INFS 5144 - Data Wrangling and Social Media Analytics

Assignment 2

> *Student ID: 110416080*
>
> *Name: Anisha Mariam Abraham*
>
> *Date: 23rd September, 2024*

Link to Notebook:
https://colab.research.google.com/drive/1jbbGo_9ITS4vH4ysSH1bcxDsaTI9pLph?usp=sharing

Word2Vec

Introduction

Word embedding is a technique in natural language processing (NLP) used to represent words as numeric vectors in a continuous vector space. The main purpose of word embedding is to capture the semantic and syntactic relationships between words so that similar words have similar vector representations. This allows computers to better understand and process text-based content.

In traditional text analysis, words were often represented using techniques like one-hot encoding, which fails to capture meaningful relationships between words. Word embedding, on the other hand, provides dense, low-dimensional representations that preserve the context of words, meaning that words with similar meanings will have vectors that are close to each other in the vector space.

The key goal of word embedding in NLP is to enable machine learning models to work more effectively with text by incorporating word meanings, relationships, and context, which improves the model's ability to process and understand language. Techniques such as Word2Vec, developed by Google in 2013, efficiently learn these embeddings by analyzing large text corpora, allowing models to infer relationships between words based on their usage.

Setting Up the Context Window and Training Data

In Word2Vec, the **context window** defines how many neighboring words around the target word are considered as context. For a **context window size of 1**, only one word on either side of the target word is used as context. Word pairs are created with a target word and its corresponding context word.

For the sentence: *'I like Data Science because Data Science is very cool'*

We form the word pairs as follows:

Word Pairs:

1. **Target word**: 'I'
   - **Context word(s)**: 'like'
   - Pair: ('I', 'like')
2. **Target word**: 'like'
   - **Context word(s)**: 'I', 'Data'
   - Pairs: ('like', 'I'), ('like', 'Data')
3. **Target word**: 'Data'
   - **Context word(s)**: 'like', 'Science'
   - Pairs: ('Data', 'like'), ('Data', 'Science')
4. **Target word**: 'Science'
   - **Context word(s)**: 'Data', 'because'
   - Pairs: ('Science', 'Data'), ('Science', 'because')
5. **Target word**: 'because'
   - **Context word(s)**: 'Science', 'Data'
   - Pairs: ('because', 'Science'), ('because', 'Data')
6. **Target word**: 'Data'
   - **Context word(s)**: 'because', 'Science'
   - Pairs: ('Data', 'because'), ('Data', 'Science')
7. **Target word**: 'Science'
   - **Context word(s)**: 'Data', 'is'
   - Pairs: ('Science', 'Data'), ('Science', 'is')
8. **Target word**: 'is'
   - **Context word(s)**: 'Science', 'very'
   - Pairs: ('is', 'Science'), ('is', 'very')
9. **Target word**: 'very'
   - **Context word(s)**: 'is', 'cool'
   - Pairs: ('very', 'is'), ('very', 'cool')
10. **Target word**: 'cool'
    - **Context word(s)**: 'very'
    - Pair: ('cool', 'very')

In **Word2Vec (Skip-Gram model)**, the goal is to predict the context words given a target word. So, the word pairs are used to train the model, where the target word is used as input, and the model tries to predict its context word(s). For example, if the

target word is 'Data', the model will try to predict context words such as 'like' and 'Science'. Similarly, if the target word is 'is', it will predict 'Science' and 'very'.

Hence, the **training data** looks like below:

Index	Main	Context	Label
1	like	Data	1
2	Data	like	1
3	Data	Science	1
4	Science	Data	1
5	like	very	0
6	Science	because	1
7	because	Science	1
8	because	Data	1
9	Data	because	1
10	because	cool	0
11	Science	very	1
12	very	Science	1
13	very	cool	1
14	cool	very	1

Label 1 indicates **positive samples** or in other words, samples that are true word pairs. **Label 0** indicates **negative samples** so that the model can learn from both positive and negative cases.

Initializing and Updating the Word2Vec Parameters

Skip-gram Pairs:

1. ("like", "Data")
2. ("Data", "like")
3. ("Data", "Science")
4. ("Science", "Data")
5. ("Science", "because")
6. ("because", "Data")
7. ("because", "Science")
8. ("Data", "because")
9. ("very", "Science")
10. ("data", "science")
11. ("science", "very")
12. ("very", "cool")

Initial Embeddings:

The dimensionality of word embeddings specifies how many numerical values (coordinates) will represent each word in the vector space. In this case, the dimensionality is **2**, meaning each word will be represented by a 2D vector (i.e., each word will have two numerical values). Typically, word vectors are **initialized with small random values** before the model starts training.

Word	Initial Embedding (Main)	Initial Embedding (Context)
like	[0.2, 0.5]	[-0.3, 0.8]
data	[0.7, -0.4]	[0.5, -0.1]
science	[0.1, -0.6]	[0.4, 0.3]
because	[-0.5, 0.2]	[-0.2, 0.7]
very	[0.3, 0.4]	[0.6, -0.3]
cool	[0.8, -0.7]	[-0.4, 0.5]

Formulas:

1. For each (Main, Context) in Training Data (D):

   SCORE = Sigmoid(Vmain .Vcontext)

   ERROR = LABEL - SCORE

   Update Vmain, Vcontext

2. Sigmoid function:

   $0\ 1\,\mathrm{h}\ \frac{\mathrm{E}}{\mathrm{E4}}$

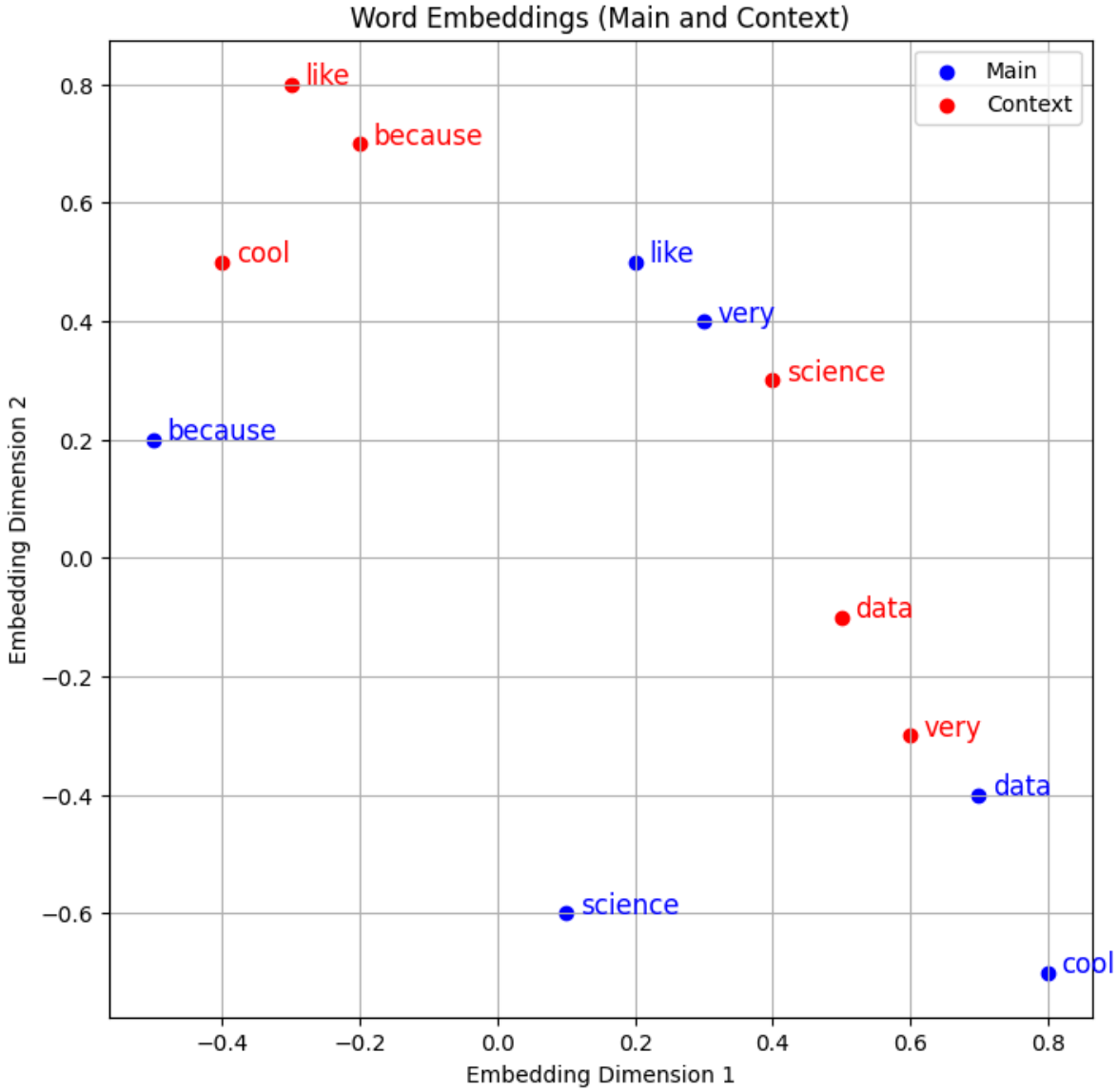
Updation condition:

For **positive samples**, we aim to move the **vectors closer** together, while for **negative samples**, we push the **vectors farther** apart. To reflect these adjustments, we modify the (x, y) coordinates by **adding or subtracting 0.1**, depending on whether the sample is positive or negative.

One Round of Iteration:

In [ ]:
```
# Vector Before

import matplotlib.pyplot as plt

# Define the words and their embeddings
words = ["like", "data", "science", "because", "very", "cool"]
main_embeddings = {
    "like": [0.2, 0.5],
    "data": [0.7, -0.4],
    "science": [0.1, -0.6],
    "because": [-0.5, 0.2],
    "very": [0.3, 0.4],
    "cool": [0.8, -0.7]
}
```

```
context_embeddings = {
    "like": [-0.3, 0.8],
    "data": [0.5, -0.1],
    "science": [0.4, 0.3],
    "because": [-0.2, 0.7],
    "very": [0.6, -0.3],
    "cool": [-0.4, 0.5]
}

# Create a plot
plt.figure(figsize=(8, 8))

# Plot main embeddings (blue)
for word, emb in main_embeddings.items():
    plt.scatter(emb[0], emb[1], color='blue', label='Main' if word == "like'
    plt.text(emb[0]+0.02, emb[1], word, fontsize=12, color='blue')

# Plot context embeddings (red)
for word, emb in context_embeddings.items():
    plt.scatter(emb[0], emb[1], color='red', label='Context' if word == "li
    plt.text(emb[0]+0.02, emb[1], word, fontsize=12, color='red')

# Labels and title
plt.title("Word Embeddings (Main and Context)")
plt.xlabel("Embedding Dimension 1")
plt.ylabel("Embedding Dimension 2")

# Avoid label repetition
handles, labels = plt.gca().get_legend_handles_labels()
by_label = dict(zip(labels, handles))
plt.legend(by_label.values(), by_label.keys())

# Show plot
plt.grid(True)
plt.show()
```


Word Embeddings (Main and Context)
Main
Context
like
because
cool
like
very
science
because
data
very
data
science
cool
Embedding Dimension 1
Embedding Dimension 2
0.8
0.6
0.4
0.2
0.0
−0.2
−0.4
−0.6
−0.4
−0.2
0.0
0.2
0.4
0.6
0.8

Word Embedding

Before matrix (initial vectors)

Sentence = "I like Data Science because Data Science is very cool."

Main	x	y
like	0.2	0.5
data	0.7	-0.4
science	0.1	-0.6
because	-0.5	0.2
very	0.3	0.4
cool	0.8	-0.7

context	x	y
like	-0.3	0.8
data	0.5	-0.1
science	0.4	0.3
because	-0.2	0.7
very	0.6	-0.3
cool	-0.4	0.5

Iteration 1

I Pair : (like, data) label : 1

V_{like} (0.2, 0.5) V_{data} (0.5, -0.1)

Score = sigmoid ($V_{like} \cdot V_{data}$) --Formula(1)

= sigmoid (0.2 × 0.5 + 0.5 × -0.1)

= sigmoid (0.1 - 0.05)

= sigmoid (0.05)

= 0.512 -- Formula(2)

Error = 1 − 0.512 -- Formula(1)

= 0.488

Since they are positive sample, the vectors need to be brought closer. The below operation will bring the vectors closer in vector space.

Vlike (+0.1, −0.1) Vdata (−0.1, +0.1)

Vlike (0.3, 0.4) Vdata (0.4, 0)

II

Main	x	y	Context	x	y
like	0.3	0.4	like	−0.3	0.8
data	0.7	−0.4	data	0.4	0
science	0.1	−0.6	science	0.4	0.3
because	−0.5	0.2	because	−0.2	0.7
very	0.3	0.4	very	0.6	−0.3
cool	0.8	−0.7	cool	−0.4	0.5

Pair : (data, like) label : 1

Vdata(0.7, −0.4) Vlike (−0.3, 0.8)

Score = Sigmoid (Vdata . Vlike)

= Sigmoid (0.7x −0.3 + −0.4 x0.8)

= Sigmoid (−0.53)

= 0.370

Error = 1 − 0.370

= 0.63

Since they are positive sample, the vectors need to be brought closer in vector space.

V_{data} (−0.1, +0.1) V_{like} (+0.1, −0.1)
Vdata (0.6, −0.3) Vlike (−0.2, 0.7)

III

Main	x	y	Context	x	y
like	0.3	0.4	like	−0.2	0.7
data	0.6	−0.3	data	0.4	0
science	0.1	−0.6	science	0.4	0.3
because	−0.5	0.2	because	−0.2	0.7
very	0.3	0.4	very	0.6	−0.3
cool	0.8	−0.7	cool	−0.4	0.5

Pair : (data, science) label : 1

Vdata (0.6, −0.3) Vscience (0.4, 0.3)

score = sigmoid (Vdata . Vscience)
= sigmoid (0.6 × 0.4 + −0.3 × 0.3)
= sigmoid (0.15)
= 0.537

Error = 1 − 0.537
= 0.463

Since they are positive sample, the vectors need to be brought closer in vector space.

Vdata (−0.1, +0.1) Vscience (+0.1, −0.1)
Vdata (0.5, −0.2) Vscience (0.5, 0.2)

IV

Main	x	y	Context	x	y
like	0.3	0.4	like	-0.2	0.7
data	0.5	-0.2	data	0.4	0
science	0.1	-0.6	science	0.5	0.2
because	-0.5	0.2	because	-0.2	0.7
very	0.3	0.4	very	0.6	-0.3
cool	0.8	-0.7	cool	-0.4	0.5

Pair : (science, data) label : 1

$V_{science}(0.1, -0.6)$ $V_{data}(0.4, 0)$

Score = sigmoid ($V_{science} \cdot V_{data}$)
= sigmoid ($0.1 \times 0.4 - 0.6 \times 0$)
= sigmoid (0.04)
= 0.509

Error = 1 - 0.509
= 0.491

Since they are positive sample, the vectors need to be brought closer in vector space

$V_{science}(+0.1, +0.1)$ $V_{data}(-0.1, -0.1)$
$V_{science}(0.2, -0.5)$ $V_{data}(0.3, -0.1)$

V

Main	x	y	Context	x	y
like	0.3	0.4	like	-0.2	0.7
data	0.5	-0.2	data	0.3	-0.1
science	0.2	-0.5	science	0.5	0.2
because	-0.5	0.2	because	-0.2	0.7
very	0.3	0.4	very	0.6	-0.3
cool	0.8	-0.7	cool	-0.4	0.5

Pair : (like, very) Label : 0

V_{like} (0.3, 0.4) V_{very} (0.6, -0.3)

Score = sigmoid ($V_{like} \cdot V_{very}$)
= sigmoid (0.3 x 0.6 + 0.4 x -0.3)
= sigmoid (0.06)
= 0.514

Error = 0 - 0.514
= -0.514

Since they are negative samples, the vectors need to be far apart in the vector space.
V_{like} (-0.1, +0.1) V_{very} (+0.1, -0.1)
V_{like} (0.2, 0.5) V_{very} (0.7, -0.4)

VI

Main	x	y	Context	x	y
like	0.2	0.5	like	-0.2	0.7
data	0.5	-0.2	data	0.3	-0.1
science	0.2	-0.5	science	0.5	0.2
because	-0.5	0.2	because	-0.2	0.7
very	0.3	0.4	very	0.7	-0.4
cool	0.8	-0.7	cool	-0.4	0.5

Pair : (science, because) Label : 1

$V_{science}$ (0.2, -0.5) $V_{because}$ (-0.2, 0.7)

Score = sigmoid ($V_{science} \cdot V_{because}$)
= sigmoid (0.2 x -0.2 + -0.5 x 0.7)
= sigmoid (-0.75)

$= 0.321$

Error $= 1 - 0.321$

$= 0.679$

Since they are positive samples, the vectors need to be brought closer in vector space.

$V_{science}(-0.1, +0.1)$ $V_{because}(+0.1, -0.1)$

$V_{science}(0.1, -0.4)$ $V_{because}(-0.1, 0.6)$

VII

Main	x	y	Context	x	y
like	0.2	0.5	like	-0.2	0.7
data	0.5	-0.2	data	0.3	-0.1
science	0.1	-0.4	science	0.5	0.2
because	-0.5	0.2	because	-0.1	0.6
very	0.3	0.4	very	0.7	-0.4
cool	0.8	-0.7	cool	-0.4	0.5

Pair: (because, science) label : 1

$V_{because}(-0.5, 0.2)$ $V_{science}(0.5, 0.2)$

Score $=$ sigmoid $(V_{because} \cdot V_{science})$

$=$ sigmoid $(-0.5 \times 0.5 + 0.2 \times 0.2)$

$=$ sigmoid $(-0.25 + 0.04)$

$=$ sigmoid (-0.21)

$= 0.447$

Error $= 1 - 0.447$

$= 0.553$

Since they are positive samples, the vectors need to be brought closer in vector space.

$V_{because}(+0.1, +0.1)$ $V_{science}(-0.1, +0.1)$
$V_{because}(-0.4, 0.3)$ $V_{science}(0.4, 0.3)$

VIII

Main	x	y	Context	x	y
like	0.2	0.5	like	-0.2	0.7
data	0.5	-0.2	data	0.3	-0.1
science	0.1	-0.4	science	0.4	0.3
because	-0.4	0.3	because	-0.1	0.6
very	0.3	0.4	very	0.7	-0.4
cool	0.8	-0.7	cool	-0.4	0.5

Pair: (because, data) Label: 1

$V_{because}(-0.4, 0.3)$ $V_{data}(0.3, -0.1)$

$$\begin{aligned} \text{Score} &= \text{sigmoid}(V_{because} \cdot V_{data}) \\ &= \text{sigmoid}(-0.4 \times 0.3 + 0.3 \times -0.1) \\ &= \text{sigmoid}(-0.15) \\ &= 0.462 \end{aligned}$$

$$\begin{aligned} \text{Error} &= 1 - 0.462 \\ &= 0.538 \end{aligned}$$

Since they are positive samples, the vectors need to be brought closer in vector space.

$V_{because}(+0.1, -0.1)$ $V_{data}(-0.1, +0.1)$
$V_{because}(-0.3, 0.2)$ $V_{data}(0.2, 0)$

IX

Main	x	y	Context	x	y
like	0.2	0.5	like	-0.2	0.7
data	0.5	-0.2	data	0.2	0
science	0.1	-0.4	science	0.4	0.3
because	-0.3	0.2	because	-0.1	0.6
very	0.3	0.4	very	0.7	-0.4
cool	0.8	-0.7	cool	-0.4	0.5

Pair : (data, because) Label : 1

$V_{data}(0.5, -0.2)$ $V_{because}(-0.1, 0.6)$

$$\begin{aligned} \text{score} &= \text{sigmoid}(V_{data} \cdot V_{because}) \\ &= \text{sigmoid}(0.5 \times -0.1 + -0.2 \times 0.6) \\ &= \text{sigmoid}(-0.05 - 0.12) \\ &= \text{sigmoid}(-0.14) \\ &= 0.457 \end{aligned}$$

$$\begin{aligned} \text{Error} &= 1 - 0.457 \\ &= 0.543 \end{aligned}$$

Since they are positive samples, the vectors need to be brought closer in vector space

$V_{data}(-0.1, +0.1)$ $V_{because}(+0.1, -0.1)$

$V_{data}(0.4, -0.1)$ $V_{because}(0, 0.5)$

Main	x	y	Context	x	y
like	0.2	0.5	like	-0.2	0.7
data	0.4	-0.1	data	0.2	0
science	0.1	-0.4	science	0.4	0.3
because	-0.3	0.2	because	0	0.5
very	0.3	0.4	very	0.7	-0.4
cool	0.8	-0.7	cool	-0.4	0.5

Pair : (because, cool) Label : 0

$V_{because}$ (-0.3, 0.2) V_{cool} (-0.4, 0.5)

score = sigmoid ($V_{because} \cdot V_{cool}$)
= sigmoid ($-0.3 \times -0.4 + 0.2 \times 0.5$)
= sigmoid (0.22)
= 0.554

Error = 0 - 0.554
= -0.554

Since they are negative samples, the vectors need to be brought farther in vector space.

$V_{because}$ (+0.1, -0.1) V_{cool} (-0.1, +0.1)
$V_{because}$ (-0.2, 0.1) V_{cool} (-0.5, 0.6)

XI

Main	x	y	Context	x	y
like	0.2	0.5	like	-0.2	0.7
data	0.4	-0.1	data	0.2	0
science	0.1	-0.4	science	0.4	0.3
because	-0.2	0.1	because	0	0.5
very	0.3	0.4	very	0.7	-0.4
cool	0.8	-0.7	cool	-0.5	0.6

Pair : (science, very) label : 1

$V_{science}$ (0.1, -0.4) V_{very} (0.7, -0.4)

$$\text{Score} = \text{sigmoid}(V_{science} \cdot V_{very})$$
$$= \text{sigmoid}(0.1 \times 0.7 - 0.4 \times -0.4)$$
$$= 0.477$$

$$\text{Error} = 1 - 0.477$$
$$= 0.523$$

Since they are positive samples, the vectors need to be brought closer in vector space.

$V_{science}$ (+0.1, +0.1) V_{very} (-0.1, +0.1)
$V_{science}$ (0.2, -0.3) V_{very} (0.6, -0.3)

XII

Main	x	y	Context	x	y
like	0.2	0.5	like	-0.2	0.7
data	0.4	-0.1	data	0.2	0
science	0.2	-0.3	science	0.4	0.3
because	-0.2	0.1	because	0	0.5
very	0.3	0.4	very	0.6	-0.3
cool	0.8	-0.7	cool	-0.5	0.6

Pair : (very, science) Label : 1

V_{very} (0.3, 0.4) $V_{science}$ (0.4, 0.3)

Score = sigmoid($V_{very} \cdot V_{science}$)
= sigmoid(0.3 × 0.4 + 0.4 × 0.3)
= sigmoid(0.24)
= 0.559

Error = 1 − 0.559
= 0.441

Since they are positive samples, the vectors need to be brought closer in vector space.

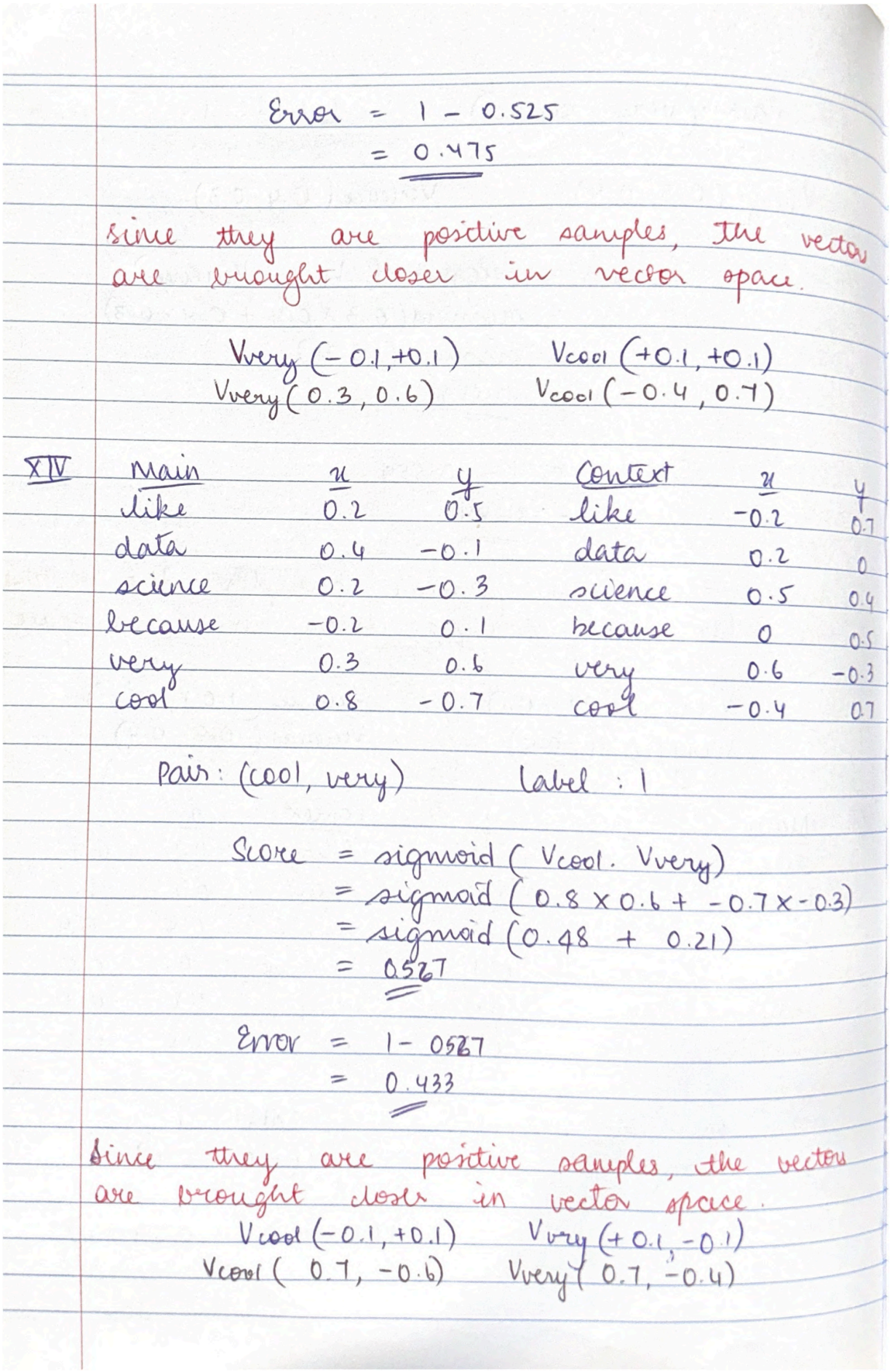
V_{very} (+0.1, +0.1) $V_{science}$ (+0.1, +0.1)
V_{very} (0.4, 0.5) $V_{science}$ (0.5, 0.4)

XIII

Main	x	y	Context	x	y
like	0.2	0.5	like	-0.2	0.7
data	0.4	-0.1	data	0.2	0
science	0.2	-0.3	science	0.5	0.4
because	-0.2	0.1	because	0	0.5
very	0.4	0.5	very	0.6	-0.3
cool	0.8	-0.7	cool	-0.5	0.6

Pair : (very, cool) Label : 1
V_{very} (0.4, 0.5) V_{cool} (−0.5, 0.6)

Score = sigmoid(0.4 × −0.5 + 0.5 × 0.6)
= sigmoid(0.1)
= 0.525



Error = 1 − 0.525
= 0.475

since they are positive samples, the vectors are brought closer in vector space.

Vvery (−0.1, +0.1) Vcool (+0.1, +0.1)
Vvery (0.3, 0.6) Vcool (−0.4, 0.7)

XIV

main	x	y	context	x	y
like	0.2	0.5	like	−0.2	0.7
data	0.4	−0.1	data	0.2	0
science	0.2	−0.3	science	0.5	0.4
because	−0.2	0.1	because	0	0.5
very	0.3	0.6	very	0.6	−0.3
cool	0.8	−0.7	cool	−0.4	0.7

Pair: (cool, very) label : 1

Score = sigmoid (Vcool . Vvery)
= sigmoid (0.8 × 0.6 + −0.7 × −0.3)
= sigmoid (0.48 + 0.21)
= 0.567

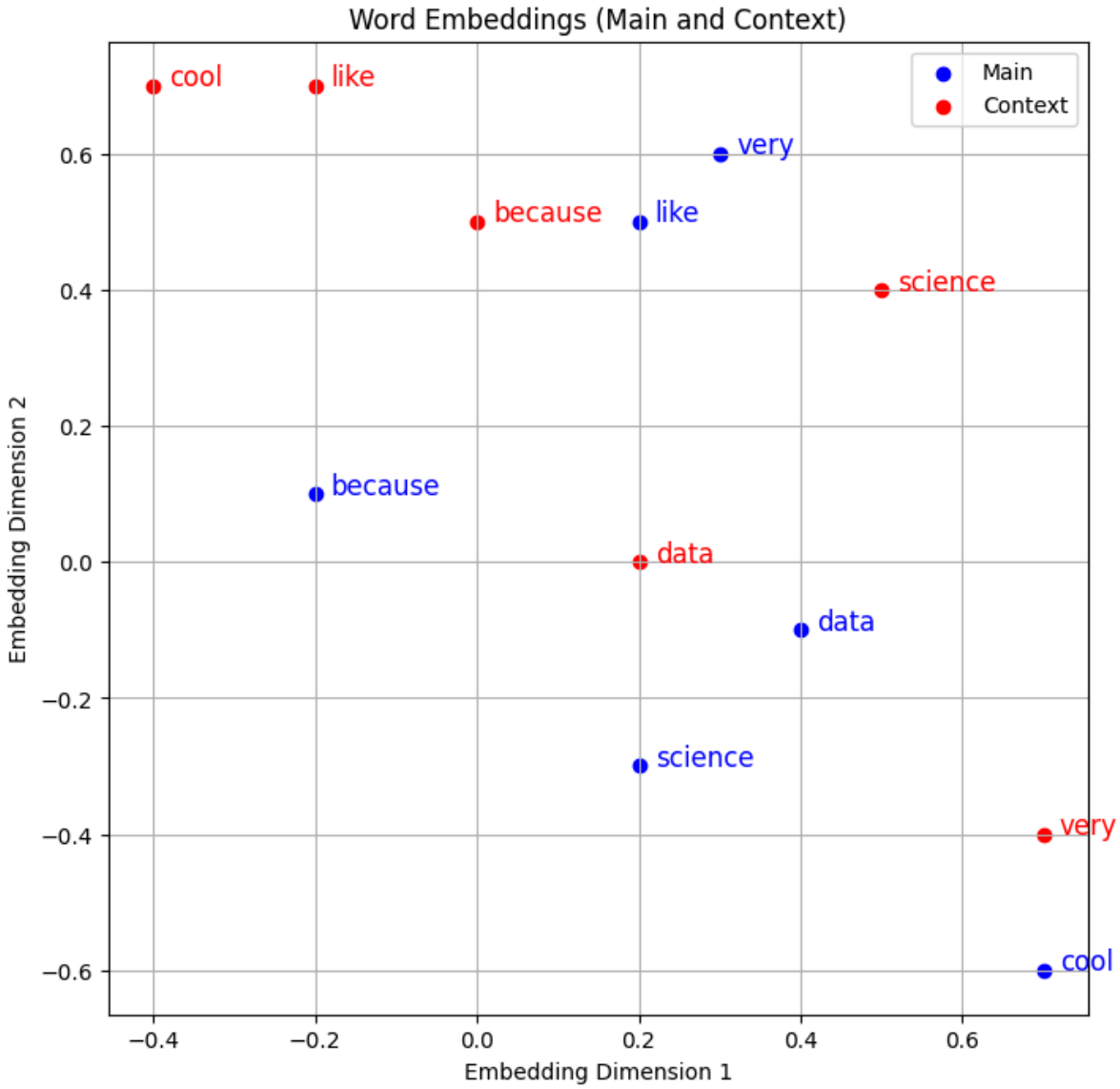
Error = 1 − 0.567
= 0.433

Since they are positive samples, the vectors are brought closer in vector space.
Vcool (−0.1, +0.1) Vvery (+0.1, −0.1)
Vcool (0.7, −0.6) Vvery (0.7, −0.4)

Final Vectors and Their Interpretation

In [ ]:
```
import matplotlib.pyplot as plt

# Define the words and their embeddings
words = ["like", "data", "science", "because", "very", "cool"]
```

```
main_embeddings = {
    "like": [0.2, 0.5],
    "data": [0.4, -0.1],
    "science": [0.2, -0.3],
    "because": [-0.2, 0.1],
    "very": [0.3, 0.6],
    "cool": [0.7, -0.6]
}
context_embeddings = {
    "like": [-0.2, 0.7],
    "data": [0.2, 0.0],
    "science": [0.5, 0.4],
    "because": [0, 0.5],
    "very": [0.7, -0.4],
    "cool": [-0.4, 0.7]
}

# Create a plot
plt.figure(figsize=(8, 8))

# Plot main embeddings (blue)
for word, emb in main_embeddings.items():
    plt.scatter(emb[0], emb[1], color='blue', label='Main' if word == "like'
    plt.text(emb[0]+0.02, emb[1], word, fontsize=12, color='blue')

# Plot context embeddings (red)
for word, emb in context_embeddings.items():
    plt.scatter(emb[0], emb[1], color='red', label='Context' if word == "li
    plt.text(emb[0]+0.02, emb[1], word, fontsize=12, color='red')

# Labels and title
plt.title("Word Embeddings (Main and Context)")
plt.xlabel("Embedding Dimension 1")
plt.ylabel("Embedding Dimension 2")

# Avoid label repetition
handles, labels = plt.gca().get_legend_handles_labels()
by_label = dict(zip(labels, handles))
plt.legend(by_label.values(), by_label.keys())

# Show plot
plt.grid(True)
plt.show()
```



In the word embedding plots, we can perform a comparison between two stages ("before" and "after"). Both plots show a 2D visualization of word embeddings, where the words are projected onto two dimensions (Embedding Dimension 1 and Embedding Dimension 2) to make the high-dimensional embeddings visually interpretable.

Key observations:

Both plots show words in two groups labeled as **"Main" (in blue)** and **"Context" (in red)**. The positions of the words have shifted slightly between the two plots, indicating that there has been a change in the embeddings, as a result of Iteration 1.

Words like "cool", "like", "science", "data" are present in both plots, but their relative distances have changed. In the "after" plot, words like "science" and "data" are brought closer to each other in both "Main" and "Context" groups, reflecting their semantic similarity.

In the "after" plot, words like "very" and "like" in the "Main" embeddings (blue) seem to have moved closer together. Similarly, the contextual embeddings of "very" and "like" (in red) have shifted closer to each other compared to the "before" plot.

The relative positioning of words like "because" and "cool" has remained somewhat stable, but there is a noticeable shift in how the word "data" and "very" relate to others. The proximity of "data" to "science" highlights how these words are contextually connected.

Perceptron Model

Introduction

A **perceptron** is the simplest type of artificial neural network used for binary classification tasks. It consists of a **single layer** of neurons (or nodes) and operates by taking a set of input values, applying weights to them, summing these weighted inputs, and then passing the result through an activation function (typically a step function) to produce a **binary output (either 0 or 1)**. The perceptron is foundational in machine learning as it introduces the concept of learning through adjustments of weights based on errors from predictions. In essence, the perceptron model mimics the way a biological neuron functions, using a straightforward approach to learning. By iteratively adjusting the weights using techniques like the perceptron learning rule, it tries to correctly classify inputs into one of two classes.

Setting up the Model

Time spent per week (h)	Course website visits per week	Pass/Fail(1/0)
10	15	1
15	12	1
6	5	0
8	8	0

To prepare the above data for training the Perceptron model, **Linear Normalization**, also known as the **"Min-Max" scaling method**, is used. This technique rescales the features to a range **between 0 and 1**.

The formula for normalization is:

$$\mathrm{h} \quad \text{————}$$

Where:

X is the original value,

is the **minimum value** of the feature,

is the **maximum value** of the feature,

is the **normalized value**.

For instance, if the time spent per week (in hours) for Student 1 is 10 hours, with a maximum of 15 hours and a minimum of 6 hours, the normalized value is calculated as follows:

$$\text{h} \frac{\text{E=}\ ]}{\text{E[}\ \ ]}\ \text{h}\ \frac{\text{T}}{\text{d}} \quad =9\text{TTT}$$

Using this method, the normalized data for the features "Time spent per week (h)", "Course website visits per week", and "Pass/Fail (1/0)" is as follows:

Time spent per week (h)	Course website visits per week	Pass/Fail(1/0)
0.444	1	1
1	0.7	1
0	0	0
0.222	0.3	0

Initialization and Assumptions

Formulas:

1. Perceptron model equation:

$$\text{h} \quad \Big) \quad _{\text{h E}} \qquad 4$$

Where:

is the **output** of the perceptron.

are the **weights**.

are the **input** features.

is the **bias** term.

is the **activation** function.

1. Binary step activation function:

$$0\ \ 1\,\text{h} \quad \begin{matrix} \text{E} & & = \\ = & \text{g} & = \end{matrix}$$

1. Error:

$$\text{t} \qquad \text{h} \qquad \approx \quad \lhd\text{u} \qquad \approx \qquad \approx \quad \lhd\text{u}$$

1. Updation:

For weights,

$$\text{h} \qquad 4$$

For bias,

$$\mathrm{h} \quad 4$$

Assumptions:

1. **Weights:** It can be assumed as 0.
2. **Bias:** It can be assumed as 0.
3. **Learning Rate:** It can be assumed as 0.1.

This model uses **Binary Step Activation Function** defined as follows:

$$0\ 1\mathrm{h} \quad \begin{matrix} \mathrm{E} & = \\ = & \mathrm{g}\ = \end{matrix}$$

If the node input value is less than 0, it returns 0 as output. Else, it returns 1. I had to run a total of **3 iterations** to get the Perceptron model with optimum parameters.

Model Training

PERCEPTRON

Time spent (hours) (pw)	Course visits (pw)	Pass / fail (1/0)
0.444	1	1
1	0.7	1
0	0	0
0.222	0.3	0

Iteration 1 :

Assumptions

$w_1 = 0 \qquad w_2 = 0 \qquad b = 0$

Student 1 $\Rightarrow$ $y = f(0 \times 0.444 + 0 \times 1 + 0)$ --- Formula (1)

$= f(0 + 0 + 0)$

$= f(0)$

$= 0$ --- Formula (2)

error $= 1 - 0$ --- Formula (3)

$= \emptyset 1$

update : $w_1 = 0 + 0.1 \times 1 \times 0.444$ --- Formula (4)

$= 0.044$

Since there is an error, we HAVE to update the weights and bias.

$w_2 = 0 + 0.1 \times 1 \times 1$

$= 0.1$

$b = 0 + 0.1 \times 1$

$= 0.1$

Assumptions

$W_1 = 0.044$ $W_2 = 0.1$ $b = 0.1$

Student 2 $\Rightarrow$ $y = f(0.044 \times 1 + 0.1 \times 0.7 + 0.1)$
$= f(0.044 + 0.07 + 0.1)$
$= f(0.214)$
$= 1$

error $= 1 - 1$
$= 0$

Since there is no error, we do not update weights & bias.

update: $W_1 = 0.044$
$W_2 = 0.1$
$b = 0.1$

Assumptions

$W_1 = 0.044$ $W_2 = 0.1$ $b = 0.1$

Student 3 $\Rightarrow$ $y = f(0.044 \times 0 + 0.1 \times 0 + 0.1)$
$= f(0.1)$
$= 1$

error $= 0 - 1$
$= -1$

Since there is an error, we need to update weights & bias.

update: $W_1 = 0.044 + 0.1 \times -1 \times 0$
$= 0.044$

$W_2 = 0.1 + 0.1 \times -1 \times 0$
$= 0.1$

-2-

$b = 0.1 + 0.1 \times -1$

$= 0$

Assumptions

$w_1 = 0.044 \quad w_2 = 0.1 \quad b = 0$

Student 4 $\Rightarrow$ $y = f(0.044 \times 0.22 + 0.1 \times 0.3 + 0)$

$= f(0.0399)$

$= 1$

$error = 0 - 1$

$= -1$

update : $w_1 = 0.044 + 0.1 \times -1 \times 0.22$

$= 0.022$

since there is an error, we have to update weights & bias. $w_2 = 0.1 + 0.1 \times -1 \times 0.3$

$= 0.07$

$b = 0 + 0.1 \times -1$

$= -0.1$

Iteration 2 :

Assumptions :

$w_1 = 0.022 \quad w_2 = 0.07 \quad b = -0.1$

Student 1 $\Rightarrow$ $y = f(0.022 \times 0.444 + 0.07 \times 1 - 0.1)$

$= f(-0.02)$

$= 0$

error = 1 − 0

= 1

update: $w_1 = 0.022 + 0.1 \times 1 \times 0.44$

$= 0.0667$

Since there is an error, update the weights & bias

$w_2 = 0.07 + 0.1 \times 1 \times 1$

$= 0.17$

$b = -0.1 + 0.1 \times 1$

$= 0$

Assumptions

$w_1 = 0.0667$ $w_2 = 0.17$ $b = 0$

Student 2 ⟹ $y = f(0.0667 \times 1 + 0.17 \times 0.7 + 0)$

$= f(0.1857)$

$= 1$

error = 1 − 1

= 0

Since there is no error, no update to weights and bias.

update: $w_1 = 0.0667$

$w_2 = 0.17$

$b = 0$

Assumptions

$w_1 = 0.0667$ $w_2 = 0.17$ $b = 0$

-4-

Student 3 $\Rightarrow$ $y = f(0.0667 \times 0 + 0.17 \times 0 + 0)$
$= f(0)$
$= 0$

error $= 0 - 0$
$= 0$

Since there is no error, no update to weights & bias.

update: $w_1 = 0.0667$
$w_2 = 0.17$
$b = 0$

Assumptions

$w_1 = 0.0667$ $w_2 = 0.17$ $b = 0$

Student 4 $\Rightarrow$ $y = f(0.0667 \times 0.222 + 0.17 \times 0.3 + 0)$
$= f(0.0658)$
$= 1$

error $= 0 - 1$
$= -1$

update: $w_1 = 0.0667 + 0.1 \times -1 \times 0.222$
$= 0.044$

Since there is an error, update the weights & bias.

$w_2 = 0.17 + 0.1 \times -1 \times 0.3$
$= 0.14$

$b = 0 + 0.1 \times -1$
$= -0.1$

-5-

Iteration 3 :

Assumptions

$w_1 = 0.044$ $w_2 = 0.14$ $b = -0.1$

Student 1 $\Rightarrow$ $y = f(0.044 \times 0.444 + 0.14 \times 1 - 0.1)$
$= f(0.059)$
$= 1$

error $= 1 - 1 = 0$ Since there is no error, no update.

update : $w_1 = 0.044$
$w_2 = 0.14$
$b = -0.1$

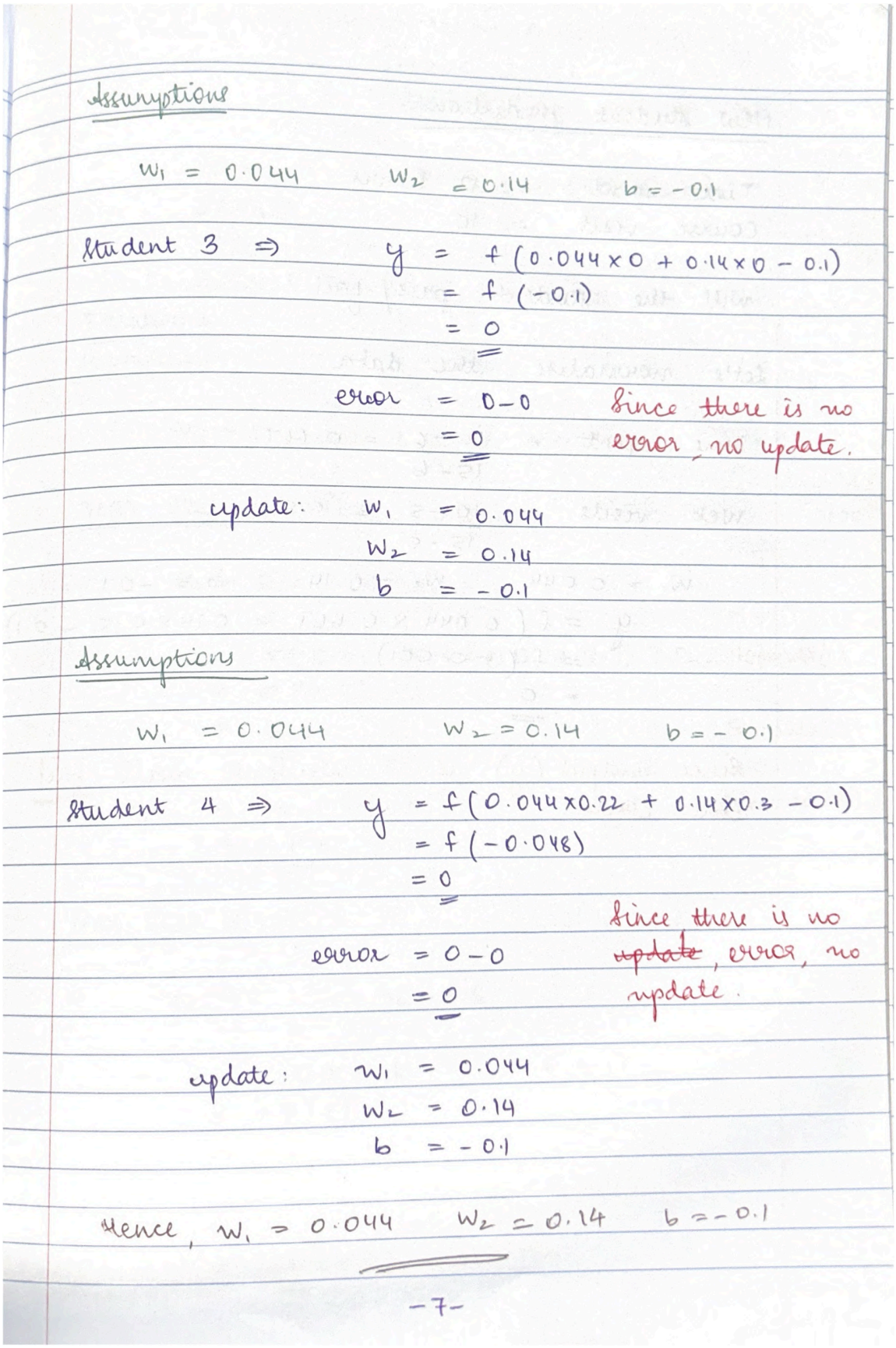
Assumptions

$w_1 = 0.044$ $w_2 = 0.14$ $b = -0.1$

Student 2 $\Rightarrow$ $y = f(0.044 \times 1 + 0.14 \times 0.7 - 0.1)$
$= f(0.042)$
$= 1$

error $= 1 - 1$ Since there is no error, no update.
$= 0$

update : $w_1 = 0.044$
$w_2 = 0.14$
$b = -0.1$



Assumptions

$w_1 = 0.044$ $w_2 = 0.14$ $b = -0.1$

Student 3 ⇒ $y = f(0.044 \times 0 + 0.14 \times 0 - 0.1)$
$= f(-0.1)$
$= 0$

error $= 0 - 0$
$= 0$

Since there is no error, no update.

update: $w_1 = 0.044$
$w_2 = 0.14$
$b = -0.1$

Assumptions

$w_1 = 0.044$ $w_2 = 0.14$ $b = -0.1$

Student 4 ⇒ $y = f(0.044 \times 0.22 + 0.14 \times 0.3 - 0.1)$
$= f(-0.048)$
$= 0$

error $= 0 - 0$
$= 0$

Since there is no ~~update~~, error, no update.

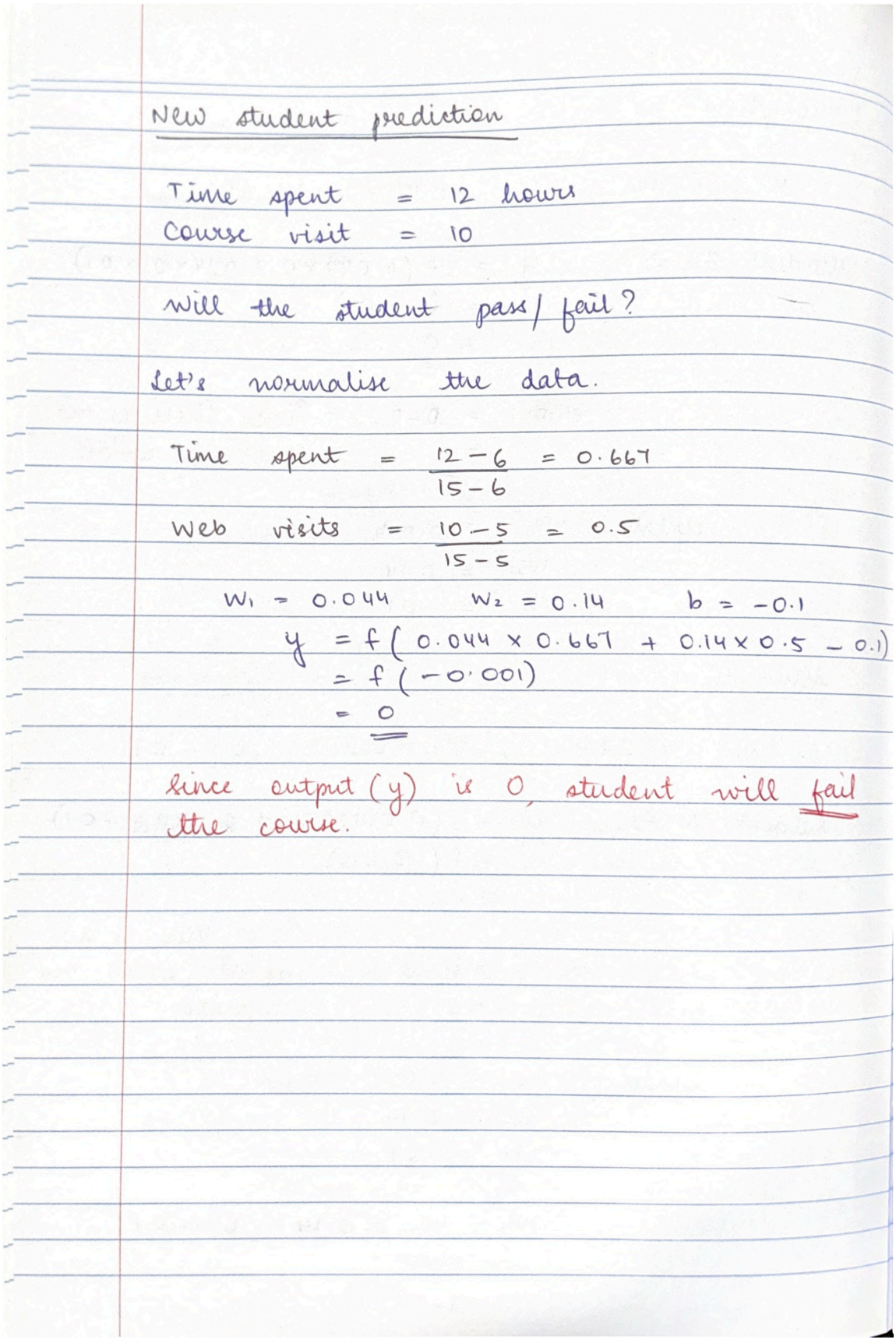
update: $w_1 = 0.044$
$w_2 = 0.14$
$b = -0.1$

Hence, $w_1 = 0.044$ $w_2 = 0.14$ $b = -0.1$

-7-

Model Prediction

To check whether a new student would pass or fail the course, we can run the details past our trained model.



In the prediction model, the task is to determine whether a student will pass or fail a course based on two key features: time spent and course visits.

To do this, the input features are first **normalized** to bring them within a comparable scale. The normalized values are then used in a **linear model**, where **each feature is multiplied by its respective weight**, representing the importance of that feature in the prediction. A **bias term** is also added to adjust the overall prediction. The final value is

calculated . In this case, the computed value is **slightly negative**, leading the model to predict a failure, since the activation function likely operates as a threshold where negative values result in a **fail prediction** (output 0).

Therefore, the model concludes that the student is expected to **fail** the course based on the provided data.

RNN

Introduction

Recurrent Neural Networks (RNNs) are a class of artificial neural networks designed for processing sequences of data. Unlike traditional feedforward neural networks, RNNs have connections that form directed cycles, allowing them to **maintain a form of memory** by retaining information from previous time steps. This temporal characteristic makes RNNs particularly well-suited for tasks involving sequential data, such as **time series prediction, natural language processing, and speech recognition**. By utilizing their internal state to capture patterns and dependencies over time, RNNs can model complex sequences and learn from context, enabling them to make predictions or generate outputs that are informed by the entire sequence of input data.

Explanation and Assumptions for RNN Training

To predict the house price in Mawson Lakes for the year 2025 using a Recurrent Neural Network (RNN), we can utilise the RNN's structure.

Input Layer: The RNN receives the historical house prices as input. In this case, the input sequence consists of the average house prices for the last three years: 1.1 million, 1.2 million, and 1.0 million dollars.

Recurrent Layer: The core of the RNN is the recurrent layer, which processes the input data sequentially. This layer consists of units that maintain a hidden state, which gets updated at each time step based on the current input and the previous hidden state. The RNN's hidden state acts as a form of memory, capturing the temporal dependencies and patterns in the data.

Hidden State: At each time step, the hidden state of the RNN is updated based on the input value (house price for that year) and the previous hidden state.

Output Layer: After processing the sequence of house prices, the RNN produces an output. For predicting a future value, such as the house price in 2025, the network's final hidden state is used to generate the forecast. This output layer often consists of a dense layer with a linear activation function that maps the hidden state to a continuous value, representing the predicted house price.

Setting up the Model

Year	House Price (in millions)
2022	1.1
2023	1.2
2024	1.0

To prepare the above data for training the RNN model, **Linear Normalization**, also known as the **"Min-Max" scaling method**, is used. This technique rescales the features to a range **between 0 and 1**.

The formula for normalization is:

$$h \quad \text{————}$$

Where:

X is the original value,

is the **minimum value** of the feature,

is the **maximum value** of the feature,

is the **normalized value**.

For instance, if the house price is 1.1, the normalized value is calculated as follows:

$$h \quad \frac{E9E \quad E9=}{E9F \quad E9=} h \quad \frac{=9E}{=9F} \qquad =9[$$

Using this method, the normalized data for the features "Year", "House Price" is as follows:

Year	House Price (in millions)
2022	0.5
2023	1
2024	0

Formulas:

1. In an RNN, the hidden state at time step is computed as follows:

$$h \quad 0 \qquad {}_{E}4 \qquad 4 \quad 1$$

where:

- is the **hidden state** at time step (t)
- is the **weight matrix** for the hidden state
- is the **weight matrix** for the input
- is the **input** at time step (t)

- is the **bias** term

1. The output at time step is computed as:

$$\mathrm{h} \quad 4$$

where:

- is the **weight matrix** for the output
- is the **bias term** for the output

1. Activation function:

The model uses a **linear activation function** in the output layer, allowing it to predict continuous house prices directly based on the weighted sum of the hidden state.

$$0\ 1\mathrm{h}$$

1. Prediction formula:

$$_4\mathrm{E}\,\mathrm{h} \quad 4 \quad 4$$

$$_4\mathrm{E}\,\mathrm{h} \quad _4\mathrm{E}\,4$$

Assumptions:

= 0

= 1

= -1

= 0.1

= 0

$_6$E = 0

Model Training

RNN

Year	House price (Normalised)
2022	0.5
2023	1
2024	0

Iteration 1

Assumptions :

$W_h = 0$ $W_x = 1$ $W_y = -1$ $b_h = 0.1$ $b_y = 0$

Year 2022 (t=0)

$x_0 = 0.5$ $h_{-1} = 0$

$h_0 = \sigma(0 \times 0 + 1 \times 0.5 + 0.1)$ -- Formula(1)

$= \sigma(0.6)$

$= 0.6$ -- Formula(3)

$y_0 = -1 \times 0.6 + 0$ -- Formula(2)

$= -0.6$

Year 2023 (t=1)

$x_0 = 1$ $h_0 = 0.6$

$h_1 = \sigma(0 \times 0.6 + 1 \times 1 + 0.1)$

$= \sigma(1.1)$

$= 1.1$

$y_1 = -1 \times 1.1 + 0$

$= -1.1$

Year 2024

$x_2 = 0 \qquad h_1 = 1.1$

$$h_2 = \sigma(0 \times 1.1 + 1 \times 0 + 0.1)$$
$$= \sigma(0.1)$$
$$= 0.1$$

$$y_2 = -1 \times 0.1 + 0$$
$$= -0.1$$

	2022	2023	2024
Actual	0.5	1	0
Predicted	-0.6	-1.1	-0.1

Since the predicted model values are far from the actual values, we can train the model again.

Keeping all values constant except $w_x = -0.5$ and $w_y = -0.5$.

Iteration 2

Assumptions

$w_h = 0 \qquad w_x = -0.5 \qquad w_y = -0.5 \qquad b_h = 0.1 \qquad b_y = 0$

Year 2022 (t=0)

$x_0 = 0.5 \qquad h_{-1} = 0$

$h_0 = \sigma(0 \times 0 + -0.5 \times 0.5 + 0.1)$
$= \sigma(-0.15)$
$= -0.15$

$y_0 = -0.5 \times -0.15 + 0$
$= 0.075$

Year 2023 (t=1)

$x_1 = 1 \qquad h_0 = -0.15$

$h_1 = \sigma(0 \times -0.15 + -0.5 \times 1 + 0.1)$
$= \sigma(-0.4)$
$= -0.4$

$y_1 = -0.5 \times -0.4 + 0$
$= 0.2$

Year 2024 (t=2)

$x_2 = 0 \qquad h_1 = -0.4$

$h_2 = \sigma(0 \times -0.4 + -0.5 \times 0 + 0.1)$
$= \sigma(0.1)$
$= 0.1$

$y_2 = -0.5 \times 0.1 + 0$
$= -0.05$

	2022	2023	2024
Actual	0.5	1	0
Predicted	0.075	0.2	-0.05

The predictions are closer now. We can try to improve the weights once more.

Keeping all values constant except $w_x = -1$, $w_y = -1$.

Iteration 3

Assumptions

$w_h = 0$ $w_x = -1$ $w_y = -1$ $b_h = 0.1$ $b_y = 0$

Year 2022 $(t=0)$

$x_0 = 0.5$ $h_{-1} = 0$

$$h_0 = \sigma(0 \times 0 + -1 \times 0.5 + 0.1)$$
$$= \sigma(-0.4)$$
$$= -0.4$$

$$y_0 = -1 \times -0.4 + 0$$
$$= 0.4$$

Year 2023 $(t=1)$

$x_1 = 1$ $h_0 = -0.4$

$h_1 = \sigma(0 \times -0.4 + -1 \times 0.5 + 0.1)$

$= \sigma(-0.9)$

$= -0.9$

$y_1 = -1 \times -0.9 + 0$

$= +0.9$

Year 2024 $(t = 2)$

$x_2 = 0 \qquad h_1 = -0.9$

$h_2 = \sigma(0 \times -0.9 + -1 \times 0 + 0.1)$

$= \sigma(0.1)$

$= 0.1$

$y_2 = -1 \times 0.1 + 0$

$= -0.1$

	2022	2023	2024
Actual	0.5	1	0
Predicted	0.4	0.9	-0.1

Since the predicted values are closer to the actual values, we can try another iteration.

Iteration 4

Assumptions

$w_h = 0 \quad w_x = -2 \quad w_y = -0.5 \quad b_h = 0 \quad b_y = 0$

Year 2022 $(t_0 = 0)$

$x_0 = 0.5 \quad h_{-1} = 0$

$h_0 = \sigma(0 \times 0 + \overset{-2}{\cancel{-0.5}} \times 0.5 + 0)$

$= \sigma(-1)$

$= -1$

$y_0 = -0.5 \times -1 + 0$

$= 0.5$

Year 2023 $(t=1)$

$x_1 = 1 \quad h_0 = -1$

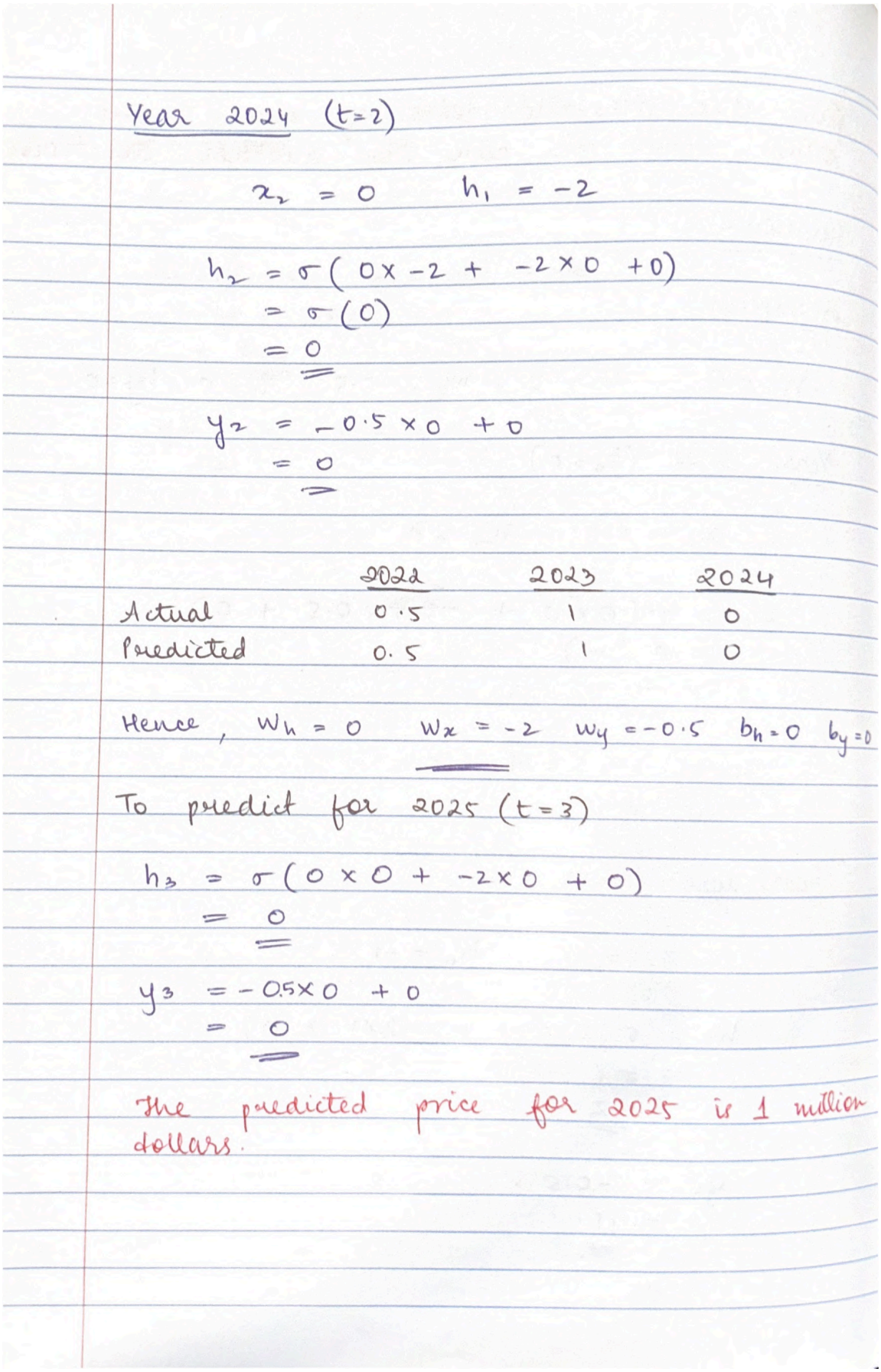
$h_1 = \sigma(0 \times -1 + -2 \times 1 + 0)$

$= \sigma(-2)$

$= -2$

$y_1 = -0.5 \times -2 + 0$

$= 1$



Year 2024 (t=2)

$x_2 = 0 \quad h_1 = -2$

$h_2 = \sigma(0 \times -2 + -2 \times 0 + 0)$
$= \sigma(0)$
$= 0$

$y_2 = -0.5 \times 0 + 0$
$= 0$

	2022	2023	2024
Actual	0.5	1	0
Predicted	0.5	1	0

Hence, $w_h = 0 \quad w_x = -2 \quad w_y = -0.5 \quad b_h = 0 \quad b_y = 0$

To predict for 2025 (t=3)

$h_3 = \sigma(0 \times 0 + -2 \times 0 + 0)$
$= 0$

$y_3 = -0.5 \times 0 + 0$
$= 0$

The predicted price for 2025 is 1 million dollars.

Prediction of House Price in 2025

Now as our model looks ready with the weights and biases, we can try running it past a new data. Using **Formula 4**, the prediction for housing prices in Mawson Lakes for 2025 has been done above.

Plugging the weights and biases for the prediction iteration and using the output in the last training iteration for the prediction iteration helped us in finding the **predicted house price of 1 million dollars in Mawson Lakes**.

LSTM

Introduction

Long Short-Term Memory (LSTM) networks are a specialized type of Recurrent Neural Network (RNN) designed to address the challenges of learning long-term dependencies in sequential data. Introduced by Hochreiter and Schmidhuber in 1997, LSTMs incorporate a unique architecture with memory cells and gating mechanisms that enable them to maintain and control information over extended periods. This allows LSTMs to effectively capture long-range dependencies and mitigate issues such as vanishing and exploding gradients that often affect standard RNNs. Due to their enhanced capacity to remember and forget information, LSTMs are widely used in applications involving complex sequences, such as language modeling, speech recognition, and time series forecasting, where capturing context and maintaining temporal relationships are crucial for accurate predictions.

Explanation and Assumptions for LSTM Training

Cell State: The LSTM introduces a cell state, a long-term memory component that carries information across long sequences. This cell state is updated via a series of gates and helps the network remember information for extended periods, overcoming the vanishing gradient problem that often affects traditional RNNs.

Gates:

- **Forget Gate**: This gate decides what portion of the cell state to discard. It takes the current input and the previous hidden state to generate a value between 0 and 1 for each element in the cell state, where 0 means "completely forget" and 1 means "completely retain."
- **Input Gate**: This gate determines which new information to add to the cell state. It consists of two parts: a sigmoid function that decides which values to update, and a tanh function that creates new candidate values for the cell state.
- **Output Gate**: This gate decides what the next hidden state should be, which is used for predictions and also fed back into the network. It uses a sigmoid function to decide which parts of the cell state to output, combined with a tanh function to scale the output.

Hidden State:

- The hidden state in an LSTM is influenced by the cell state and the output gate. It is used for the network's output and the next time step's input.

Differences from Traditional RNN

1. **Handling Long-Term Dependencies**: Traditional RNNs struggle with long-term dependencies due to the vanishing gradient problem, where gradients become too small to update the weights effectively. LSTMs address this by using the cell state and gates to maintain and update long-term information more effectively.

2. **Memory Management**: LSTMs have a more sophisticated memory management system than standard RNNs. The cell state and gates allow LSTMs to decide which information to keep or discard, making them better at handling sequences with long-term dependencies.

3. **Learning Capability**: Due to their ability to manage memory over longer sequences, LSTMs can learn more complex patterns from historical data. This is particularly useful for predicting future values in time series, such as house prices, where trends and seasonal patterns can span multiple years.

Setting up the Model

Year	House Price (in millions)
2022	1.1
2023	1.2
2024	1.0

To prepare the above data for training the RNN model, **Linear Normalization**, also known as the **"Min-Max" scaling method**, is used. This technique rescales the features to a range **between 0 and 1**.

The formula for normalization is:

h ————

Where:

X is the original value,

is the **minimum value** of the feature,

is the **maximum value** of the feature,

is the **normalized value**.

For instance, if the house price is 1.1, the normalized value is calculated as follows:

h h

Using this method, the normalized data for the features "Year", "House Price" is as follows:

Year	House Price (in millions)
2022	0.5
2023	1
2024	0

Formulas:

In an LSTM network, the main gates are the **Forget Gate**, **Input Gate**, and **Output Gate**, which are used to update the cell state and hidden state.

1. Forget Gate

The forget gate decides which information to discard from the previous cell state e $_{E}$:

$$h \quad 0 \qquad _{E}5 \quad 4 \quad 1$$

1. Input Gate

The input gate decides which information from the current input should be added to the cell state. It consists of two parts:

$$h \quad 0 \qquad _{E}5 \quad 4 \quad 1$$

- The candidate cell state:

$$e \quad h \qquad 0 \; _{e} \qquad _{E}5 \quad 4 \; _{e}1$$

1. Cell State Update

The new cell state is a combination of the previous cell state and the new candidate cell state, controlled by the forget gate and input gate:

$$e \quad h \qquad e \; _{E}4 \qquad e$$

1. Output Gate

The output gate determines the hidden state, which is used for the output at the current time step:

$$h \quad 0 \qquad _{E}5 \quad 4 \quad 1$$

1. Hidden State

The hidden state is a filtered version of the cell state, scaled by the output gate:

$$h \qquad 0e \; 1$$

Components:

- : Input at time step .
- $_{E}$: Hidden state from the previous time step.
- e $_{E}$: Cell state from the previous time step.

- $5 \quad 5 \quad {}_{e}5$: Weight matrices for the forget, input, candidate cell, and output gates.
- $5 \quad 5 \; {}_{e}5$: Bias terms for each gate.
- : Sigmoid activation function.
- : Hyperbolic tangent function.

1. Activation function:

The sigmoid activation function, often used in neural networks, is defined as:

$$0 \; 1\mathrm{h} \; \frac{\mathrm{E}}{\mathrm{E}4}$$

where:

- is the input,
- is the base of the natural logarithm.

Assumptions:

= 1

= 1

= 1

= 1

= 0

= 0

= 0

= 0

e = 0

= 0

Model Training

LSTM

Assumptions :

$W_f = 1$ $b_f = 0$

$W_i = 1$ $b_i = 0$ $C_0 = 0$

$W_c = 1$ $b_c = 0$ $h_0 = 0$

$W_o = 1$ $b_o = 0$

Normalise data,

$x_1 = 1.1 \sim 0.5$

$x_2 = 1.2 \sim 1$

$x_3 = 1.0 \sim 0$

Iteration 1

$t = 1, \quad x_1 = 0.5$

1. Forget gate :

$$f_1 = \sigma(1(0 + 0.5) + 0)$$
$$= \sigma(0.5)$$
$$= 0.62$$

2. Input gate :

$$i_1 = \sigma(1(0 + 0.5) + 0)$$
$$= \sigma(0.5)$$
$$= 0.62$$

$$\bar{C}_1 = \tanh(1(0.5) + 0)$$
$$= 0.46$$

3. Update cell state

$$C_1 = 0.62 \times 0 + 0.62 \times 0.46$$
$$= 0.28$$

4. Output gate

$$O_1 = \sigma(1(0 + 0.5) + 0)$$
$$= \sigma(0.5)$$
$$= 0.62$$

$$h_1 = 0.62 \times \tanh(0.28)$$
$$= 0.16$$

$t = 2, \; x_2 = 1$

$C_1 = 0.28$
$h_1 = 0.16$

1. Forget gate:

$$f_2 = \sigma(1(0.16 + 1) + 0)$$
$$= \sigma(1.16)$$
$$= 0.76$$

2. Input gate:

$$i_2 = \sigma(1(0.16 + 1) + 0)$$
$$= \sigma(1.16)$$
$$= 0.76$$

$$\bar{C}_2 = \tanh(1(0.161) + 0)$$
$$= \tanh(1.16)$$
$$= 0.82$$

3. Update cell state

$$C_2 = 0.76 \times 0.28 + 0.76 \times 0.82$$
$$= 0.836$$

4. Output gate

$$O_2 = \sigma(1(0.16+1)+0)$$
$$= \sigma(1.16)$$
$$= 0.76$$

$$h_2 = 0.76 \times \tanh(0.836)$$
$$= 0.51$$

$t = 3, \; x_3 = 0$

$C_2 = 0.836$

$h_2 = 0.51$

1. Forget gate:

$$f_3 = \sigma(1(0.51+0)+0)$$
$$= \sigma(0.51)$$
$$= 0.62$$

2. Input gate:

$$i_3 = \sigma(1(0.51)+0)$$
$$= \sigma(0.51)$$
$$= 0.62$$

$$\tilde{C}_3 = \tanh(0.51) = 0.46$$

3. Update cell state

$$C_3 = 0.62 \times 0.836 + 0.62 \times 0.46$$
$$= 0.8$$

4. Output gate:

$$O_3 = \sigma(1(0.51) + 0)$$
$$= 0.62$$

$$h_3 = 0.62 \times \tanh(0.8)$$
$$= 0.41$$

	t=1	t=2	t=3
Actual	0.5	1	0
Predict	0.16	0.51	0.41

$W_f = 1$ $b_f = 1$
$W_i = 1$ $b_i = 1$
$W_c = 1$ $b_c = 1$
$W_o = 1$ $b_o = 1$

} update bias by 1.

Iteration 2

t=1 $x_1 = 0.5$

1. Forget gate

$$f_1 = \sigma(1(0 + 0.5) + 1)$$
$$= \sigma(1.5)$$
$$= 0.81$$

2. Input gate :

$$i_1 = \sigma(1.5) = 0.81$$

$$\bar{C}_1 = \tanh(1.5) = 0.9$$

3. Update cell state

$$C_1 = 0.81 \times 0 + 0.81 \times 0.9 = 0.729$$

4. Output gate

$$O_1 = \sigma(1.5) = 0.81$$

$$h_1 = 0.81 \times \tanh(0.729) = 0.5$$

$t = 2 \quad x_2 = 1$

1. Forget gate :

$$f_2 = \sigma(1.5 + 1) = \sigma(2.5) = 0.9$$

2. Input gate :

$$i_2 = \sigma(2.5) = 0.9$$

$\bar{C}_2 = \tanh(2.5)$

$= 0.98$

3. Update cell state:

$C_2 = 0.9 \times 0.729 + 0.9 \times 0.98$

$= 1.54$

4. Output gate:

$O_2 = \sigma(2.5)$

$= 0.9$

$h_2 = 0.9 \times \tanh(1.54)$

$= 0.82$

$t=3, x_3=0$

1. Forget gate:

$f_3 = \sigma(1(0.82) + 1)$

$= \sigma(1.82)$

$= 0.86$

2. Input gate:

$I_3 = \sigma(1.82) = 0.86$

$\bar{C}_3 = \tanh(1.82) = 0.94$

3. Update cell state:

$$C_3 = 0.86 \times 1.54 + 0.86 \times 0.94$$
$$= 2.14$$

4. Output gate:

$$O_3 = \sigma(1.82) = 0.86$$
$$h_3 = 0.86 \times \tanh(2.14)$$
$$= 0.83$$

Predict:

$t = 4$

1. Forget gate

$$f_4 = \sigma(1(0.83 + 0.86) + 1)$$
$$= 0.84$$

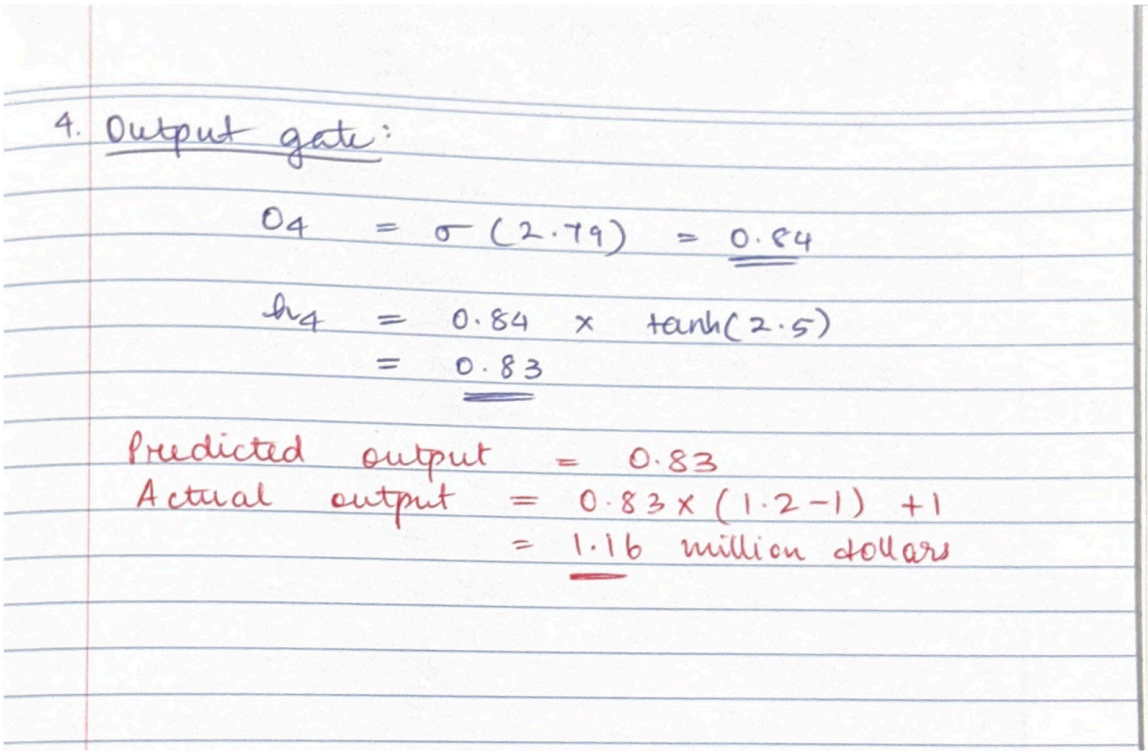
2. Input gate

$$i_4 = \sigma(2.79) = 0.84$$

$$\bar{C}_4 = \tanh(1.69) = 0.93$$

3. Update cell state:

$$C_4 = 0.84 \times 2.14 + 0.84 \times 0.93$$
$$= 2.5$$



Prediction of House Price in 2025 Using LSTM

Now as our model looks ready with the weights and biases, we can try running it past a new data. Using the above formulas, the prediction for housing prices in Mawson Lakes for 2025 has been done above.

Plugging the weights and biases for the prediction iteration and using the output in the last training iteration for the prediction iteration helped us in finding the **predicted house price of 1.16 million dollars in Mawson Lakes**.

Practicals

STEP 1: Prepare the environment.

In [1]:
```
# Import the required libraries:
import io
import os
import re
import sys
import time
import tqdm
import spacy
import shutil
import string
import tensorflow as tf
import numpy as np
from gensim.models import Word2Vec
import pandas as pd
from nltk.tokenize import word_tokenize
from nltk.corpus import stopwords
import matplotlib.pyplot as plt

from tensorflow.keras import backend as K
```

```
from tensorflow.keras import Sequential, layers, models
from tensorflow.keras.layers import Dense, GlobalAveragePooling1D, TextVecto
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow.keras.models import load_model

from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, precision_score, recall_score,
```

Step 2: Setting a Seed for Reproducibility.

In [2]:
```
SEED = 42  # Set a seed value for reproducibility in random operations
AUTOTUNE = tf.data.AUTOTUNE  # Automatically determine the optimal number o
```

Step 3: Download and Prepare the Toxic Dataset

In [3]:
```
from google.colab import files

uploaded = files.upload()
```

Choose files No file chosen Upload widget is only available when the cell has been executed in the current browser session. Please rerun this cell to enable.

```
Saving Toxic.csv to Toxic.csv
```

Step 4: Loading the data

In [4]:
```
# Load the CSV data
data = pd.read_csv('Toxic.csv')

data.head()
```

Out[4]:

	id	comment_text	toxic	severe_toxic	obscene	threat	insult	identity
0	0000997932d777bf	Explanation\nWhy the edits made under my usern...	0	0	0	0	0	
1	000103f0d9cfb60f	D'aww! He matches this background colour I'm s...	0	0	0	0	0	
2	000113f07ec002fd	Hey man, I'm really not trying to edit war. It...	0	0	0	0	0	
3	0001b41b1c6bb37e	"\nMore\nI can't make any real suggestions on ...	0	0	0	0	0	
4	0001d958c54c6e35	You, sir, are my hero. Any chance you remember...	0	0	0	0	0	

The **data** has eight columns : `id, comment_text, toxic, severe_toxic, obscene, threat, insult and identity_hate` . These columns take in **binary values 0 and 1**. If the comment belongs to any of these categories, the corresponding column has a binary value 1, else 0.

Word2Vec

Step 5: Text Preprocessing for NLP

The function converts the **entire comment to lowercase** to ensure uniformity. This is crucial because in NLP, "Word" and "word" might be considered different tokens unless normalized. Using the Spacy library, a list of default stopwords is extracted. These are common words (e.g., "the", "is", "and") that do not carry significant meaning in the context of text analysis. The function **filters out these stopwords** from the input comment. If the input contains newline characters (\n), they are **explicitly removed** from the comment. Punctuation is removed from the text using Python's string.punctuation and str.translate() method. This step **eliminates unnecessary symbols** such as periods, commas, and exclamation points that could interfere with text analysis. The final output is a **cleaned and normalized version** of the original comment, which can then be further processed for NLP tasks like tokenization or word embeddings.

In [5]:
```
nlp = spacy.load('en_core_web_sm', disable=['parser', 'tagger', 'ner'])
stopWords = list(nlp.Defaults.stop_words)
puncs = string.punctuation

def processComment(comment: str):
    comment = comment.lower()
    comment = comment.split()
    comment = [word for word in comment if word not in stopWords]
    if "\n" in comment:
        comment.remove("\n")
    comment = ' '.join(comment)
    comment = comment.translate(str.maketrans('', '', puncs))
    return comment
```

Step 6: Preprocessing Toxic Comment Data

In this step, the raw data is being transformed into a format suitable for classification. The `processComment` function **cleans the text** by removing stopwords, punctuation, and converting the text to lowercase. Then, the columns in the original dataset representing different types of toxic behaviors (such as "severe toxic," "obscene," etc.) are **aggregated** into a binary "toxic" label, where a comment is labeled as toxic (`1`) if any of the toxic behavior columns have a positive value, otherwise it's labeled as non-toxic (`0`). The final structured dataset is stored as a DataFrame named `toxicData`.

In [6]:
```
toxicData = {
    "id" : [],
    "commentText" : [],
    "toxic" : []
}
for index, row in data.iterrows():
    toxicData["id"].append(row["id"])
    toxicData["commentText"].append(processComment(row["comment_text"]))
    toxic = row["toxic"] + row["severe_toxic"] + row["obscene"] + row["threa
    toxicData["toxic"].append(1 if toxic > 0 else 0)

toxicData = pd.DataFrame(toxicData)
toxicData
```

Out[6]:

	id	commentText	toxic
0	0000997932d777bf	explanation edits username hardcore metallica ...	0
1	000103f0d9cfb60f	daww matches background colour im seemingly st...	0
2	000113f07ec002fd	hey man im trying edit war its guy constantly ...	0
3	0001b41b1c6bb37e	cant real suggestions improvement wondered s...	0
4	0001d958c54c6e35	you sir hero chance remember page thats on	0
...	...	...	...
159566	ffe987279560d7ff	and second time asking view completely contrad...	0
159567	ffea4adeee384e90	ashamed horrible thing talk page 128611993	0
159568	ffee36eab5c267c9	spitzer umm theres actual article prostitution...	0
159569	fff125370e4aaaf3	looks like actually speedy version deleted loo...	0
159570	fff46fc426af1f9a	dont think understand came idea bad right aw...	0

159571 rows × 3 columns

Step 7: Preparing Data for Model Training with Batching

In this step, the dataset `toxicData` is being prepared for model training by creating batches and optimizing data loading. The from_tensor_slices method converts the columns of comment texts and corresponding toxicity labels into a TensorFlow dataset format. The dataset is then divided into smaller groups, or "batches," of 32 samples each, using the specified `batch_size` of 32. This helps in efficient training by allowing the model to process multiple samples at once, reducing memory usage. Additionally, the `cache()` function is used to store the dataset in memory for faster access during training, and `prefetch(buffer_size=AUTOTUNE)` is employed to optimize data loading by overlapping data preprocessing and model execution, further improving training speed.

In [7]:
```
batch_size = 32

fullData = tf.data.Dataset.from_tensor_slices(
    (toxicData["commentText"].values, toxicData["toxic"].values)
).batch(batch_size)
fullData = fullData.cache().prefetch(buffer_size=AUTOTUNE)
```

Step 8: Text Vectorization and Custom Standardization for Model Input

This section focuses on preparing textual data for model input through vectorization and standardization. The function `custom_standardization` first converts the input text to lowercase, removes any HTML tags (`<br />`), and eliminates punctuation marks using regular expressions. This helps in cleaning the raw text data for processing. The `TextVectorization` layer is then defined with the custom standardization function, a vocabulary size of 10,000 words, and an output sequence length set to the batch size (32). This layer transforms text into integer sequences, where each word is replaced by its corresponding integer index. The `text_ds` dataset, created by mapping only the textual data (`x`) from the `fullData` , is then passed to the `adapt()` method of the

`vectorize_layer`. This trains the vectorizer to build a vocabulary from the dataset, ensuring that the model receives preprocessed, numerical input.

In [8]:
```
def custom_standardization(input_data):
    lowercase = tf.strings.lower(input_data)
    stripped_html = tf.strings.regex_replace(lowercase, '<br />', ' ')
    return tf.strings.regex_replace(stripped_html, '[%s]' % re.escape(string

vocab_size = 10000

sequence_length = batch_size

vectorize_layer = TextVectorization(
    standardize=custom_standardization,
    max_tokens=vocab_size,
    output_mode='int',
    output_sequence_length=sequence_length)


text_ds = fullData.map(lambda x, y: x)

vectorize_layer.adapt(text_ds)
```

Step 9: Text Preprocessing for Sentiment Analysis

In this code snippet, we define a function `preprocess_text` that processes input text for sentiment analysis. The function first converts the text to lowercase, ensuring uniformity, and replaces HTML line breaks with spaces for better readability. It then removes punctuation using regular expressions, which helps to clean the text for subsequent analysis. The processed text is returned as a list of words. The main loop iterates through batches of text data (`fullData`), applying the `preprocess_text` function to each text entry and appending the resulting list of words to the sentences list, creating a structured format suitable for further analysis or model training.

In [9]:
```
def preprocess_text(text):
    text = tf.strings.lower(text)
    text = tf.strings.regex_replace(text, '<br />', ' ')
    text = tf.strings.regex_replace(text, '[%s]' % re.escape(string.punctuat
    return text.numpy().decode('utf-8').split()

sentences = []

for text_batch, _ in fullData:
    for text in text_batch:
        sentences.append(preprocess_text(text))
```

Step 10: Training a Word2Vec Model

In this code snippet, we are training a Word2Vec model using the `sentences` prepared from the text preprocessing step. The `Word2Vec` constructor specifies several parameters to configure the model: `vector_size` is set to 100, meaning each word will be represented by a 100-dimensional vector; `window` is set to 5, which defines the maximum distance between the current and predicted word within a sentence; `min_count` is set to 5, ensuring that only words appearing at least five times in the dataset are considered, which helps filter out rare words; and `workers` is set to 4, enabling parallel processing to speed up training. This setup allows the model to learn

meaningful word representations from the given text data, capturing semantic relationships between words.

In [10]:
```
word2vec_model = Word2Vec(
    sentences = sentences,
    vector_size=100,
    window=5,
    min_count=5,
    workers=4
)
```

Step 11: Creating an Embedding Layer with Pretrained Word Vectors

In this code snippet, we extract the word vectors from the trained Word2Vec model and use them to create an embedding layer in TensorFlow. The variable `weights` holds the word vectors, where `weights.shape[0]` represents the number of unique words (the input dimension) and `weights.shape[1]` denotes the dimensionality of the word vectors (the output dimension). We initialize the embedding layer using these pretrained weights through the `embeddings_initializer` parameter, specifically using a constant initializer to maintain the learned representations. Setting `trainable=True` allows the embedding weights to be updated during model training, enabling the model to fine-tune the embeddings for better performance on the specific task at hand. This embedding layer can then be integrated into a neural network for tasks like text classification or sentiment analysis.

In [11]:
```
weights = word2vec_model.wv.vectors

embedding_layer = tf.keras.layers.Embedding(
    input_dim=weights.shape[0],
    output_dim=weights.shape[1],
    embeddings_initializer=tf.keras.initializers.Constant(weights),
    trainable=True
)
```

Splitting dataset

Step 12: Split into training and testing data

In [12]:
```
train, test = train_test_split(toxicData, test_size=0.2, random_state=SEED)

trainDataset = tf.data.Dataset.from_tensor_slices(
    (train["commentText"].values, train["toxic"].values)
).batch(batch_size)
testDataset = tf.data.Dataset.from_tensor_slices(
    (test["commentText"].values, test["toxic"].values),
).batch(batch_size)
```

Step 13: Optimizing Dataset Loading with Caching and Prefetching

In [13]:
```
trainDataset = trainDataset.cache().prefetch(buffer_size=AUTOTUNE)
testDataset = testDataset.cache().prefetch(buffer_size=AUTOTUNE)
```

FNN

Step 1: Defining a Feedforward Neural Network Model

Requirements: 2 hidden layers with 64 neurons each, ReLU activation for hidden layers, 1 output neuron with sigmoid activation, Adam optimizer, learning rate of 0.001, 5 epochs, 32 batch size.

In this code snippet, we define a function `fnnModel()` that constructs a feedforward neural network (FNN) for binary classification tasks, such as sentiment analysis. The model uses a sequential architecture, starting with a `vectorize_layer` to convert input text into numerical format. Next, it incorporates an `embedding_layer` to represent words as dense vectors, followed by a `GlobalAveragePooling1D` layer that averages the embeddings across the sequence length, effectively reducing the dimensionality while retaining important features. Two dense layers with 64 units and ReLU activation are added to capture complex patterns in the data. The final output layer uses a sigmoid activation function to produce a probability score for the binary classification.

The model is compiled with the Adam optimizer and a learning rate of 0.001, using binary crossentropy as the loss function to measure performance during training.

In [ ]:
```
def fnnModel():
    model = models.Sequential([
        vectorize_layer,
        embedding_layer,
        GlobalAveragePooling1D(),
        Dense(64, activation='relu'),
        Dense(64, activation='relu'),
        Dense(1, activation='sigmoid')
    ])

    model.compile(
        optimizer = tf.keras.optimizers.Adam(learning_rate=0.001),
        loss = tf.keras.losses.BinaryCrossentropy(),
        metrics = [
            "accuracy",
            "precision",
            "recall"
        ]
    )
    return model
```

Step 2: Training the Feedforward Neural Network Model

In this code snippet, we create an instance of the FNN model defined by the `fnnModel()` function and initiate the training process using the `fit()` method. The model is trained on the `trainDataset` and evaluated on the `testDataset` for a specified number of epochs (5 in this case). The `batch_size` parameter determines how many samples are processed before the model's internal parameters are updated, helping to manage memory usage and training efficiency.

The training progress, including metrics such as loss and accuracy, is logged using a TensorBoard callback, which saves logs in the specified `log_dir` ("logs"). This allows for real-time visualization of training metrics and model performance, helping to

diagnose issues and assess how well the model is learning over time. This structured training approach facilitates effective model optimization and performance tracking.

In [ ]:
```
model = fnnModel()
history = model.fit(
    trainDataset,
    validation_data=testDataset,
    epochs=5,
    batch_size=batch_size,
    callbacks=[tf.keras.callbacks.TensorBoard(log_dir="logs")]
)
```

```
Epoch 1/5
3990/3990 ━━━━━━━━━━━━━━━━━━━━ 197s 49ms/step - accuracy: 0.9174 - loss: 0.
2457 - precision: 0.7493 - recall: 0.2459 - val_accuracy: 0.9357 - val_los
s: 0.1778 - val_precision: 0.6585 - val_recall: 0.7633
Epoch 2/5
3990/3990 ━━━━━━━━━━━━━━━━━━━━ 178s 45ms/step - accuracy: 0.9556 - loss: 0.
1260 - precision: 0.8700 - recall: 0.6639 - val_accuracy: 0.9460 - val_los
s: 0.1522 - val_precision: 0.7196 - val_recall: 0.7673
Epoch 3/5
3990/3990 ━━━━━━━━━━━━━━━━━━━━ 202s 45ms/step - accuracy: 0.9606 - loss: 0.
1098 - precision: 0.8716 - recall: 0.7195 - val_accuracy: 0.9495 - val_los
s: 0.1456 - val_precision: 0.7477 - val_recall: 0.7592
Epoch 4/5
3990/3990 ━━━━━━━━━━━━━━━━━━━━ 202s 44ms/step - accuracy: 0.9632 - loss: 0.
1014 - precision: 0.8726 - recall: 0.7491 - val_accuracy: 0.9540 - val_los
s: 0.1417 - val_precision: 0.7913 - val_recall: 0.7435
Epoch 5/5
3990/3990 ━━━━━━━━━━━━━━━━━━━━ 177s 44ms/step - accuracy: 0.9649 - loss: 0.
0947 - precision: 0.8726 - recall: 0.7684 - val_accuracy: 0.9559 - val_los
s: 0.1440 - val_precision: 0.8156 - val_recall: 0.7321
```

Step 3: Evaluation on test data and Metrics

In [ ]:
```
testLossFnn, testAccuracyFnn, testPrecisionFnn, testRecallFnn = model.evalua
testF1ScoreFnn = 2 * (testPrecisionFnn * testRecallFnn) / (testPrecisionFnn

print(f"Test Loss: {testLossFnn}")
print(f"Test Accuracy: {testAccuracyFnn}")
print(f"Test Precision: {testPrecisionFnn}")
print(f"Test Recall: {testRecallFnn}")
print(f"Test F1 Score: {testF1ScoreFnn}")
```

```
998/998 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.9568 - loss: 0.1436
- precision: 0.8175 - recall: 0.7348
Test Loss: 0.1440490484237671
Test Accuracy: 0.9559454917907715
Test Precision: 0.8155906796455383
Test Recall: 0.7321208119392395
Test F1 Score: 0.7716049326231422
```

1. **Test Loss (0.1440):** This value represents the error between the model's predictions and the true labels on the test data. A lower loss indicates better model performance. In this case, a loss of 0.1440 suggests the model has learned to predict fairly well with minimal error.

2. **Test Accuracy (0.9559):** This indicates that the model correctly classified 95.59% of the test samples. Accuracy is a general measure of performance, showing the proportion of correctly classified instances out of the total.

3. **Test Precision (0.8156):** Precision reflects how many of the instances predicted as positive (toxic, in this case) are actually positive. With a precision of 81.56%, this means that of all comments the model classified as toxic, 81.56% were correct.

4. **Test Recall (0.7321):** Recall (also known as sensitivity) measures how many of the actual positive instances the model correctly identified. With a recall of 73.21%, the model correctly detected 73.21% of all true toxic comments.

5. **Test F1 Score (0.7716):** The F1 score is the harmonic mean of precision and recall, providing a balanced measure of both. With a score of 77.16%, it indicates a good trade-off between precision and recall, ensuring the model is not overly biased toward either metric.

Logistic Regression (LR)

STEP 1: Extracting Labels from Datasets

In this code snippet, we extract the labels from the `trainDataset` and `testDataset` to create the numpy arrays `y_train` and `y_test`. Using a list comprehension, we iterate over the training and testing datasets, accessing each batch of labels. The `label_batch.numpy()` method converts the TensorFlow tensor to a NumPy array, allowing us to work with standard numerical operations.

The `np.concatenate()` function combines these arrays along the specified axis (0, meaning vertically) to form a single array of labels for the entire training and testing datasets.

In [ ]:
```
yTrain = np.concatenate([label_batch.numpy() for _, label_batch in trainData
yTest = np.concatenate([label_batch.numpy() for _, label_batch in testDatase
```

STEP 2: Defining a Logistic Regression-like Neural Network Model

Requirements: 2 hidden layers with 64 neurons each, ReLU activation for hidden layers, 1 output neuron with sigmoid activation, Adam optimizer, learning rate of 0.001, 5 epochs, 32 batch size.

In this code snippet, the `LR_MODEL()` function defines a logistic regression-inspired neural network for binary classification. The architecture uses a sequential model that starts with a `vectorize_layer` to convert text input into numerical format, followed by an `embedding_layer` to map words to dense vectors. A `GlobalAveragePooling1D` layer reduces the dimensionality of the embeddings by averaging over the sequence length, followed by two fully connected (`Dense`) layers with ReLU activation to capture patterns in the data. A `Dropout` layer with a 50% dropout rate is introduced to prevent overfitting by randomly ignoring half the neurons during training. The final output layer uses a sigmoid activation function to output probabilities for binary classification. The model is compiled with the Adam optimizer, binary crossentropy loss (with logits), and metrics for accuracy, precision, and recall, ensuring effective performance evaluation.

In [ ]:
```
def LR_MODEL():
    model = models.Sequential([
        vectorize_layer,
        embedding_layer,
        GlobalAveragePooling1D(),
        Dense(64, activation='relu'),
        Dense(64, activation='relu'),
        Dropout(0.5),
        Dense(1, activation='sigmoid')
    ])
    model.compile(
        optimizer = tf.keras.optimizers.Adam(learning_rate=0.001),
        loss = tf.keras.losses.BinaryCrossentropy(from_logits=True),
        metrics = [
            "accuracy",
            "precision",
            "recall"
        ]
    )
    return model
```

STEP 3: Training and Validating the Neural Network with Dropout

This code trains the logistic regression-inspired neural network defined in `LR_MODEL()` on the `trainDataset` and evaluates its performance on the `testDataset` for 5 epochs. The model processes data in batches defined by `batch_size`, and the `TensorBoard` callback logs training metrics, allowing real-time performance visualization. The architecture includes dropout to prevent overfitting, making the model more robust. Throughout the training process, the model adjusts its weights based on binary crossentropy loss while monitoring accuracy, precision, and recall. This setup ensures that the model generalizes well and is evaluated continuously against the test data.

In [ ]:
```
model = LR_MODEL()
history = model.fit(
    trainDataset,
    validation_data=testDataset,
    epochs=5,
    batch_size=batch_size,
    callbacks=[tf.keras.callbacks.TensorBoard(log_dir="logs")]
)
```

```
Epoch 1/5
/usr/local/lib/python3.10/dist-packages/keras/src/backend/tensorflow/nn.py:
681: UserWarning: "`binary_crossentropy` received `from_logits=True`, but t
he `output` argument was produced by a Sigmoid activation and thus does not
represent logits. Was this intended?
  output, from_logits = _get_logits(
```

```
3990/3990 ━━━━━━━━━━━━━━━━━━━━ 215s 52ms/step - accuracy: 0.9578 - loss: 0.
1191 - precision: 0.8718 - recall: 0.6836 - val_accuracy: 0.9516 - val_los
s: 0.1830 - val_precision: 0.7728 - val_recall: 0.7423
Epoch 2/5
3990/3990 ━━━━━━━━━━━━━━━━━━━━ 227s 57ms/step - accuracy: 0.9712 - loss: 0.
0745 - precision: 0.9000 - recall: 0.8071 - val_accuracy: 0.9518 - val_los
s: 0.2343 - val_precision: 0.7889 - val_recall: 0.7176
Epoch 3/5
3990/3990 ━━━━━━━━━━━━━━━━━━━━ 263s 66ms/step - accuracy: 0.9745 - loss: 0.
0653 - precision: 0.9196 - recall: 0.8218 - val_accuracy: 0.9520 - val_los
s: 0.2636 - val_precision: 0.7932 - val_recall: 0.7139
Epoch 4/5
3990/3990 ━━━━━━━━━━━━━━━━━━━━ 259s 65ms/step - accuracy: 0.9766 - loss: 0.
0605 - precision: 0.9296 - recall: 0.8342 - val_accuracy: 0.9510 - val_los
s: 0.2788 - val_precision: 0.7877 - val_recall: 0.7093
Epoch 5/5
3990/3990 ━━━━━━━━━━━━━━━━━━━━ 239s 59ms/step - accuracy: 0.9796 - loss: 0.
0545 - precision: 0.9410 - recall: 0.8538 - val_accuracy: 0.9489 - val_los
s: 0.3118 - val_precision: 0.7717 - val_recall: 0.7065
```

STEP 4: Extracting and Reshaping Features from the Model

In this code, the `predict()` method is used to generate output features from the trained model for both the `trainDataset` and `testDataset`. The predictions, stored in `featuresTrain` and `featuresTest`, represent the final layer's output (probabilities or logits, depending on the model setup). These feature vectors are then reshaped using `.reshape(featuresTrain.shape[0], -1)` and `.reshape(featuresTest.shape[0], -1)` to ensure they are in a 2D format, where each row corresponds to a sample, and columns represent the features. Reshaping is necessary because the model's output could be multi-dimensional, and this transformation flattens it into a format suitable for further analysis, such as feeding into another classifier or performing feature-based evaluations.

In [ ]:
```
featuresTrain = model.predict(trainDataset)
featuresTest = model.predict(testDataset)

featuresTrain = featuresTrain.reshape(featuresTrain.shape[0], -1)
featuresTest = featuresTest.reshape(featuresTest.shape[0], -1)
```

```
3990/3990 ━━━━━━━━━━━━━━━━━━━━ 12s 3ms/step
998/998 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step
```

STEP 5: Train and Evaluate the Logistic Regression Model

In this code snippet, a logistic regression model (`log_reg`) is trained using features extracted from a neural network. The model is trained on the reshaped `featuresTrain` and corresponding labels `yTrain`, with training time measured using `time.time()`. After fitting the model, predictions are made on `featuresTest`, and various performance metrics are calculated, including accuracy, precision, recall, F1-score, and ROC-AUC, which assess the model's ability to correctly classify the test data. Additionally, an approximate test loss is computed using the negative log of the predicted probabilities. Finally, the results and the time taken for training are printed. This approach demonstrates how logistic regression can be used as a second-stage classifier on neural network-derived features to improve or analyze classification performance.

In [ ]:
```
log_reg = LogisticRegression(max_iter=1000)

start_time = time.time()
log_reg.fit(featuresTrain, yTrain)

end_time = time.time()

log_reg_predictions = log_reg.predict(featuresTest)

lr_accuracy = accuracy_score(yTest, log_reg_predictions)
lr_precision = precision_score(yTest, log_reg_predictions)
lr_recall = recall_score(yTest, log_reg_predictions)
lr_f1 = f1_score(yTest, log_reg_predictions)
lr_roc_auc = roc_auc_score(yTest, log_reg_predictions)

lr_test_loss = -np.mean(np.log(log_reg.predict_proba(featuresTest)[np.arange

# Print the performance metrics:
print(f"Logistic Regression Results:")
print(f"Accuracy: {lr_accuracy:.4f}")
print(f"Precision: {lr_precision:.4f}")
print(f"Recall: {lr_recall:.4f}")
print(f"F1-Score: {lr_f1:.4f}")
print(f"ROC-AUC: {lr_roc_auc:.4f}")
print(f"Test Loss (approx.): {lr_test_loss:.4f}")
print(f"Training Time: {end_time - start_time:.2f} seconds")
```

```
Logistic Regression Results:
Accuracy: 0.9495
Precision: 0.7832
Recall: 0.6961
F1-Score: 0.7371
ROC-AUC: 0.8371
Test Loss (approx.): 0.1870
Training Time: 0.11 seconds
```

1. **Accuracy (0.9495):** The model correctly classified 94.95% of the test samples, indicating strong overall performance.
2. **Precision (0.7832):** Of all the comments predicted as toxic, 78.32% were actually toxic. This shows how well the model avoids false positives.
3. **Recall (0.6961):** The model identified 69.61% of the actual toxic comments, meaning it has some difficulty capturing all true positives (slightly lower sensitivity).
4. **F1-Score (0.7371):** The F1-score balances precision and recall, indicating the trade-off between avoiding false positives and false negatives, with a reasonable balance at 73.71%.
5. **ROC-AUC (0.8371):** This measures the model's ability to distinguish between toxic and non-toxic comments. A score of 0.8371 suggests good discriminatory power, but not perfect.
6. **Test Loss (approx. 0.1870):** This represents an estimate of how well the model's predicted probabilities match the true labels, with lower values indicating better model calibration.

7. **Training Time (0.11 seconds):** The logistic regression model trained very quickly, which highlights the efficiency of using logistic regression on pre-extracted features.

RNN

STEP 1: Text Vectorization using Keras TextVectorization Layer

This code defines a `TextVectorization` layer (`encoder`) from TensorFlow/Keras, which transforms raw text data into a format suitable for machine learning models by converting text into numerical tokens. The `max_tokens` parameter sets the maximum vocabulary size, limiting how many unique words will be represented. The `adapt()` method is then called on the `fullData`, mapping only the text (ignoring labels), to learn the vocabulary based on the dataset. This process prepares the text for input into neural network models by creating a consistent and standardized numerical representation of the words.

In [14]:
```
encoder = tf.keras.layers.TextVectorization(
    max_tokens=vocab_size)
encoder.adapt(fullData.map(lambda text, label: text))
```

STEP 2: Extracting and Viewing the Vocabulary

In this code, the vocabulary learned by the `TextVectorization` layer is retrieved using the `get_vocabulary()` method and stored as a NumPy array (`vocab`). The first 20 words from this vocabulary are then displayed using `vocab[:20]`. This allows us to inspect the most frequent words in the dataset, as the vocabulary is typically ordered by frequency. Viewing the top words provides insight into the most common terms encountered during training and helps in understanding the structure of the text data that the model will use.

In [15]:
```
vocab = np.array(encoder.get_vocabulary())
vocab[:20]
```

Out[15]:
```
array(['', '[UNK]', 'article', 'page', 'wikipedia', 'talk', 'like',
       'dont', 'think', 'it', 'im', 'know', 'people', 'its', 'edit',
       'articles', 'you', 'use', 'time', 'thanks'], dtype='<U39')
```

STEP 3: Defining a Bidirectional LSTM RNN Model

Requirements: 2 hidden layers with 64 neurons each, ReLU activation for hidden layers, 1 output neuron with sigmoid activation, Adam optimizer, learning rate of 0.001, 5 epochs, 32 batch size.

The `RNN_MODEL()` function defines a recurrent neural network (RNN) model using a bidirectional LSTM (Long Short-Term Memory) layer for text classification tasks. The model begins with the `encoder` layer to convert raw text into integer tokens, followed by an embedding layer that maps these tokens to dense 64-dimensional vectors. The `mask_zero=True` parameter ensures that padding tokens are ignored during training.

A `Bidirectional` LSTM layer is used to process the sequence in both forward and backward directions, allowing the model to capture context from both past and future words. Two fully connected (`Dense`) layers with 64 units and ReLU activation are included to learn deeper representations of the features. The final output layer uses a sigmoid activation to generate a probability for binary classification. The model is compiled with the Adam optimizer, binary crossentropy loss, and evaluation metrics for accuracy, precision, and recall.

In [ ]:
```
def RNN_MODEL():
    model = tf.keras.Sequential([
              encoder,
              tf.keras.layers.Embedding(
                  input_dim=len(encoder.get_vocabulary()),
                  output_dim=64,
                  mask_zero=True
              ),
              tf.keras.layers.Masking(),
              tf.keras.layers.Bidirectional(tf.keras.layers.LSTM(64, use_cu
              tf.keras.layers.Dense(64, activation='relu'),
              tf.keras.layers.Dense(64, activation='relu'),
              tf.keras.layers.Dense(1, activation = 'sigmoid')
          ])

    model.compile(
              optimizer = tf.keras.optimizers.Adam(learning_rate=0.001),
              loss = tf.keras.losses.BinaryCrossentropy(from_logits=False),
              metrics = [
                  "accuracy",
                  "precision",
                  "recall"
              ]
          )
    return model
```

STEP 4: Checking Masking Support in RNN Model Layers

In this code snippet, the `RNN_MODEL()` function is called to create an instance of the recurrent neural network model, and then a list comprehension is used to check which layers in the model support masking. The `supports_masking` attribute indicates whether a layer can handle masked inputs, which is important for ignoring padding tokens during training and evaluation. The output will be a list of Boolean values, where each element corresponds to a layer in the model, showing `True` for layers that can appropriately process masked inputs (like the embedding and LSTM layers) and `False` for those that cannot. This check ensures that the model is correctly set up to handle variable-length sequences with padding, allowing for effective training on text data.

In [ ]:
```
model = RNN_MODEL()
print([layer.supports_masking for layer in model.layers])
```

```
[False, True, True, True, True, True, True]
```

STEP 5: Training the Bidirectional LSTM RNN Model

In this code snippet, the `RNN_MODEL()` function is invoked to create an instance of the bidirectional LSTM model, which is then trained using the `fit()` method. The model is trained on the `trainDataset` , with validation performed on the `testDataset` over

5 epochs. The `validation_steps` parameter is set to 30, which specifies the number of steps (batches) to use for validating the model after each epoch. During training, the model updates its weights based on the training data while simultaneously evaluating its performance on the validation data. This process allows for monitoring metrics like accuracy, precision, and recall, helping to assess how well the model is learning and generalizing to unseen data. The use of a bidirectional LSTM helps capture contextual information from both directions of the input sequence, making it well-suited for text classification tasks.

In [ ]:
```
model = RNN_MODEL()
history = model.fit(
        trainDataset,
        validation_data=testDataset,
        epochs=5,
        validation_steps=30,
    )
```

```
Epoch 1/5
3990/3990 ━━━━━━━━━━━━━━━━━━━━ 1454s 362ms/step - accuracy: 0.9412 - loss:
0.1780 - precision: 0.8109 - recall: 0.5211 - val_accuracy: 0.9667 - val_lo
ss: 0.0974 - val_precision: 0.9333 - val_recall: 0.7216
Epoch 2/5
3990/3990 ━━━━━━━━━━━━━━━━━━━━ 1361s 341ms/step - accuracy: 0.9626 - loss:
0.1007 - precision: 0.8754 - recall: 0.7386 - val_accuracy: 0.9563 - val_lo
ss: 0.1070 - val_precision: 0.7889 - val_recall: 0.7553
Epoch 3/5
3990/3990 ━━━━━━━━━━━━━━━━━━━━ 1349s 338ms/step - accuracy: 0.9683 - loss:
0.0826 - precision: 0.8864 - recall: 0.7909 - val_accuracy: 0.9542 - val_lo
ss: 0.1429 - val_precision: 0.8173 - val_recall: 0.7727
Epoch 4/5
3990/3990 ━━━━━━━━━━━━━━━━━━━━ 1353s 339ms/step - accuracy: 0.9746 - loss:
0.0642 - precision: 0.8875 - recall: 0.8603 - val_accuracy: 0.9500 - val_lo
ss: 0.1473 - val_precision: 0.6735 - val_recall: 0.8049
Epoch 5/5
3990/3990 ━━━━━━━━━━━━━━━━━━━━ 1360s 341ms/step - accuracy: 0.9808 - loss:
0.0511 - precision: 0.9126 - recall: 0.8984 - val_accuracy: 0.9521 - val_lo
ss: 0.2180 - val_precision: 0.7379 - val_recall: 0.8000
```

STEP 6: Evaluating the Bidirectional LSTM RNN Model

In this code snippet, the trained bidirectional LSTM model is evaluated on the testDataset using the evaluate() method, which returns the test loss, accuracy, precision, and recall. These metrics provide insights into the model's performance on unseen data.

In [ ]:
```
testLossRnn, testAccuracyRnn, testPrecisionRnn, testRecallRnn = model.evalua
testF1ScoreRnn = 2 * (testPrecisionRnn * testRecallRnn) / (testPrecisionRnn

print(f"Test Loss: {testLossRnn}")
print(f"Test Accuracy: {testAccuracyRnn}")
print(f"Test Precision: {testPrecisionRnn}")
print(f"Test Recall: {testRecallRnn}")
print(f"Test F1 Score: {testF1ScoreRnn}")
```

```
998/998 ━━━━━━━━━━━━━━━━━━━━ 80s 80ms/step - accuracy: 0.9490 - loss: 0.197
1 - precision: 0.7429 - recall: 0.7541
Test Loss: 0.19891712069511414
Test Accuracy: 0.9499294757843018
Test Precision: 0.752764105796814
Test Recall: 0.7555487155914307
Test F1 Score: 0.7541538402552855
```

1. **Test Loss (0.1989)**: This value indicates the average error in the model's predictions compared to the actual labels. A loss of 0.1989 suggests that while the model performs reasonably well, there is still room for improvement in terms of prediction accuracy.

2. **Test Accuracy (0.9499)**: The model achieved an accuracy of 94.99%, meaning it correctly classified nearly 95% of the test samples. This high accuracy indicates that the model is effective in distinguishing between toxic and non-toxic comments.

3. **Test Precision (0.7527)**: Precision at 75.27% indicates that when the model predicts a comment as toxic, it is correct 75.92% of the time. This reflects the model's performance in minimizing false positives, which is particularly important in contexts where misclassifying a non-toxic comment as toxic could lead to negative consequences.

4. **Test Recall (0.7555)**: With a recall of 75.55%, the model successfully identifies about 75.55% of the actual toxic comments in the dataset. This shows that while the model is good at predicting toxic comments, there is a notable proportion of true positives that it fails to capture.

5. **Test F1 Score (0.7541)**: The F1 score, at 75.41%, provides a balance between precision and recall. It suggests that while the model performs well, there are trade-offs between accurately predicting toxic comments and capturing all instances of them.

LSTM

STEP 1: Defining a Stacked Bidirectional LSTM Model

Requirements: 2 hidden layers with 64 neurons each, ReLU activation for hidden layers, 1 output neuron with sigmoid activation, Adam optimizer, learning rate of 0.001, 5 epochs, 32 batch size.

The `LSTM_MODEL()` function constructs a sequential model that incorporates two layers of bidirectional LSTMs, enhancing the model's capacity to capture temporal dependencies in the input text. Initially, the model applies the `encoder` to transform raw text into integer tokens, followed by an embedding layer that maps these tokens to dense vectors with a dimension of 64. The first bidirectional LSTM layer is configured to return sequences, allowing the subsequent LSTM layer to process the full sequence of outputs from the first layer. The second bidirectional LSTM layer reduces the output to a single sequence representation. After the LSTM layers, two fully connected (`Dense`) layers with ReLU activation are included to learn complex patterns, followed by a final

output layer that employs a sigmoid activation function for binary classification. The model is compiled with the Adam optimizer and binary crossentropy loss, along with metrics for accuracy, precision, and recall.

In [16]:
```
def LSTM_MODEL():
    model = tf.keras.Sequential([
              encoder,
              tf.keras.layers.Embedding(
                  input_dim=len(encoder.get_vocabulary()),
                  output_dim=64,
                  mask_zero=True
              ),
              tf.keras.layers.Masking(),
              tf.keras.layers.Bidirectional(tf.keras.layers.LSTM(64, return_
              tf.keras.layers.Bidirectional(tf.keras.layers.LSTM(64)),
              tf.keras.layers.Dense(64, activation='relu'),
              tf.keras.layers.Dense(64, activation='relu'),
              tf.keras.layers.Dense(1, activation = 'sigmoid')
            ])

    model.compile(
              optimizer = tf.keras.optimizers.Adam(learning_rate=0.001),
              loss = tf.keras.losses.BinaryCrossentropy(from_logits=False),
              metrics = [
                  "accuracy",
                  "precision",
                  "recall"
              ]
          )
    return model
```

STEP 2: Training the Stacked Bidirectional LSTM Model

In this code snippet, the `LSTM_MODEL()` function is called to create an instance of the stacked bidirectional LSTM model, which is then trained using the `fit()` method. The model is trained on the `trainDataset`, with validation conducted on the `testDataset` over a specified number of epochs (5 in this case). The `validation_steps` parameter is set to 30, indicating that the validation will evaluate the model on 30 batches of the validation dataset after each epoch. During the training process, the model learns to minimize the binary crossentropy loss while optimizing for accuracy, precision, and recall. This approach allows for continuous monitoring of the model's performance on both the training and validation datasets, helping to identify overfitting or underfitting and ensuring that the model generalizes well to unseen data. The use of stacked bidirectional LSTMs enables the model to capture complex patterns and relationships in the input text more effectively, making it well-suited for tasks such as text classification or sentiment analysis.

In [17]:
```
model = LSTM_MODEL()

history = model.fit(
    trainDataset,
    validation_data=testDataset,
    epochs=5,
    validation_steps=30,)
```

```
Epoch 1/5
3990/3990 ━━━━━━━━━━━━━━━━━━━━ 2818s 703ms/step - accuracy: 0.9408 - loss:
0.1792 - precision: 0.7970 - recall: 0.5242 - val_accuracy: 0.9646 - val_lo
ss: 0.1011 - val_precision: 0.9091 - val_recall: 0.7216
Epoch 2/5
3990/3990 ━━━━━━━━━━━━━━━━━━━━ 2836s 702ms/step - accuracy: 0.9623 - loss:
0.1044 - precision: 0.8661 - recall: 0.7460 - val_accuracy: 0.9594 - val_lo
ss: 0.1118 - val_precision: 0.7895 - val_recall: 0.7979
Epoch 3/5
3990/3990 ━━━━━━━━━━━━━━━━━━━━ 2802s 702ms/step - accuracy: 0.9684 - loss:
0.0847 - precision: 0.8787 - recall: 0.8009 - val_accuracy: 0.9594 - val_lo
ss: 0.1327 - val_precision: 0.8989 - val_recall: 0.7273
Epoch 4/5
3990/3990 ━━━━━━━━━━━━━━━━━━━━ 2843s 713ms/step - accuracy: 0.9752 - loss:
0.0671 - precision: 0.8972 - recall: 0.8551 - val_accuracy: 0.9510 - val_lo
ss: 0.1536 - val_precision: 0.7059 - val_recall: 0.7317
Epoch 5/5
3990/3990 ━━━━━━━━━━━━━━━━━━━━ 2914s 716ms/step - accuracy: 0.9817 - loss:
0.0500 - precision: 0.9244 - recall: 0.8945 - val_accuracy: 0.9573 - val_lo
ss: 0.1659 - val_precision: 0.7812 - val_recall: 0.7895
```

STEP 3: Evaluating the Stacked Bidirectional LSTM Model

In this code snippet, the trained stacked bidirectional LSTM model is evaluated on the testDataset using the evaluate() method. This method calculates several key performance metrics: test loss, accuracy, precision, and recall.

In [18]:
```
testLossLSTM, testAccuracyLSTM, testPrecisionLSTM, testRecallLSTM = model.ev
testF1ScoreLSTM = 2 * (testPrecisionLSTM * testRecallLSTM) / (testPrecisionL

print(f"Test Loss: {testLossLSTM}")
print(f"Test Accuracy: {testAccuracyLSTM}")
print(f"Test Precision: {testPrecisionLSTM}")
print(f"Test Recall: {testRecallLSTM}")
print(f"Test F1 Score: {testF1ScoreLSTM}")
```

```
998/998 ━━━━━━━━━━━━━━━━━━━━ 156s 157ms/step - accuracy: 0.9523 - loss: 0.1
687 - precision: 0.7695 - recall: 0.7511
Test Loss: 0.17285260558128357
Test Accuracy: 0.9525614976882935
Test Precision: 0.7765344977378845
Test Recall: 0.7487669587135315
Test F1 Score: 0.762397979426378
```

1. **Test Loss (0.1728)**: This value indicates the average error in the model's predictions compared to the actual labels. A test loss of 0.1728 suggests that the model's predictions are reasonably close to the true labels, though there's still potential for improvement.

2. **Test Accuracy (0.9525)**: The model achieved an accuracy of 95.25%, indicating that it correctly classified over 95% of the test samples. This high accuracy reflects the model's effectiveness in distinguishing between toxic and non-toxic comments.

3. **Test Precision (0.7765)**: With a precision of 77.65%, when the model predicts a comment as toxic, it is correct 77.65% of the time. This indicates a solid performance in minimizing false positives, which is crucial in scenarios where incorrectly labeling non-toxic comments as toxic could have negative implications.

4. **Test Recall (0.7487)**: The recall of 74.87% indicates that the model successfully identifies about 73.21% of the actual toxic comments in the dataset. While this shows good sensitivity, there is room for improvement, as some toxic comments are being missed.

5. **Test F1 Score (0.7623)**: The F1 score, at 76.23%, balances precision and recall, reflecting the model's ability to maintain a reasonable trade-off between identifying true positives and minimizing false positives.

Attention Mechanism (AM)

STEP 1: Inspecting the Shapes of Text and Label Batches

In this code snippet, the `take(1)` method is used to retrieve a single batch from the `trainDataset`, allowing for a quick inspection of the shapes of both the text and label batches. The shapes are printed to provide insight into the dimensions of the data being processed. Specifically, `text_batch.shape` reveals the size and structure of the text data (e.g., the number of samples and the length of each sequence), while `label_batch.shape` indicates the size of the corresponding labels (usually a binary label for each text sample).

In [19]:
```
for text_batch, label_batch in trainDataset.take(1):
    print("Text batch shape:", text_batch.shape)
    print("Label batch shape:", label_batch.shape)
```

```
Text batch shape: (32,)
Label batch shape: (32,)
```

STEP 2: Custom Attention Layer Implementation

- **Initialization**: The `Attention` class inherits from `tf.keras.layers.Layer`. In the `__init__` method, no specific parameters are initialized.
- **Building Weights**: In the `build` method, three trainable weight matrices are created:
  - `W` for transforming the input features,
  - `b` for biasing the transformation,
  - `u` for computing attention scores.
- **Forward Pass**: In the `call` method, the attention mechanism is implemented:
  - A score is computed for each time step by applying a linear transformation followed by a `tanh` activation.
  - The attention weights are obtained by applying the softmax function to these scores, normalizing them across the sequence length.
  - The input sequences are then weighted by these attention scores, and the weighted sequences are summed to produce a context vector that represents the most relevant information from the entire sequence.

The output is this context vector, which can be used in subsequent layers to make predictions, effectively allowing the model to concentrate on important parts of the input

data, thereby improving performance on tasks such as text classification or machine translation.

In [20]:
```
# Custom Attention Layer
class Attention(tf.keras.layers.Layer):
    def __init__(self):
        super(Attention, self).__init__()

    def build(self, input_shape):
        self.W = self.add_weight(shape=(input_shape[-1], input_shape[-1]), i
        self.b = self.add_weight(shape=(input_shape[-1],), initializer='zero
        self.u = self.add_weight(shape=(input_shape[-1],), initializer='rand

    def call(self, inputs):
        # Compute score for each time step
        score = tf.nn.tanh(tf.tensordot(inputs, self.W, axes=1) + self.b)
        # Compute softmax over scores
        attention_weights = tf.nn.softmax(tf.tensordot(score, self.u, axes=1
        # Multiply inputs by attention weights and sum across the time steps
        context_vector = attention_weights[:, :, tf.newaxis] * inputs
        context_vector = tf.reduce_sum(context_vector, axis=1)
        return context_vector  # Return only the context vector
```

STEP 3: RNN Model with Attention Mechanism

Requirements: 2 hidden layers with 64 neurons each, ReLU activation for hidden layers, 1 output neuron with sigmoid activation, Adam optimizer, learning rate of 0.001, 5 epochs, 32 batch size.

The `RNN_MODEL_with_attention()` function constructs a recurrent neural network model that incorporates an attention mechanism to enhance the processing of sequential data.

- **Model Architecture**: The model begins with the `encoder` layer to convert input text into integer sequences, followed by an embedding layer that transforms these integers into dense vector representations of size 64. The `mask_zero=True` parameter allows the model to ignore padding tokens during training.
- **Bidirectional LSTM**: A bidirectional LSTM layer is included, set to return sequences, which enables the model to capture context from both directions of the input data.
- **Attention Layer**: The custom `Attention` layer follows the LSTM, allowing the model to focus on the most relevant parts of the sequence. This layer computes attention weights and produces a context vector that summarizes important information.
- **Dense Layers**: Two fully connected (`Dense`) layers with ReLU activation follow the attention layer, further processing the context vector before the final output layer.
- **Output Layer**: The last layer uses a sigmoid activation function for binary classification, producing a probability that the input text belongs to the toxic or non-toxic class.

- **Compilation**: The model is compiled with the Adam optimizer and binary crossentropy loss, alongside metrics for accuracy, precision, and recall. This architecture is designed to effectively handle tasks such as text classification, where understanding the context and relevance of different input parts is crucial for performance.

In [21]:
```
# RNN Model with Attention
def RNN_MODEL_with_attention():
    model = tf.keras.Sequential([
        encoder,  # The encoder is expected to convert input text into integ
        tf.keras.layers.Embedding(
            input_dim=len(encoder.get_vocabulary()),  # Vocabulary size from
            output_dim=64,  # Embedding dimension
            mask_zero=True  # Masking for padding
        ),
        tf.keras.layers.Masking(),
        tf.keras.layers.Bidirectional(tf.keras.layers.LSTM(64, return_sequer
        Attention(),  # Custom Attention Layer
        tf.keras.layers.Dense(64, activation='relu'),  # Dense layer after a
        tf.keras.layers.Dense(64, activation='relu'),
        tf.keras.layers.Dense(1, activation='sigmoid')  # Output layer for b
    ])

    model.compile(
        optimizer=tf.keras.optimizers.Adam(learning_rate=0.001),
        loss=tf.keras.losses.BinaryCrossentropy(from_logits=False),
        metrics=['accuracy', 'precision', 'recall']
    )

    return model
```

STEP 4: Training the RNN Model with Attention Mechanism

In this code snippet, the `RNN_MODEL_with_attention()` function is called to create an instance of the recurrent neural network model that incorporates an attention mechanism. The model is then trained using the `fit()` method, which uses the `trainDataset` for training and the `testDataset` for validation. The training process runs for 5 epochs, with `validation_steps` set to 30, indicating that the validation will be evaluated on 30 batches of the validation dataset after each epoch.

During training, the model learns to minimize the binary crossentropy loss while optimizing for metrics such as accuracy, precision, and recall. The attention mechanism allows the model to focus on relevant portions of the input sequences, which can enhance performance on tasks like text classification by improving the model's ability to identify important features within the data.

In [22]:
```
model = RNN_MODEL_with_attention()

history = model.fit(
        trainDataset,
        validation_data=testDataset,
        epochs=5,
        validation_steps=30,
    )
```

```
Epoch 1/5
```

```
/usr/local/lib/python3.10/dist-packages/keras/src/layers/layer.py:915: User
Warning: Layer 'attention' (of type Attention) was passed an input with a m
ask attached to it. However, this layer does not support masking and will t
herefore destroy the mask information. Downstream layers will not see the m
ask.
  warnings.warn(
3990/3990 ━━━━━━━━━━━━━━━━━━━━ 1490s 371ms/step - accuracy: 0.9367 - loss:
0.1951 - precision: 0.7887 - recall: 0.4670 - val_accuracy: 0.9656 - val_lo
ss: 0.0974 - val_precision: 0.8902 - val_recall: 0.7526
Epoch 2/5
3990/3990 ━━━━━━━━━━━━━━━━━━━━ 1500s 376ms/step - accuracy: 0.9634 - loss:
0.0985 - precision: 0.8734 - recall: 0.7500 - val_accuracy: 0.9625 - val_lo
ss: 0.0982 - val_precision: 0.8152 - val_recall: 0.7979
Epoch 3/5
3990/3990 ━━━━━━━━━━━━━━━━━━━━ 1474s 364ms/step - accuracy: 0.9696 - loss:
0.0793 - precision: 0.8832 - recall: 0.8091 - val_accuracy: 0.9552 - val_lo
ss: 0.1612 - val_precision: 0.8317 - val_recall: 0.7636
Epoch 4/5
3990/3990 ━━━━━━━━━━━━━━━━━━━━ 1467s 368ms/step - accuracy: 0.9788 - loss:
0.0575 - precision: 0.9117 - recall: 0.8772 - val_accuracy: 0.9563 - val_lo
ss: 0.1432 - val_precision: 0.7381 - val_recall: 0.7561
Epoch 5/5
3990/3990 ━━━━━━━━━━━━━━━━━━━━ 1601s 401ms/step - accuracy: 0.9850 - loss:
0.0396 - precision: 0.9376 - recall: 0.9144 - val_accuracy: 0.9448 - val_lo
ss: 0.2324 - val_precision: 0.7059 - val_recall: 0.7579
```

STEP 5: Evaluating the RNN Model with Attention Mechanism

In this code snippet, the trained RNN model with an attention mechanism is evaluated on the testDataset using the evaluate() method. This evaluation yields key performance metrics: test loss, accuracy, precision, and recall.

In [24]:
```
testLossAM, testAccuracyAM, testPrecisionAM, testRecallAM = model.evaluate(t
testF1ScoreAM = 2 * (testPrecisionAM * testRecallAM) / (testPrecisionAM + te

print(f"Test Loss: {testLossAM}")
print(f"Test Accuracy: {testAccuracyAM}")
print(f"Test Precision: {testPrecisionAM}")
print(f"Test Recall: {testRecallAM}")
print(f"Test F1 Score: {testF1ScoreAM}")
```

```
998/998 ━━━━━━━━━━━━━━━━━━━━ 86s 87ms/step - accuracy: 0.9495 - loss: 0.205
6 - precision: 0.7417 - recall: 0.7644
Test Loss: 0.21055659651756287
Test Accuracy: 0.9495848417282104
Test Precision: 0.7464576363563538
Test Recall: 0.76325523853302
Test F1 Score: 0.7547629893978501
```

1. **Test Loss (0.2106)**: This metric represents how well the model performed on the test dataset in terms of the loss function (binary cross-entropy in this case). A lower loss indicates a better fit of the model to the data. Here, the loss is reasonably low, showing that the model is performing well in distinguishing between toxic and non-toxic comments.

2. **Test Accuracy (0.9496 or 94.96%)**: Accuracy measures the proportion of correctly predicted labels out of all predictions. In this case, the model has correctly classified about 94.96% of the comments as either toxic or non-toxic, which indicates high performance overall.

3. **Test Precision (0.7465 or 74.65%)**: Precision refers to the proportion of positive identifications (toxic comments) that were actually correct. A precision of 74.65% means that out of all the comments the model predicted as toxic, 74.65% were indeed toxic. Precision is important when false positives (incorrectly identifying non-toxic comments as toxic) need to be minimized.

4. **Test Recall (0.7633 or 76.33%)**: Recall (also known as sensitivity or true positive rate) measures the proportion of actual toxic comments that were correctly identified by the model. With a recall of 76.33%, the model successfully identified 76.33% of the toxic comments. Recall is crucial when false negatives (failing to identify toxic comments) are a bigger concern.

5. **Test F1 Score (0.7548 or 75.48%)**: The F1 score is the harmonic mean of precision and recall, and it provides a balanced measure of the two. It is particularly useful when there is an uneven class distribution or when both false positives and false negatives are important. Here, the F1 score of 75.48% indicates a good trade-off between precision and recall.

Comparison of Evaluation metrics

In [32]:
```
# Create a dictionary with all the metrics
data = {
    "Model": ["FNN", "Logistic Regression", "RNN", "LSTM", "RNN with Attenti
    "Accuracy": [testAccuracyFnn, lr_accuracy, testAccuracyRnn, testAccuracy
    "Precision": [testPrecisionFnn, lr_precision, testPrecisionRnn, testPrec
    "Recall": [testRecallFnn, lr_recall, testRecallRnn, testRecallLSTM, test
    "F1 Score": [testF1ScoreFnn, lr_f1, testF1ScoreRnn, testF1ScoreLSTM, tes
}

# Create a DataFrame to store the metrics
evaluation_metrics = pd.DataFrame(data)

# Optionally, sort the DataFrame by Accuracy or any other metric
evaluation_metrics_sorted = evaluation_metrics.sort_values(by="Accuracy", as

# Print sorted table
print("\nSorted by Accuracy:\n", evaluation_metrics_sorted)
```

```
Sorted by Accuracy:
                  Model  Accuracy  Precision    Recall  F1 Score
0                   FNN  0.955945   0.815591  0.732121  0.771605
3                  LSTM  0.952561   0.776534  0.748767  0.762398
2                   RNN  0.949929   0.752764  0.755549  0.754154
4   RNN with Attention  0.949585   0.746458  0.763255  0.754763
1  Logistic Regression  0.949500   0.783200  0.696100  0.737100
```

1. **Accuracy**: FNN leads with the highest accuracy (95.59%), followed closely by LSTM, RNN, and the RNN with Attention model. Logistic Regression (94.95%) is only marginally lower, indicating that traditional machine learning still competes well with deep learning models. However, accuracy alone doesn't capture the full picture, as it doesn't reflect the model's ability to balance precision and recall effectively in detecting toxic comments.

2. **Precision vs. Recall**:
    - **Precision** measures how well the model avoids false positives (i.e., labeling non-toxic comments as toxic). Here, FNN (81.56%) excels, followed by Logistic Regression (78.32%), making these models highly reliable for reducing false positives. This makes them ideal for applications where minimizing false accusations of toxicity is critical, such as automated moderation systems.
    - **Recall**, on the other hand, indicates the model's sensitivity to detecting true positives (i.e., catching all toxic comments). Models with higher recall like RNN with Attention (76.33%) and LSTM (74.88%) are more likely to catch most toxic comments but might classify more non-toxic comments as toxic, leading to false positives.
3. **Balancing Precision and Recall**:
    - **F1 Score**: The F1 score provides a balanced view of both precision and recall, where **LSTM** (76.24%) and **RNN with Attention** (75.48%) strike a good balance, slightly favoring recall. These models are suitable for scenarios where both false positives and false negatives are equally important to avoid.
    - FNN (77.16%) has the highest F1 score due to its higher precision, while Logistic Regression, despite its high precision, has the lowest F1 score (73.71%) because of its poorer recall (69.61%).
4. **Attention Mechanism**: The **RNN with Attention** model adds a slight boost to recall (76.33%) over the standard RNN (75.55%), demonstrating its ability to focus on the most relevant time steps in sequence data, improving the model's ability to identify toxic comments. However, this comes at the cost of slightly reduced precision, as the attention mechanism might overemphasize certain features, leading to more false positives.

5. **Deep Learning vs. Logistic Regression**: While **Logistic Regression** achieves good precision (78.32%), it suffers from low recall (69.61%), resulting in a lower F1 score compared to the deep learning models. This suggests that Logistic Regression is more conservative in identifying toxic comments, missing a significant portion of them but making fewer mistakes in terms of false positives. In contrast, deep learning models, especially those with LSTM or RNN architectures, show superior performance by balancing precision and recall more effectively.

Summary:

- The **FNN** model has the highest accuracy and precision, making it the most reliable in classifying toxic comments but less sensitive in detecting all toxic comments (lower recall).
- **LSTM** achieves a strong balance between precision and recall, with competitive accuracy, making it one of the more balanced models.
- **RNN** and **RNN with Attention** both offer a good balance between precision and recall, though the **attention mechanism** slightly improves recall while slightly lowering precision.
- **Logistic Regression** lags behind deep learning models in recall and F1 score, though it maintains high precision, showing it is a simpler yet reasonably effective

baseline model.