveraging sophisticated techniques to capture nuanced patterns in textual data, paving the way for more reliable and scalable solutions to combat misinformation.

References

[1] Sameer Patel, *Fake News Detection*. (Database) https://www.kaggle.com/code/therealsampat/fake-news-detection

[2] Saurabh Shahane, *Fake News Classification - WELFake*. (database) https://www.kaggle.com/datasets/saurabhshahane/fake-news-classification

[3] *Pre-trained Word2vec Embedding*. (embedding) https://github.com/mmihaltz/word2vec-GoogleNews-vectors