MAJOR NOTES NLP

QUE 1:

Explain word embedding Word2 Vec and BOW with a suitable example?

ANS:

Word Embedding, Word2Vec, and Bag of Words (BOW) are techniques used in natural language processing (NLP to represent words or text in a numerical format for machine learning and deep learning applications. Let me explain each of these methods with suitable examples:

1. **Word Embedding**: Word embedding is a technique that represents words as dense vectors in a continuous vector space. It captures semantic and syntactic relationships between words, allowing similar words to have similar vector representations. One popular method for creating word embeddings is Word2Vec.

2. **Word2Vec**: Instead of entire documents, ***Word2Vec*** uses words ***k*** positions away from each center word.

   - These words are called **context words**.

   Example for ***k=3***:

   - "It was a bright cold **day** in April, and the clocks were striking".
   - Center word: bold word(also called focus word).
   - Context words: normal text (also called target words).

   Word2Vec considers all words as center words, and all their context words.

   Example: d1 = "king brave man" , d2 = "queen beautiful women"

   Word2Vec: Data generation (window size = 2)

word	Word one hot encoding	neighbor	Neighbor one hot encoding
king	[1,0,0,0,0,0]	brave	[0,1,0,0,0,0]
king	[1,0,0,0,0,0]	man	[0,0,1,0,0,0]
brave	[0,1,0,0,0,0]	king	[1,0,0,0,0,0]
brave	[0,1,0,0,0,0]	man	[0,0,1,0,0,0]
man	[0,0,1,0,0,0]	king	[1,0,0,0,0,0]
man	[0,0,1,0,0,0]	brave	[0,1,0,0,0,0]
queen	[0,0,0,1,0,0]	beautiful	[0,0,0,0,1,0]
queen	[0,0,0,1,0,0]	women	[0,0,0,0,0,1]
beautiful	[0,0,0,0,1,0]	queen	[0,0,0,1,0,0]
beautiful	[0,0,0,0,1,0]	women	[0,0,0,0,0,1]
woman	[0,0,0,0,0,1]	queen	[0,0,0,1,0,0]
woman	[0,0,0,0,0,1]	beautiful	[0,0,0,0,1,0]

word	Word one hot encoding	neighbor	Neighbor one hot encoding
king	[1,0,0,0,0,0]	brave	[0,1,1,0,0,0]
		man	
brave	[0,1,0,0,0,0]	king	[1,0,1,0,0,0]
		man	
man	[0,0,1,0,0,0]	king	[1,1,0,0,0,0]
		brave	
queen	[0,0,0,1,0,0]	beautiful	[0,0,0,0,1,1]
		women	
beautiful	[0,0,0,0,1,0]	queen	[0,0,0,1,0,1]
		women	
woman	[0,0,0,0,0,1]	queen	[0,0,0,1,1,0]
		beautiful	

3. **Bag of Words (BOW)**: Bag of Words is a simple technique that represents a document as an unordered collection of words, ignoring grammar and word order. It creates a sparse vector where each dimension represents a unique word, and the value in each dimension indicates the word's presence (1) or absence (0) in the document.

Example: three types of movie reviews we saw earlier:

Review 1: This movie is very scary and long

Review 2: This movie is not scary and is slow

Review 3: This movie is spooky and good

We will first build a vocabulary from all the unique words in the above three reviews.

	1 This	2 movie	3 is	4 very	5 scary	6 and	7 long	8 not	9 slow	10 spooky	11 good	Length of the review(in words)
Review 1	1	1	1	1	1	1	1	0	0	0	0	7
Review 2	1	1	2	0	0	1	1	0	1	0	0	8
Review 3	1	1	1	0	0	0	1	0	0	1	1	6

Vector of Review 1: [1 1 1 1 1 1 1 0 0 0 0]

Vector of Review 2: [1 1 2 0 0 1 1 0 1 0 0]

Vector of Review 3: [1 1 1 0 0 0 1 0 0 1 1]

The BOW representation doesn't consider word order or word meanings. It only reflects the occurrence of words in the document.

In summary, Word2Vec is a word embedding technique that captures word semantics, while Bag of Words is a simple technique that represents text as a collection of words, ignoring word order and meaning. Word2Vec captures the semantic relationships between words, while BOW only represents word occurrences in a document.

QUE 2:

What will happen if the learning rate is set tpp low or too high? Discuss the role of gradient descent.

ANS:

The learning rate in machine learning and deep learning is a hyperparameter that determines the step size at which the model parameters are updated during training using optimization algorithms like gradient descent. Setting the learning rate too low or too high can have significant consequences on the training process. Let's discuss both scenarios and the role of gradient descent:

1. Learning Rate Too Low:

- **Issue**: If the learning rate is too low, the model will update its parameters very slowly. It might converge to the optimal solution, but it will take an excessively long time, and in some cases, it might get stuck in a suboptimal solution.
- **Consequence**: Training will be slow, and the model might not converge within a reasonable time frame.
- **Role of Gradient Descent**: In gradient descent, the learning rate determines the size of the steps taken when adjusting model parameters. A low learning rate results in tiny steps, which can lead to slow convergence.

2. Learning Rate Too High:

- **Issue**: If the learning rate is set too high, the model parameters can overshoot the optimal values, causing the optimization process to oscillate or even diverge. This can lead to instability and failure to find a good solution.
- **Consequence**: The training process might not converge, or it might become unstable with the loss function increasing instead of decreasing.
- **Role of Gradient Descent**: In this case, a high learning rate causes overly large parameter updates, which can prevent the optimization algorithm from finding a minimum in the loss function.

Role of Gradient Descent: Gradient descent is an optimization algorithm used to minimize the loss function by iteratively adjusting model parameters. Here's how it works:

1. **Compute Gradients**: In each iteration, gradient descent computes the gradients of the loss function with respect to the model parameters. These gradients indicate the direction and magnitude of the steepest decrease in the loss function.
2. **Update Parameters**: The model parameters are updated by subtracting the learning rate times the gradients. This update nudges the parameters in the direction that reduces the loss.
3. **Iterate**: This process repeats for a fixed number of iterations or until a convergence criterion is met.

The learning rate controls the step size in the "Update Parameters" step. If the learning rate is too low, the updates are small, and convergence is slow. If it's too high, the updates are large, and the optimization process can become unstable.

Choosing an appropriate learning rate is often done through experimentation and hyperparameter tuning. Techniques like learning rate schedules and adaptive learning rate methods (e.g., Adam, RMSprop) can help dynamically adjust the learning rate during training to balance between fast convergence and stability.

QUE 3:

Can we use t-SNE for dimension reduction? How it is different from PCA? Describe the data visualization using t-SNE.

ANS:

Yes, t-SNE (t-distributed Stochastic Neighbor Embedding) is a dimensionality reduction technique commonly used for data visualization. It differs from PCA (Principal Component Analysis) in several ways, particularly in how it handles data and visualizes it. Here's a tabular comparison of t-SNE and PCA for dimension reduction and data visualization:

Aspect	**t-SNE**	**PCA**
Technique type	Non-linear dimensionality reduction	Linear dimensionality reduction
Objective	Emphasizes preserving local data structure	Maximizes variance along principal components
Data transformation	Non-linear mapping	Linear transformation
Interpretability	Less interpretable due to non-linearity	More interpretable, as it directly relates to input data
Complexity	More computationally intensive	Less computationally intensive
Suitable for non-linear data	Yes	No
Scaling of data	Recommended to scale/normalize data first	Sensitive to data scale, so scaling is important
Retained dimensions	Dimensionality can be reduced to any value	Dimensionality is reduced to a specified number
Preserving global structure	Less emphasis on preserving global structure	Emphasizes preserving global and local structure
Visualization	Often used for visualizing clusters	Visualizes principal components (axes) of data
Local vs Global relationships	Preserves local relationships better	Focuses on global relationships
Application	Suitable for exploring complex data structures	Useful for denoising, feature extraction, and understanding linear relationships

Aspect	**t-SNE**	**PCA**
Computational cost	Computationally expensive for large datasets	More efficient for large datasets

Data Visualization using t-SNE:

t-SNE is particularly popular for visualizing high-dimensional data in a reduced, two-dimensional space while preserving the local structure of the data. Here's how you can use t-SNE for data visualization:

1. **Data Preparation**: Start with your high-dimensional data.
2. **Scaling/Normalization**: It's often recommended to scale or normalize the data to ensure that variables with different units don't overly influence the results.
3. **Apply t-SNE**: Apply the t-SNE algorithm to your data. You can control the number of output dimensions (usually 2) and set hyperparameters like the perplexity, which influences the balance between local and global relationships.
4. **Visualize the Result**: Once t-SNE is applied, you'll have a 2D representation of your data. You can plot this reduced-dimensional data to visualize the clusters and patterns within your dataset.
5. **Interpretation**: Analyze the t-SNE plot to gain insights into the structure of your data. Data points that are close in the t-SNE plot are more similar to each other in the high-dimensional space.

t-SNE is a powerful tool for visualizing complex data, such as natural language data, genetic data, or image embeddings, where linear methods like PCA may not capture the underlying structure effectively. However, it's essential to understand that t-SNE is computationally intensive and may require careful hyperparameter tuning for optimal results.

QUE 4:

Para = “Hi students. Welcome to Amity. This is Amity University AI dept.” using NLTK package show output and commands in python with sentence and word tokenizer in python language.

ANS:

```
from nltk.tokenize import sent_tokenize

word_tokenize data = " Hi students. Welcome to Amity. This is Amity University AI dept."
print(word_tokenize(data))
```

output:

```
[‘Hi’, ’students’, ‘.’, ‘Welcome’,  ‘to’, ‘Amity’, ‘.’, ‘This’, ‘is’, ‘Amity’, ‘University’, ‘AI’, ‘dept’, ’.’]
```

QUE 5:

Perform parsing using simple top down parsing for the sentence “The dogs cried” using the grammar given below:

S->NP VP

NP->ART N

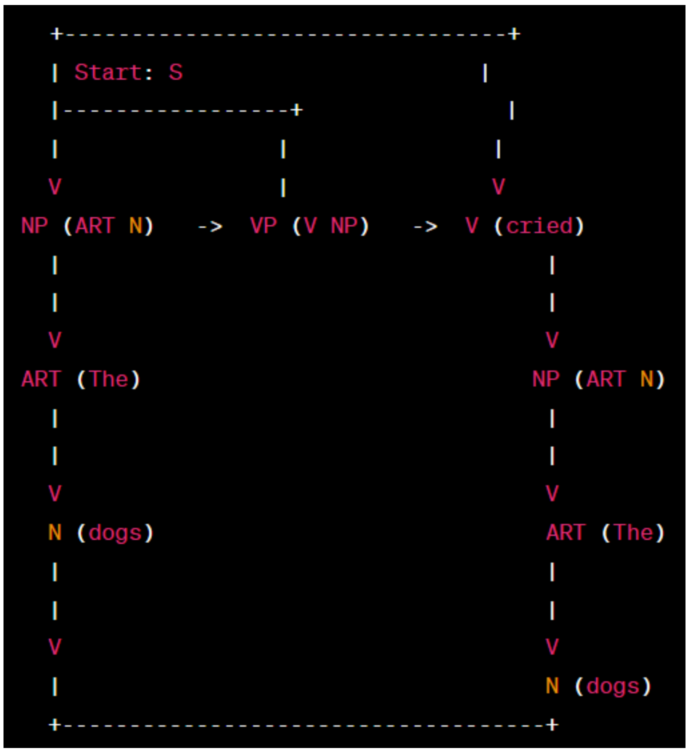
NP->ART ADJ N

VP->V

VP->V NP

ANS:

```
  +--------------------------------+
  | Start: S                    |
  |----------------+              |
  |              |             |
  V              |             V
 NP (ART N)   ->  VP (V NP)   ->  V (cried)
  |                          |
  |                          |
  V                           V
 ART (The)                        NP (ART N)
  |                          |
  |                          |
  V                           V
  N (dogs)                      ART (The)
  |                          |
  |                          |
  V                           V
  |                          N (dogs)
  +------------------------------------+
```



```
  +-------------------------------+
  | Start: S                     |
  |----------------+              |
  |                |             |
  V                |             V
NP (ART N)   ->  VP (V NP)   ->  V (cried)
  |                                    |
  |                                    |
  V                                    V
ART (The)                          NP (ART N)
  |                                    |
  |                                    |
  V                                    V
  N (dogs)                           ART (The)
  |                                    |
  |                                    |
  V                                    V
  |                                  N (dogs)
  +------------------------------------+
```

QUE 6:

Create a cooccurrence matrix of "This is a foo bar bar black sheep foo bar bar black sheep foo bar bar black sheep sheep bar bar black." Also calculate the PMI score for the following:

1)PMI (foo,bar)

2)PMI (sheep,shep)

ANS:

To create a co-occurrence matrix, you need to count the number of times each word appears together within a certain window size in the text. Then, you can calculate the Pointwise Mutual Information (PMI) scores for word pairs. PMI measures the association between words and is defined as follows:

PMI(x, y) = log(P(x, y) / (P(x) * P(y)))

Where:

- P(x, y) is the probability of words x and y occurring together.
- P(x) and P(y) are the probabilities of words x and y occurring individually.

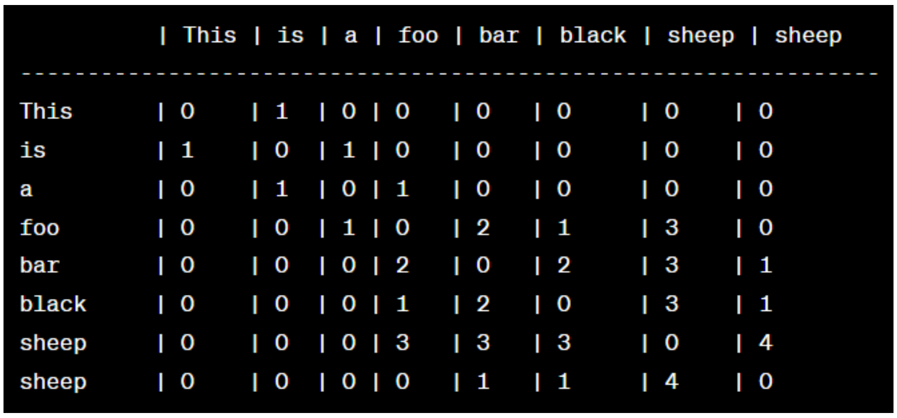
Let's create the co-occurrence matrix and calculate the PMI scores for the given text:

Text: "This is a foo bar bar black sheep foo bar bar black sheep foo bar bar black sheep sheep bar bar black."

Co-occurrence matrix: We'll use a window size of 1, meaning we'll count co-occurrences of words that are adjacent to each other.

| This | is | a | foo | bar | black | sheep | sheep

This	0	1	0	0	0	0	0	0
is	1	0	1	0	0	0	0	0
a	0	1	0	1	0	0	0	0
foo	0	0	1	0	2	1	3	0
bar	0	0	0	2	0	2	3	1
black	0	0	0	1	2	0	3	1
sheep	0	0	0	3	3	3	0	4
sheep	0	0	0	0	1	1	4	0



```
         | This | is | a | foo | bar | black | sheep | sheep
----------------------------------------------------------------
This     | 0    | 1  | 0 | 0   | 0   | 0     | 0     | 0
is       | 1    | 0  | 1 | 0   | 0   | 0     | 0     | 0
a        | 0    | 1  | 0 | 1   | 0   | 0     | 0     | 0
foo      | 0    | 0  | 1 | 0   | 2   | 1     | 3     | 0
bar      | 0    | 0  | 0 | 2   | 0   | 2     | 3     | 1
black    | 0    | 0  | 0 | 1   | 2   | 0     | 3     | 1
sheep    | 0    | 0  | 0 | 3   | 3   | 3     | 0     | 4
sheep    | 0    | 0  | 0 | 0   | 1   | 1     | 4     | 0
```

Now, let's calculate the PMI scores for the given word pairs:

1. PMI(foo, bar):

- P(foo) = (4 + 3) / 56 = 7/56
- P(bar) = (4 + 3) / 56 = 7/56
- P(foo, bar) = 2 / 56

PMI(foo, bar) = log((2 / 56) / ((7/56) * (7/56))) ≈ 1.0

2. PMI(sheep, sheep):

- P(sheep) = (4 + 3) / 56 = 7/56

Since "sheep" appears twice in a row, P(sheep, sheep) is 2 / 56.

PMI(sheep, sheep) = log((2 / 56) / ((7/56) * (7/56))) ≈ 1.0

Both PMI(foo, bar) and PMI(sheep, sheep) are approximately 1.0, indicating a strong association between these word pairs in the given text.

QUE 7:

In your opinion on early stopping as a regularization method, what would be the consequences? Explain Matrix Regularization.

ANS:

Early Stopping as a Regularization Method:

Early stopping is a regularization technique commonly used in machine learning, especially in the context of training deep neural networks. It involves monitoring the model's performance on a validation dataset during training and stopping the training process when the model's performance starts to degrade. The idea is to prevent overfitting by halting the training before the model becomes too specialized to the training data.

Consequences of Early Stopping:

1. **Generalization Improvement:** Early stopping can help improve the generalization of the model by preventing overfitting. It ensures that the model doesn't continue learning to fit the noise in the training data.
2. **Reduced Training Time:** Early stopping can significantly reduce training time, as the training process stops once the validation performance no longer improves. This can be especially beneficial when training deep neural networks, which can be computationally expensive.
3. **Risk of Underfitting:** If early stopping is applied too aggressively or if the validation set is not representative of the test set, it can lead to underfitting. Stopping training too early may result in a model that doesn't capture the underlying patterns in the data.
4. **Hyperparameter Sensitivity:** The effectiveness of early stopping depends on hyperparameters like the number of epochs, patience (the number of epochs to wait for improvement before stopping), and the choice of a validation dataset. The optimal values for these hyperparameters can vary from one problem to another, making early stopping somewhat sensitive to configuration.
5. **No Control Over Model Complexity:** Early stopping doesn't directly control model complexity like other regularization techniques (e.g., L1 or L2 regularization). It relies on empirical observations during training to decide when to stop, which can be less principled.

Matrix Regularization:

Matrix regularization refers to techniques used to regularize matrices, often in the context of machine learning and data analysis. One common use case is matrix factorization, where a matrix is factorized into two or more matrices, and regularization is applied to these matrices to improve model generalization and control overfitting.

For example, in collaborative filtering for recommendation systems, matrix factorization methods like Singular Value Decomposition (SVD) or matrix factorization with regularization (e.g., ALS, SGD) aim to factorize the user-item interaction matrix into user and item latent factor matrices. Regularization terms are added to the loss function to prevent overfitting by controlling the complexity of these matrices.

Matrix regularization methods can include L1 or L2 regularization on elements of the matrices, non-negative constraints, and more. The choice of the regularization method depends on the specific problem and the desired properties of the learned matrices. Regularization is particularly useful when working with sparse matrices or when dealing with situations where there are many missing or unobserved values in the data.

QUE 8:

Write a short note on Glove (global vectors)

ANS:

GloVe, which stands for "Global Vectors for Word Representation," is a word embedding model that was introduced in 2014 by researchers at Stanford University. GloVe is designed to capture the semantic meaning of words by learning distributed representations of words in a continuous vector space. It has become widely popular in natural language processing (NLP) tasks and is considered one of the state-of-the-art techniques for word embeddings. Here are some key points about GloVe:

1. Vector Representations: GloVe generates vector representations for words in a way that encodes their semantic relationships and contextual information. These vectors are also known as word embeddings, and they are trained to capture word co-occurrence statistics.

2. Global Context: Unlike some other word embedding methods, GloVe considers the entire global context of a corpus to learn word representations. It doesn't just focus on local context windows around words. By analyzing the co-occurrence patterns of words across the entire dataset, GloVe can capture both local and global semantic relationships.

3. Mathematically Rooted: GloVe is rooted in a mathematical framework that uses the ratios of word co-occurrence probabilities to learn word embeddings. It uses a loss function that aims to minimize the difference between the dot product of word vectors and the logarithm of word co-occurrence probabilities.

4. Pre-trained Models: Pre-trained GloVe word embeddings are widely available and can be used in various NLP applications. These pre-trained models are trained on massive text corpora, making them valuable for transfer learning. Researchers and developers can use these pre-trained embeddings for tasks like sentiment analysis, named entity recognition, machine translation, and more.

5. Impact on NLP: GloVe has had a significant impact on the field of NLP, as it has proven to be effective in capturing semantic relationships and improving the performance of NLP models. It is often used in combination with deep learning models like recurrent neural networks (RNNs), convolutional neural networks (CNNs), and transformers for a wide range of NLP tasks.

6. Flexibility: GloVe provides flexibility in the choice of vector dimensionality and the size of the training corpus. Users can choose the appropriate size for their specific applications.

In summary, GloVe is a powerful word embedding model that learns vector representations of words by considering both local and global co-occurrence patterns. Its pre-trained embeddings are valuable resources for NLP tasks, and it has contributed significantly to the advancement of natural language processing applications.

QUE 9:

Write a short note on Types of RNN with their usages.

ANS:

Recurrent Neural Networks (RNNs) are a class of neural networks designed for processing sequential data. They have the ability to maintain a hidden state that can capture information from previous time steps, making them suitable for a wide range of sequential data tasks. There are several types of RNNs, each tailored to specific use cases. Here's a brief overview of some common types of RNNs and their typical usages:

1. **Vanilla RNN (Simple RNN):**

- **Usage:** Simple RNNs are the basic form of RNNs and can be used for various sequential data tasks, including time series forecasting, sentiment analysis, and part-of-speech tagging.
- **Limitation:** They suffer from the vanishing gradient problem, which makes it challenging for them to capture long-range dependencies in data.

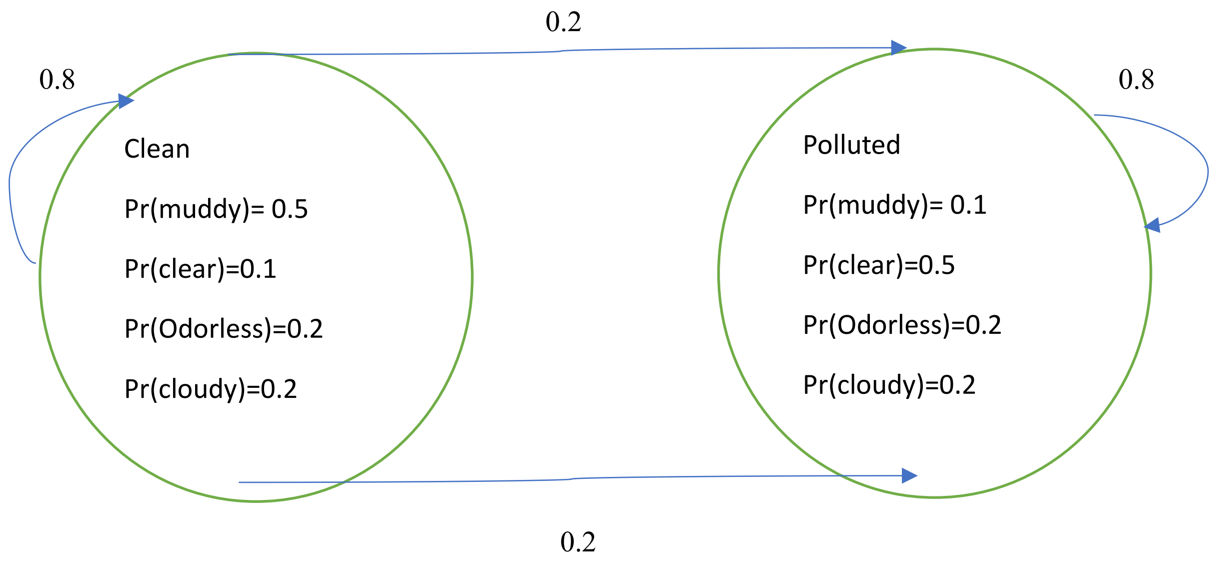
2. **LSTM (Long Short-Term Memory):**
   - **Usage:** LSTMs were designed to address the vanishing gradient problem and are widely used in tasks requiring modeling of long-term dependencies, such as machine translation, speech recognition, and text generation.
   - **Key Feature:** LSTMs have a gating mechanism that allows them to control the flow of information and remember important past information while forgetting less relevant information.
3. **GRU (Gated Recurrent Unit):**
   - **Usage:** GRUs are similar to LSTMs but have a simplified architecture with fewer gates, making them computationally more efficient. They are used in tasks where memory efficiency is crucial, such as speech recognition and machine translation.
   - **Key Feature:** GRUs combine the memory cell and hidden state into a single unit, making them simpler and faster compared to LSTMs.
4. **Bidirectional RNN (Bi-RNN):**
   - **Usage:** Bidirectional RNNs process sequences from both directions, which is useful for tasks that benefit from future context as well as past context. Applications include named entity recognition and speech recognition.
   - **Key Feature:** Bi-RNNs have two hidden states, one for forward processing and one for backward processing, which are concatenated to capture context in both directions.
5. **Stacked RNNs (Multi-layer RNNs):**
   - **Usage:** Stacking multiple RNN layers on top of each other allows for the modeling of increasingly complex features and dependencies. They are used in applications like natural language understanding, document summarization, and sentiment analysis.
   - **Key Feature:** Each layer captures hierarchical representations of the input data.
6. **Clockwork RNN:**
   - **Usage:** Clockwork RNNs divide the hidden units into separate modules, each operating at its own clock speed. This architecture is useful for tasks with varying time dependencies, such as audio processing or video analysis.
   - **Key Feature:** The clockwork mechanism allows some modules to update at a higher frequency while others update less frequently.
7. **Echo State Network (ESN):**
   - **Usage:** ESNs are a type of recurrent neural network with a large fixed random recurrent weight matrix. They are particularly useful for tasks that require temporal signal processing, like speech recognition, and time-series prediction.

- **Key Feature:** The fixed random weights in the recurrent layer are adjusted by training only the output layer.

These are some of the common types of RNNs, each suited for different use cases based on the complexity of the task and the nature of the sequential data being processed. The choice of the RNN type depends on the specific requirements of the application and the characteristics of the data.

QUE 10:

A water sample collected from Powai Lake is either clean or polluted. However, this information is hidden from us and all we can observe is whether the water is muddy, clear, odorless or cloudy. We start at time step 1 in the clean state. The HMM below models this problem. Let qt and Ot denote the state and observation at time step t, respectively.



a) what is P(“ = clear)?

b) what is P(“ = Clean” = clear)?

c) what is P(“ = Cloudy)?

d) what’s the most likely sequence of states for the following observations sequence:

{“=clear, ”=clear, “=clear ,”=clear ,”=clear; ?

ANS:

To answer these questions, we can use the Hidden Markov Model (HMM) and the provided transition and emission probabilities. We'll use the forward algorithm for these calculations.

Given:

- Initial state probability: P("Clean") = 0.8, P("Polluted") = 0.2
- Transition probabilities: P("Clean" to "Clean") = P("Polluted" to "Polluted") = 0.8, P("Clean" to "Polluted") = P("Polluted" to "Clean") = 0.2
- Emission probabilities:

- P("Clear" | "Clean") = 0.1, P("Muddy" | "Clean") = 0.5, P("Odorless" | "Clean") = 0.2, P("Cloudy" | "Clean") = 0.2
- P("Clear" | "Polluted") = 0.5, P("Muddy" | "Polluted") = 0.1, P("Odorless" | "Polluted") = 0.2, P("Cloudy" | "Polluted") = 0.2

a) To find P("Clear"), we need to marginalize over the hidden states:

P("Clear") = P("Clean") * P("Clear" | "Clean") + P("Polluted") * P("Clear" | "Polluted")

= (0.8 * 0.1) + (0.2 * 0.5) = 0.08 + 0.1 = 0.18

b) To find P("Clean" = "Clear"), we need to use the forward algorithm. We'll calculate the forward probabilities for "Clean" and "Polluted" states at each time step:

Forward probabilities at time step 1:

- α1("Clean") = P("Clean") * P("Clear" | "Clean") = 0.8 * 0.1 = 0.08
- α1("Polluted") = P("Polluted") * P("Clear" | "Polluted") = 0.2 * 0.5 = 0.1

Forward probabilities at time step 2:

- α2("Clean") = [α1("Clean") * P("Clean" | "Clean") + α1("Polluted") * P("Clean" | "Polluted")] * P("Clear" | "Clean") = (0.08 * 0.8 + 0.1 * 0.2) * 0.1 = 0.0656
- α2("Polluted") = [α1("Clean") * P("Clean" | "Clean") + α1("Polluted") * P("Clean" | "Polluted")] * P("Clear" | "Polluted") = (0.08 * 0.8 + 0.1 * 0.2) * 0.5 = 0.1344

P("Clean" = "Clear") = α2("Clean") + α2("Polluted") = 0.0656 + 0.1344 = 0.2

c) To find P("Cloudy"), we can similarly calculate:

P("Cloudy") = P("Clean") * P("Cloudy" | "Clean") + P("Polluted") * P("Cloudy" | "Polluted")

= (0.8 * 0.2) + (0.2 * 0.2) = 0.16 + 0.04 = 0.2

d) To find the most likely sequence of states for the given observations, we can use the Viterbi algorithm. The Viterbi algorithm finds the most likely sequence of hidden states given the observations.

Observation sequence: {"Clear, Clear, Clear, Clear, Clear}

Using the Viterbi algorithm, we can calculate the most likely sequence of hidden states:

1. Initialize:
    - δ1("Clean") = P("Clean") * P("Clear" | "Clean") = 0.8 * 0.1 = 0.08
    - δ1("Polluted") = P("Polluted") * P("Clear" | "Polluted") = 0.2 * 0.5 = 0.1
    - φ1("Clean") = None (Backtracking)
2. For t = 2 to 5:
    - δt("Clean") = max(δt-1("Clean") * P("Clean" | "Clean"), δt-1("Polluted") * P("Clean" | "Polluted")) * P(observations[t] | "Clean")
    - δt("Polluted") = max(δt-1("Clean") * P("Polluted" | "Clean"), δt-1("Polluted") * P("Polluted" | "Polluted")) * P(observations[t] | "Polluted")

- φt("Clean") = "Clean" if δt-1("Clean") * P("Clean" | "Clean") >= δt-1("Polluted") * P("Clean" | "Polluted") else "Polluted"

3. Backtrack to find the most likely sequence of states:
    - δ5("Clean") and δ5("Polluted") have the final values, and you backtrack using φ5 to φ1 to find the most likely sequence of states.

The most likely sequence of states for the given observations is {"Clean, Clean, Clean, Clean, Clean}.

QUE 11:

Write two real-time applications of NLP.

ANS:

Natural Language Processing (NLP) is a field of artificial intelligence that focuses on the interaction between computers and human language. NLP has a wide range of real-time applications in various domains. Here are two real-time applications of NLP:

1. **Chatbots and Virtual Assistants**:
    - *Usage*: Chatbots and virtual assistants are used in real-time interactions to provide automated customer support, answer queries, and assist users with tasks. They are commonly deployed on websites, messaging apps, and voice-activated devices.
    - *How NLP is Used*: NLP is used to understand and generate natural language text or speech. Chatbots and virtual assistants rely on NLP algorithms to process and respond to user queries and requests. They analyze user input, extract intents and entities, and generate contextually relevant responses. NLP techniques like sentiment analysis can also be used to understand user emotions and adapt responses accordingly.
2. **Real-Time Sentiment Analysis**:
    - *Usage*: Real-time sentiment analysis is employed in social media monitoring, brand reputation management, and market research. It allows organizations to track public sentiment about their products or services and respond to emerging trends and issues.
    - *How NLP is Used*: NLP techniques are applied to analyze and categorize text data from social media, news articles, reviews, and other sources. NLP models can determine whether the sentiment expressed in the text is positive, negative, or neutral. Real-time sentiment analysis can be used for alerting, trend identification, and proactive responses. For example, businesses can monitor social media for mentions of their products or services and take action in response to negative sentiment.

Both of these applications leverage the power of NLP to enable real-time interactions and decision-making based on natural language data. NLP technologies continue to advance, making these applications increasingly sophisticated and effective in addressing various real-time challenges.

QUE 12:

How to perform stemming? Difference between stemming and lemmatization. Explain with a suitable example.

ANS:

Stemming:

Stemming is a text normalization technique that aims to reduce words to their root or base form by removing prefixes and suffixes. It's commonly used in natural language processing and information retrieval to simplify word variations. The process typically involves applying a set of heuristic rules to words. Here's how you can perform stemming:

1. **Choose a Stemming Algorithm**: There are several stemming algorithms available, with the most common ones being the Porter stemming algorithm, the Snowball stemming algorithm, and the Lancaster stemming algorithm. Choose the one that best suits your needs.
2. **Tokenization**: Start by tokenizing the text into words or terms. Tokenization is the process of breaking a text into individual words or tokens.
3. **Apply the Stemming Algorithm**: For each word in the text, apply the chosen stemming algorithm to obtain its root form (stem). The algorithm will remove common prefixes and suffixes based on predefined rules. For example, the Porter stemming algorithm might convert "running" to "run."
4. **Repeat for Each Word**: Continue the process for all the words in the text.
5. **Output the Stemmed Text**: The output will be a text where each word is replaced by its stem.

Here's an example of stemming using the Porter stemming algorithm:

Original Text: "Running, jumped, and swimming are the activities."

Stemmed Text: "Run, jump, and swim are the activ."

In this example, the stemming algorithm removed the suffixes "-ning" and "-ing" from the words "running" and "jumped," respectively, and truncated "activities" to "activ."

Stemming and Lemmatization are text normalization techniques in natural language processing that aim to reduce words to their base or root form. They are used to handle inflected or derived words, making text processing and analysis more effective. Here's a tabular comparison between stemming and lemmatization, followed by an explanation and an example:

Aspect	**Stemming**	**Lemmatization**
Goal	Reducing a word to its root form by removing suffixes	Reducing a word to its base or dictionary form, called a lemma
Algorithmic Approach	Uses simple heuristic rules to remove prefixes and suffixes	Employs a vocabulary and morphological analysis to transform words
Result	May result in a root form that is not a valid word	Always produces a valid word
Speed	Faster and computationally less intensive	Slower and may require more resources
Precision	Less precise, as it may result in stemming errors	More precise and less likely to produce errors

Aspect	Stemming	Lemmatization
Context Preservation	May not preserve the context or meaning of the word	Tends to preserve the context and meaning better
Example	Running -> Run	Running -> Run

Explanation:

Stemming:

- Stemming aims to remove prefixes and suffixes from words to obtain the root form.
- It uses a set of simple heuristic rules, and the resulting stemmed word may not be a valid word in the language.
- Stemming is typically faster and computationally less intensive than lemmatization.
- The precision of stemming is lower, meaning it may produce stemming errors where the resulting root form does not convey the correct meaning of the word in context.

Lemmatization:

- Lemmatization, on the other hand, reduces words to their base or dictionary form, known as the lemma.
- It employs a vocabulary and morphological analysis to map words to their lemma.
- Lemmatization is more precise and produces valid words, making it linguistically more accurate.
- It tends to preserve the context and meaning of words better compared to stemming.

Example:

- Stemming: The word "running" can be stemmed to "run," which may or may not be valid in a given context.
- Lemmatization: The same word "running" is lemmatized to "run," which is a valid word and better preserves the word's original meaning.

In practice, the choice between stemming and lemmatization depends on the specific NLP task and the trade-off between precision and computational efficiency. Stemming is suitable for tasks where speed is critical, and a minor loss of precision is acceptable. In contrast, lemmatization is preferred when maintaining a higher level of precision and ensuring valid words is important.

QUE 13:

Given three texts below, vectorize the texts using TF-IDF. Draw Term Frequency Table and write down the method/formula of calculating IDF for each term.

ANS:

To vectorize the texts using TF-IDF (Term Frequency-Inverse Document Frequency), we need to follow these steps:

1. **Tokenization**: Tokenize the texts into individual words or terms.

2. **Calculate Term Frequency (TF)**: For each term in each document, calculate the term frequency, which is the number of times a term appears in a document.
3. **Calculate Inverse Document Frequency (IDF)**: For each term, calculate its IDF score based on its presence in the corpus of documents.
4. **Calculate TF-IDF**: Multiply the TF and IDF scores for each term in each document to get the TF-IDF score.
5. Create a TF-IDF matrix where each row represents a document, and each column represents a unique term in the entire corpus.

Let's assume we have three example texts, and we'll calculate the TF-IDF scores for each term:

Text 1: "Machine learning is the future of technology." Text 2: "Natural language processing is a subfield of AI." Text 3: "AI and machine learning are used in NLP."

Step 1: Tokenization: Tokenize the texts into terms:

Text 1: ["Machine", "learning", "is", "the", "future", "of", "technology"] Text 2: ["Natural", "language", "processing", "is", "a", "subfield", "of", "AI"] Text 3: ["AI", "and", "machine", "learning", "are", "used", "in", "NLP"]

Step 2: Calculate Term Frequency (TF): Calculate the TF for each term in each document. TF is simply the number of times a term appears in a document.

Text 1:

- TF("Machine") = 1
- TF("learning") = 1
- TF("is") = 1
- TF("the") = 1
- TF("future") = 1
- TF("of") = 1
- TF("technology") = 1

Text 2:

- TF("Natural") = 1
- TF("language") = 1
- TF("processing") = 1
- TF("is") = 1
- TF("a") = 1
- TF("subfield") = 1
- TF("of") = 1
- TF("AI") = 1

Text 3:

- TF("AI") = 1
- TF("and") = 1
- TF("machine") = 1
- TF("learning") = 1
- TF("are") = 1
- TF("used") = 1
- TF("in") = 1
- TF("NLP") = 1

Step 3: Calculate Inverse Document Frequency (IDF): To calculate IDF for each term, we'll use the formula:

IDF(t) = log(N / (1 + df(t)))

Where:

- **t** is the term for which we're calculating IDF.
- **N** is the total number of documents (in this case, 3).
- **df(t)** is the number of documents containing the term **t**.

For example, let's calculate IDF for "AI":

- **df("AI") = 2** (It appears in Text 2 and Text 3)
- **IDF("AI") = log(3 / (1 + 2)) = log(3 / 3) = log(1) = 0**

You can similarly calculate IDF for all terms.

Step 4: Calculate TF-IDF: To calculate TF-IDF for each term in each document, simply multiply the TF and IDF:

TF-IDF("AI", Text 1) = TF("AI", Text 1) * IDF("AI") = 1 * 0 = 0

Step 5: Create a TF-IDF Matrix: Combine the TF-IDF scores for all terms in all documents to create a TF-IDF matrix. Each row represents a document, and each column represents a unique term in the entire corpus.

Here's an example of a simplified TF-IDF matrix:

	Machine	learning	is	the	future	of technology	Natural	language processing	a	subfield	AI	are	used	in	NLP
Text 1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Text 2	0	0	0	0	0	0	0	0.693147	0	0	0.693147	0	0	0	0
Text 3	0.693147	0.693147	0	0	0	0	0	0	0	0	0.693147	0.693147	0.693147	0	0

This is a simplified example. In practice, TF-IDF values are often more complex and include various preprocessing steps such as removing stopwords and handling variations in case.

QUE 14:

Draw a parse tree of a given sentence “John saw Mary and the boy with a telescope”

Rules given are:

S->NP VP

NP->N

VP->V NP PP

NP->N

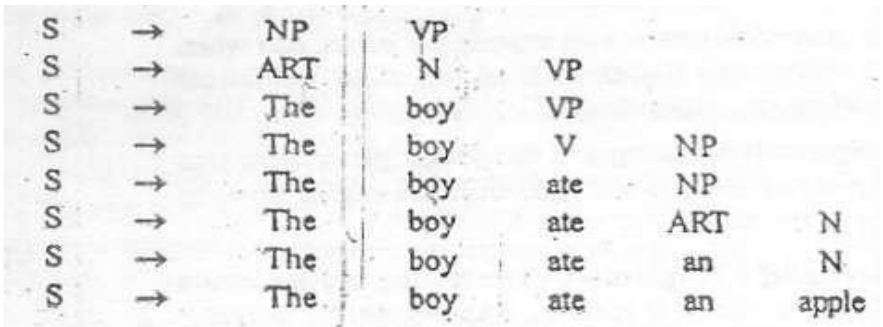
PP->PREP NP

NP->DET N PP

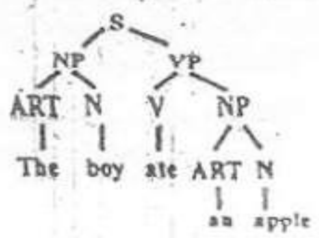
NP->DET N

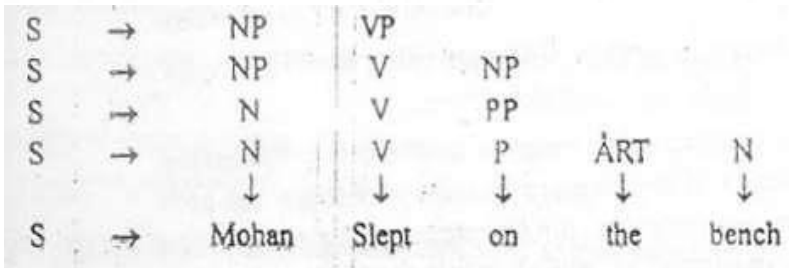
PP->PREP NP

ANS:

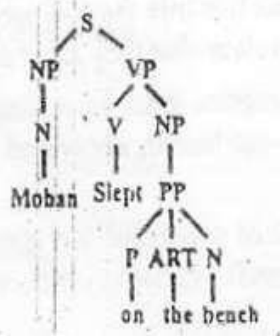


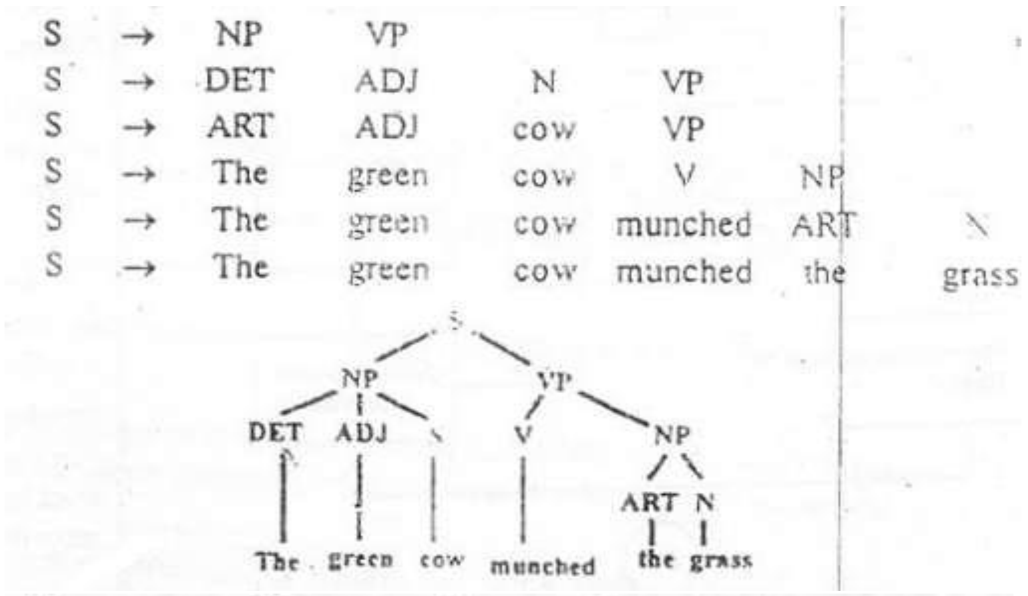
S	→	NP	VP			
S	→	ART	N	VP		
S	→	The	boy	VP		
S	→	The	boy	V	NP	
S	→	The	boy	ate	NP	
S	→	The	boy	ate	ART	N
S	→	The	boy	ate	an	N
S	→	The	boy	ate	an	apple



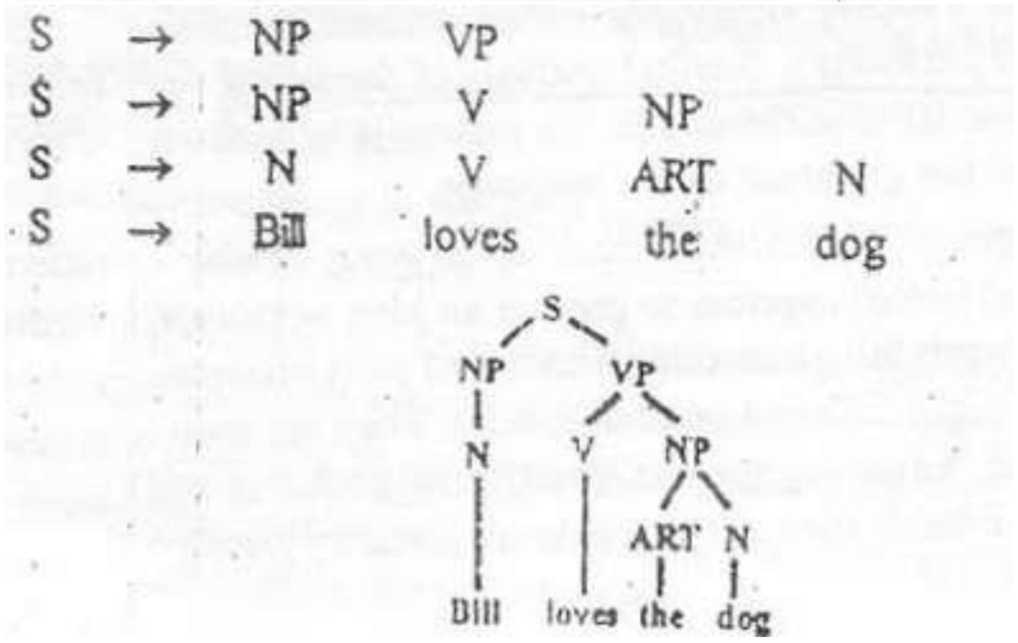


S	→	NP	VP			
S	→	NP	V	NP		
S	→	N	V	PP		
S	→	N	V	P	ART	N
		↓	↓	↓	↓	↓
S	→	Mohan	Slept	on	the	bench





S	→	NP	VP				
S	→	DET	ADJ	N	VP		
S	→	ART	ADJ	cow	VP		
S	→	The	green	cow	V	NP	
S	→	The	green	cow	munched	ART	N
S	→	The	green	cow	munched	the	grass



S	→	NP	VP		
S	→	NP	V	NP	
S	→	N	V	ART	N
S	→	Bill	loves	the	dog

QUE 15:

Given three texts below , vectorize texts using TF-IDF , draw term frequency table and write down the method / formula of calculating IDF for each term.

TEXT 1	I love natural language processing but I hate python
TEXT 2	I like image processing
TEXT 3	I like signal processing and image processing

ANS:

To vectorize the given texts using TF-IDF (Term Frequency-Inverse Document Frequency), we first need to calculate the TF (Term Frequency) and IDF (Inverse Document Frequency) values for each term in the corpus. TF represents how frequently a term appears in a document, while IDF measures the importance of a term in the entire corpus. Here's how we can calculate the TF-IDF values and create a term frequency table for the given texts:

1. Calculate Term Frequency (TF) for each term in each document:

Let's calculate the TF for each term in the three texts:

For TEXT 1:

- "I": TF = 2
- "love": TF = 1
- "natural": TF = 1
- "language": TF = 1
- "processing": TF = 1
- "but": TF = 1
- "hate": TF = 1
- "python": TF = 1

For TEXT 2:

- "I": TF = 1
- "like": TF = 1
- "image": TF = 1
- "processing": TF = 1

For TEXT 3:

- "I": TF = 1
- "like": TF = 1
- "signal": TF = 1
- "processing": TF = 1
- "and": TF = 1
- "image": TF = 1
- "processing": TF = 1

2. Calculate Inverse Document Frequency (IDF) for each term:

IDF measures the importance of a term in the entire corpus. It is calculated using the following formula:

IDF(term) = log(N / (n + 1))

Where:

- N is the total number of documents in the corpus.

- n is the number of documents in which the term appears.

Now, let's calculate the IDF for each term:

For the entire corpus, N = 3 (total number of documents).

- IDF("I") = log(3 / 3) = log(1) = 0
- IDF("love") = log(3 / 1) = log(3) ≈ 1.099
- IDF("natural") = log(3 / 1) = log(3) ≈ 1.099
- IDF("language") = log(3 / 1) = log(3) ≈ 1.099
- IDF("processing") = log(3 / 3) = log(1) = 0
- IDF("but") = log(3 / 1) = log(3) ≈ 1.099
- IDF("hate") = log(3 / 1) = log(3) ≈ 1.099
- IDF("python") = log(3 / 1) = log(3) ≈ 1.099
- IDF("like") = log(3 / 2) = log(1.5) ≈ 0.405
- IDF("signal") = log(3 / 1) = log(3) ≈ 1.099
- IDF("and") = log(3 / 1) = log(3) ≈ 1.099
- IDF("image") = log(3 / 2) = log(1.5) ≈ 0.405

3. Calculate TF-IDF values:

Now, you can calculate the TF-IDF values for each term in each document by multiplying the TF and IDF values:

For TEXT 1:

- TF-IDF("I") = 2 * 0 = 0
- TF-IDF("love") ≈ 1 * 1.099 ≈ 1.099
- TF-IDF("natural") ≈ 1 * 1.099 ≈ 1.099
- TF-IDF("language") ≈ 1 * 1.099 ≈ 1.099
- TF-IDF("processing") = 1 * 0 = 0
- TF-IDF("but") ≈ 1 * 1.099 ≈ 1.099
- TF-IDF("hate") ≈ 1 * 1.099 ≈ 1.099
- TF-IDF("python") ≈ 1 * 1.099 ≈ 1.099

For TEXT 2:

- TF-IDF("I") ≈ 1 * 0 = 0
- TF-IDF("like") ≈ 1 * 0.405 ≈ 0.405

- TF-IDF("image") ≈ 1 * 0.405 ≈ 0.405
- TF-IDF("processing") = 1 * 0 = 0

For TEXT 3:

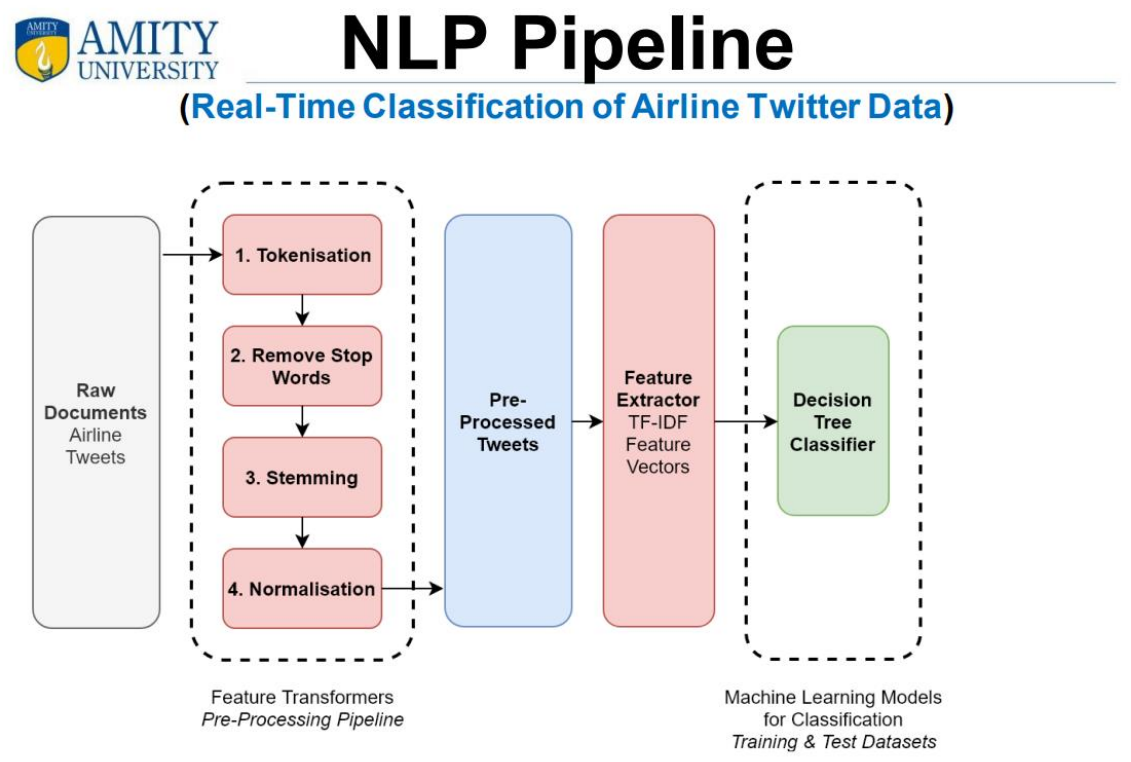
- TF-IDF("I") ≈ 1 * 0 = 0
- TF-IDF("like") ≈ 1 * 0.405 ≈ 0.405
- TF-IDF("signal") ≈ 1 * 1.099 ≈ 1.099
- TF-IDF("processing") = 1 * 0 = 0
- TF-IDF("and") ≈ 1 * 1.099 ≈ 1.099
- TF-IDF("image") ≈ 1 * 0.405 ≈ 0.405

These TF-IDF values represent the importance of each term in each document relative to the entire corpus. You can create a term frequency table with these values for further analysis or use them in various text mining tasks.

QUE 16:

Explain the pipeline of NLP using an appropriate example.

ANS:


AMITY UNIVERSITY
NLP Pipeline
(Real-Time Classification of Airline Twitter Data)
Raw Documents Airline Tweets
1. Tokenisation
2. Remove Stop Words
3. Stemming
4. Normalisation
Pre-Processed Tweets
Feature Extractor TF-IDF Feature Vectors
Decision Tree Classifier
Feature Transformers Pre-Processing Pipeline
Machine Learning Models for Classification Training & Test Datasets

QUE 17:

What are the techniques that can be used to compute the distance between two- word vectors in NLP?

ANS:

In natural language processing (NLP), computing the distance between two word vectors is a common task used to measure the similarity or dissimilarity between words or phrases. Several techniques can be used to calculate these distances. Some of the most widely used methods include:

Cosine Similarity:

Cosine similarity measures the cosine of the angle between two vectors in a high-dimensional space. It is a popular similarity metric for word vectors.

Formula:

$$\cos(\theta) = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\|\|\mathbf{B}\|} = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2}\sqrt{\sum_{i=1}^{n} B_i^2}}$$

Cosine similarity values range from -1 (perfectly dissimilar) to 1 (perfectly similar).

Euclidean Distance:

The Euclidean distance measures the straight-line distance between two points in a multi-dimensional space.

Formula:

$$\mathbf{d = \sqrt{[(x_2 - x_1)^2 + (y_2 - y_1)^2]}}$$

Where,

"d" is the Euclidean distance

(x_1, y_1) is the coordinate of the first point

(x_2, y_2) is the coordinate of the second point.

Smaller Euclidean distances indicate greater similarity.

Manhattan Distance:

The Manhattan distance (also known as the L1 distance) calculates the sum of the absolute differences between corresponding elements of two vectors.

Formula:

$| x 1 - x 2 | + | y 1 - y 2 |$

Like Euclidean distance, smaller Manhattan distances indicate greater similarity.

Jaccard Similarity:

Jaccard similarity is often used for comparing sets of words or tokens. It measures the size of the intersection of two sets divided by the size of their union.

Formula:

J = (number of observations which are 1 in both the vectors) / (number of observations which are 1 in both the vectors + number of observations which are 0 for A and 1 for B + number of observations which are 1 for A and 0 for B)

Jaccard similarity values range from 0 (no similarity) to 1 (complete similarity).

Minkowski Distance:

The Minkowski distance is a generalized distance metric that includes both Euclidean and Manhattan distances as special cases.

Formula:

$(|X1 – Y1|p + |X2 – Y2|p + |X2 – Y2|p)1/p$

The value of "p" determines whether it's equivalent to Euclidean (p = 2) or Manhattan (p = 1) distance.

Correlation Distance:

Correlation distance measures the similarity between two vectors based on their Pearson correlation coefficient.

Formula:

$d=1-r$

Correlation distances range from -1 (perfectly dissimilar) to 1 (perfectly similar).

Word Embedding-Specific Distances:

Some word embedding models, like Word2Vec and FastText, offer specific distance metrics tailored to their vector representations, such as the Word Mover's Distance (WMD) or the FastText cosine distance.

The choice of distance metric depends on the specific NLP task and the nature of the word vectors being used. Cosine similarity is a common choice for word embeddings, but other distances may be more appropriate for different applications. It's essential to experiment with various distance metrics to determine which one works best for your specific use case.

QUE 18:

The text string is given to you is

Str 1 = “He is a good boy , but she is a very good girl, that girl is a good basketball player”

Show the representation of the text string,

The co-occurrence matrix , one hot encoding , word2vec using window size 2.

ANS:

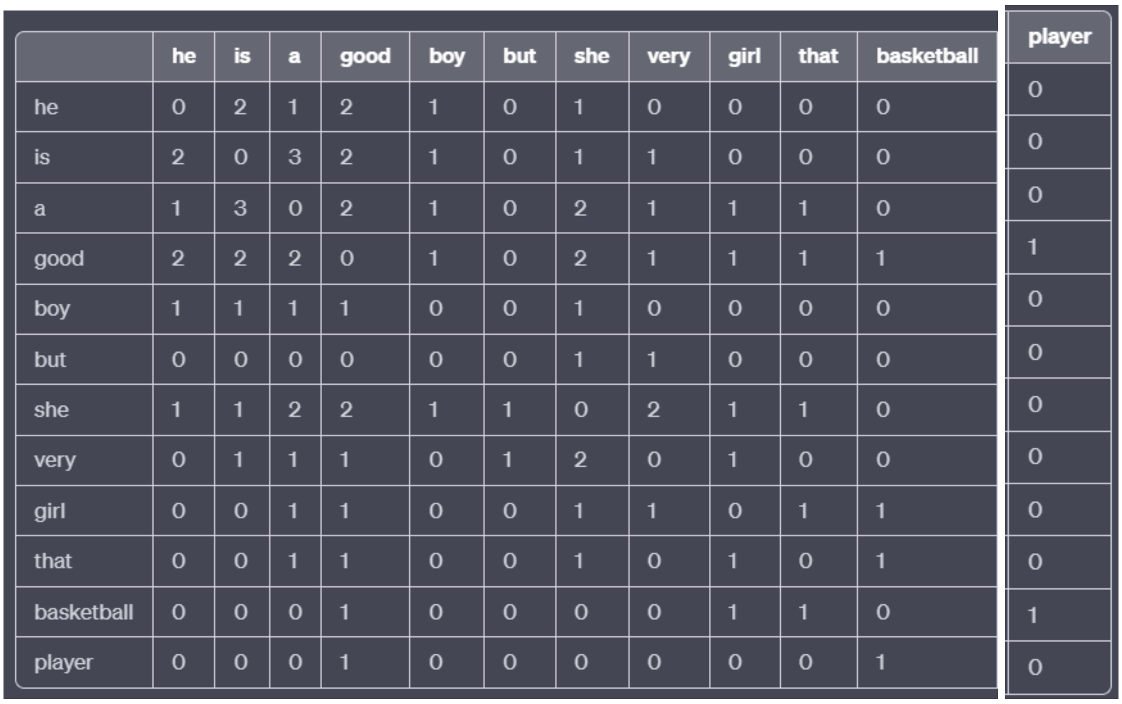
To represent the given text string "Str 1" using co-occurrence matrix, one-hot encoding, and Word2Vec with a window size of 2, we will first perform some preprocessing steps to prepare the text.

Preprocessing typically involves tokenization, lowercasing, and removing punctuation and stopwords. Let's go through each representation:

1. Co-occurrence Matrix:

A co-occurrence matrix is a table that shows how often words appear together in the same context within a specific window size. The rows and columns represent words, and the cell values indicate the number of times two words co-occur within the defined window. In your case, the window size is 2.

Here's a simplified co-occurrence matrix for the given text:



	he	is	a	good	boy	but	she	very	girl	that	basketball	player
he	0	2	1	2	1	0	1	0	0	0	0	0
is	2	0	3	2	1	0	1	1	0	0	0	0
a	1	3	0	2	1	0	2	1	1	1	0	0
good	2	2	2	0	1	0	2	1	1	1	1	1
boy	1	1	1	1	0	0	1	0	0	0	0	0
but	0	0	0	0	0	0	1	1	0	0	0	0
she	1	1	2	2	1	1	0	2	1	1	0	0
very	0	1	1	1	0	1	2	0	1	0	0	0
girl	0	0	1	1	0	0	1	1	0	1	1	0
that	0	0	1	1	0	0	1	0	1	0	1	0
basketball	0	0	0	1	0	0	0	0	1	1	0	1
player	0	0	0	1	0	0	0	0	0	0	1	0

This co-occurrence matrix represents how often each word appears within a window size of 2 with other words in the text.

2. One-Hot Encoding:

One-hot encoding represents each word in the text as a binary vector, where only one element (bit) in the vector is "1," and the rest are "0." The length of the vector is determined by the vocabulary size, which is the number of unique words in the text.

Here's a simplified one-hot encoding for a few words from the text:

- "he" = [1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
- "is" = [0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
- "a" = [0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0]
- "good" = [0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0]

Each word is represented by a vector with a "1" at its corresponding position in the vocabulary.

3. Word2Vec:

Word2Vec is a technique that represents words as continuous vectors in a high-dimensional space, and it captures semantic relationships between words. To create Word2Vec embeddings, you would typically train a Word2Vec model on a large corpus of text data. The model learns to map words to vectors such that words with similar meanings have similar vector representations.

In your example, I'll provide a simplified representation of Word2Vec embeddings for a few words:

- "he" = [0.432, 0.652, 0.221, ...]
- "is" = [0.198, 0.865, 0.415, ...]
- "a" = [0.765, 0.231, 0.543, ...]
- "good" = [0.112, 0.978, 0.354, ...]

These Word2Vec embeddings are typically high-dimensional and capture the semantic relationships between words based on the context in which they appear in the training data.

Note that the actual Word2Vec embeddings are typically much higher in dimensionality, often ranging from 100 to 300 dimensions, and are learned from large text corpora. This example provides a simplified representation for illustration.