**Practical 1: Edit Distance**

### **Main Functions and Key Points to Remember:**

1. **editDistance()** - Recursive function to calculate the minimum number of operations (insert, remove, replace) to convert one string to another.
2. **Base Conditions**:
   * When one string is empty, return the length of the other.
3. **Recursive Calls**:
   * **Insert**: editDistance(str1, str2, m, n-1)
   * **Remove**: editDistance(str1, str2, m-1, n)
   * **Replace**: editDistance(str1, str2, m-1, n-1)

### **Pseudocode:**

1. **If m is 0**, return n (insert all characters).
2. **If n is 0**, return m (remove all characters).
3. **If the last characters are equal**, recurse with m-1, n-1.
4. **Otherwise**:
   * Return 1 + minimum of:
     + **Insert**: recurse with n-1.
     + **Remove**: recurse with m-1.
     + **Replace**: recurse with m-1, n-1.

Code:

| **def** **editDistance**(str1, str2, m, n):  **if** m == 0:  **return** n  **if** n == 0:  **return** m    **if** str1[m-1] == str2[n-1]:  **return** editDistance(str1, str2, m-1, n-1)    **return** 1 + min(  editDistance(str1, str2, m, n-1), # Insert  editDistance(str1, str2, m - 1, n), # Remove  editDistance(str1, str2, m-1, n-1)  ) str1 = "sunday" str2 = "saturday" editDistance(str1, str2) |
| --- |

**Practical 1B: Write a program to implement Part of Speech Tagging**

### **Main Functions and Key Points to Remember:**

1. **word\_tokenize()** - Tokenizes the input text into words (from nltk.tokenize).
2. **pos\_tag()** - Performs part-of-speech tagging on the tokenized words (from nltk).
3. **Function pos\_tagging()** - Combines tokenization and POS tagging for a given text.

### **Pseudocode:**

1. **Tokenize text** using word\_tokenize().
2. **Apply POS tagging** to the tokenized words using pos\_tag().
3. **Return POS tags** as a list of (word, tag) pairs.
4. **Print results**: Original text and corresponding POS tags.

Code:

| **from** nltk.tokenize **import** word\_tokenize **from** nltk **import** pos\_tag  **def** **pos\_tagging**(text):  words = word\_tokenize(text)  pos\_tags = pos\_tag(words)   **return** pos\_tags   text = "I love programming"  print(f"Orginal Text: {text}")  tags\_arr = pos\_tagging(text) **for** word, tag **in** tags\_arr:  print(f"{word, tag}") |
| --- |

**Practical 2: Write a program to implement sentence segmentation and word tokenization**

### **Main Functions and Key Points to Remember:**

1. **sent\_tokenize()** - Tokenizes the text into sentences (from nltk.tokenize).
2. **word\_tokenize()** - Tokenizes the text into words (from nltk.tokenize).

### **Pseudocode:**

1. **Sentence Segmentation**:
   * Use sent\_tokenize() to split the input text into sentences.
   * Loop through the sentence tokens and print each sentence.
2. **Word Tokenization**:
   * Use word\_tokenize() to split the input text into words.
   * Loop through the word tokens and print each word.

Code:

| # Sentence segmentation **from** nltk.tokenize **import** sent\_tokenize  sample = "Hi there. I am hrisabh!. I like to code" sent\_tokens = sent\_tokenize(sample)  **for** s **in** sent\_tokens:  print(s)  # Word tokenization **from** nltk.tokenize **import** word\_tokenize  sample = "Hrisabh, Abhijeet, Anurag" tokens = word\_tokenize(sample) **for** t **in** tokens:  print(t) |
| --- |

**Practical 3: Write a program to Implement stemming and lemmatization**

### **Main Functions and Key Points to Remember:**

1. **LancasterStemmer()** - Reduces words to their root form using the Lancaster Stemming algorithm (from nltk.stem).
2. **PorterStemmer()** - Applies the Porter Stemming algorithm to words (from nltk.stem).
3. **WordNetLemmatizer()** - Lemmatizes words, converting them to their base form using WordNet (from nltk.stem).

### **Pseudocode:**

1. **Lancaster Stemmer**:
   * Initialize LancasterStemmer().
   * Loop through the word list and apply stem() to each word.
   * Print original word and its stemmed version.
2. **Porter Stemmer**:
   * Initialize PorterStemmer().
   * Loop through the word list and apply stem() to each word.
   * Print original word and its stemmed version.
3. **WordNet Lemmatizer**:
   * Initialize WordNetLemmatizer().
   * Loop through the word list, convert words to lowercase, and apply lemmatize() with verb context.
   * Print the original word and its lemmatized version.

| **from** nltk.stem **import** LancasterStemmer, PorterStemmer, WordNetLemmatizer  word\_list = ["Programming", "Running", "Eating", "Sleeping"]  **def** **lan\_stemmer**(str\_list):  stemmer = LancasterStemmer()  print("LancasterStemmer")  **for** w **in** str\_list:  root = stemmer.stem(w)  print(f"{w} after stemming: {root}")  print()  **def** **por\_stemmer**(str\_list):  stemmer = PorterStemmer()  print("PorterStemmer")  **for** w **in** str\_list:  root = stemmer.stem(w)  print(f"{w} after stemming: {root}")  print()  **def** **wn\_lemma**(str\_list):  lemma\_ins = WordNetLemmatizer()  print("WordNetLemmatizer")  **for** w **in** str\_list:  root = lemma\_ins.lemmatize(w.lower(), "v")  print(f"{w} after lemmatizing: {root}")  lan\_stemmer(word\_list) por\_stemmer(word\_list) wn\_lemma(word\_list) |
| --- |

**Practical 4: Write a program to Implement Text Summarization for the given sample text**

### **Main Functions and Key Points to Remember:**

1. **stopwords.words("english")** - To remove common English stopwords from the text (from nltk.corpus).
2. **word\_tokenize(text)** - Tokenizes text into individual words (from nltk.tokenize).
3. **sent\_tokenize(text)** - Tokenizes text into individual sentences (from nltk.tokenize).
4. **defaultdict(int)** - Dictionary that defaults to 0 for counting frequencies (from collections).
5. **Text Summarization Approach**:
   * Calculate word frequencies excluding stopwords.
   * Calculate sentence scores based on word frequencies.
   * Create a summary by selecting sentences with higher-than-average scores.

### **Pseudocode:**

1. **Initialize Stopwords**: Load a set of stopwords for filtering common words.
2. **Tokenize Words**: Convert the input text into a list of words.
3. **Frequency Count**:
   * Create a frequency map for words that are not in the stopwords list.
4. **Tokenize Sentences**: Break the text into individual sentences.
5. **Sentence Scoring**:
   * For each sentence, sum the frequencies of words it contains.
6. **Calculate Average Score**: Compute the average sentence score.
7. **Generate Summary**:
   * Include sentences with scores greater than 1.2 times the average score.

Code:

| **from** collections **import** defaultdict **from** nltk.corpus **import** stopwords **from** nltk.tokenize **import** sent\_tokenize, word\_tokenize  example\_text = """ Deep learning (also known as deep structured learning) is part of a broader family of machine learning methods based on artificial neural networks with representation learning. Learning can be supervised, semi-supervised or unsupervised. Deep-learning architectures such as deep neural networks, deep belief networks, deep reinforcement learning, recurrent neural networks and convolutional neural networks have been applied to fields including computer vision, speech recognition, natural language processing, machine translation, bioinformatics, drug design, medical image analysis, material inspection and board game programs, where they have produced results comparable to and in some cases surpassing human expert performance. Artificial neural networks (ANNs) were inspired by information processing and distributed communication nodes in biological systems. ANNs have various differences from biological brains. Specifically, neural networks tend to be static and symbolic, while the biological brain of most living organisms is dynamic (plastic) and analogue. The adjective "deep" in deep learning refers to the use of multiple layers in the network. Early work showed that a linear perceptron cannot be a universal classifier, but that a network with a nonpolynomial activation function with one hidden layer of unbounded width can. Deep learning is a modern variation which is concerned with an unbounded number of layers of bounded size, which permits practical application and optimized implementation, while retaining theoretical universality under mild conditions. In deep learning the layers are also permitted to be heterogeneous and to deviate widely from biologically informed connectionist models, for the sake of efficiency, trainability and understandability, whence the structured part."""  stopwrd\_set = set(stopwords.words("english"))  words = word\_tokenize(example\_text) freqMap = defaultdict(int)  **def** **count\_freq**(word\_list):  **for** w **in** word\_list:  lowered = w.lower()  **if** (lowered **not** **in** stopwrd\_set):  freqMap[lowered] += 1  count\_freq(words)  print("Frequency of words") print(list(freqMap.items())[:5], "....")  sentences = sent\_tokenize(example\_text) sentMap = defaultdict(int)  **for** s **in** sentences:  **for** word, freq **in** freqMap.items():  **if** word **in** s.lower():  sentMap[s] += freq  sumVal = 0 **for** s **in** sentMap:  sumVal += sentMap[s]  avg = int(sumVal / len(sentMap)) print(avg)  summary = "" **for** s **in** sentences:  **if** s **in** sentMap **and** (sentMap[s] > (1.2 \* avg)):  summary += " " + s print(summary) |
| --- |

**Practical 5a: Write a program to Implement a n-gram model**

### **Main Functions and Key Points to Remember:**

1. **ngrams(sent.split(), n)** - Generates n-grams from a sentence (from nltk.util).
   * n = 1 for unigrams
   * n = 2 for bigrams
   * n = 3 for trigrams
2. **everygrams(sent.split())** - Generates all possible n-grams from the sentence (from nltk.util).

### **Pseudocode:**

1. **Input Sentence**: Split the sentence into words.
2. **Generate N-grams**:
   * Generate unigrams (n = 1).
   * Generate bigrams (n = 2).
   * Generate trigrams (n = 3).
3. **Generate Everygrams**:
   * Generate every possible n-gram (n = 1 to len(sentence)).
4. **Output**:
   * Print the unigrams, bigrams, trigrams, and everygrams.

Code:

| **from** nltk.util **import** ngrams, everygrams  n = 1 sent = "This is first sample sentence for practical 5" uni\_grams = ngrams(sent.split(), n)  n = 2 bi\_grams = ngrams(sent.split(), n)  n = 3 tri\_grams = ngrams(sent.split(), n)  ery\_gram = everygrams(sent.split())  print("Unigrams: ") print(list(uni\_grams)) print()  print("Bigrams: ") print(list(bi\_grams)) print()  print("Trigrams: ") print(list(tri\_grams)) print()  print("Everygrams: ") print(list(ery\_gram)) print() |
| --- |

### **Practical 5b: Write a program to Implement Tri-gram model to predict next word probability**

### **Main Functions and Key Points to Remember:**

1. **nltk.download('reuters')**: Downloads the Reuters corpus.
2. **trigrams (from nltk)**: Creates trigrams from sentences.
   * **pad\_right=True and pad\_left=True**: Pads the sentence with None values on both sides.
3. **defaultdict(lambda: defaultdict(lambda: 0))**: A nested dictionary that initializes missing values to zero.
4. **Trigram Model Creation**:
   * **model[(w1, w2)][w3]**: Counts occurrences of (w1, w2) -> w3 trigram.
5. **Normalization**: Convert counts to probabilities.
6. **Sorting**: Sort predicted next word probabilities for the bigram ("the", "news").

### **Pseudocode:**

1. **Download Reuters Corpus**.
2. **Initialize Trigram Model**:
   * Use defaultdict to store trigram counts.
3. **Count Trigrams**:
   * For each sentence in the corpus, generate trigrams (w1, w2, w3).
   * Increment count of w3 for bigram (w1, w2).
4. **Convert Counts to Probabilities**:
   * For each bigram (w1, w2):
     + Calculate total occurrences.
     + Normalize counts of w3 to probabilities.
5. **Predict Next Word**:
   * For bigram ("the", "news"), sort possible next words by their probabilities.
6. **Output**: Print sorted results of next word probabilities.

Code:

| **import** nltk  nltk.download('reuters') nltk.download('punkt')  **from** nltk.corpus **import** reuters **from** nltk **import** trigrams **from** collections **import** defaultdict  model = defaultdict(**lambda**: defaultdict(**lambda**: 0))  **for** sentence **in** reuters.sents():  **for** w1, w2, w3 **in** trigrams(sentence, pad\_right=**True**, pad\_left=**True**):  model[(w1, w2)][w3] += 1  **for** w1\_w2 **in** model:  total\_count = float(sum(model[w1\_w2].values()))  **for** w3 **in** model[w1\_w2]:  model[w1\_w2][w3] /= total\_count  res = sorted(dict(model["the", "news"]).items(), key=**lambda** x: -1 \* x[1]) print(res) |
| --- |

**Practical 6: Write a program to Implement Named Entity Recognition (NER)**

### **Main Functions and Key Points to Remember:**

1. **spacy.load("en\_core\_web\_sm")**: Loads the pre-trained English NLP model.
2. **nlp(content)**: Processes the text to extract entities.
3. **doc.ents**: Accesses recognized entities from the processed document.
4. **displacy.render(doc, style="ent")**: Visualizes named entities.
5. **pandas.DataFrame**: Creates a DataFrame from the extracted entity information.

### **Pseudocode:**

1. **Load Pre-trained Model**:
   * nlp = spacy.load("en\_core\_web\_sm")
2. **Process Text**:
   * doc = nlp(content)
3. **Print Named Entities**:
   * For each entity in doc.ents:
     + Print entity text, start and end character positions, and label.
4. **Visualize Entities**:
   * Use displacy.render(doc, style="ent") to render entity visualization.
5. **Create DataFrame**:
   * Extract entity text, type, and lemma.
   * Create and print DataFrame using pandas.

Code:

| **import** spacy nlp = spacy.load("en\_core\_web\_sm")  content = "The quick brown fox jumps over the lazy dog. The United Nations is a global organization. The Eiffel Tower is a famous landmark in Paris. Tesla is an electric car company. The Amazon rainforest is the largest rainforest in the world."  doc = nlp(content)  **for** ent **in** doc.ents:  print(ent.text, ent.start\_char, ent.end\_char, ent.label\_)  # Visualize **from** spacy **import** displacy displacy.render(doc, style="ent")  **import** pandas **as** pd entities = [(ent.text, ent.label\_, ent.lemma\_) **for** ent **in** doc.ents] df = pd.DataFrame(entities, columns=["text", "type", "lemma"]) print(df) |
| --- |

**Practical 8: Write a program to Implement syntactic parsing of a given text**

### **Main Functions and Key Points to Remember:**

1. **CFG.fromstring()**: Defines a context-free grammar (CFG) from a string.
2. **ChartParser(grammar)**: Creates a parser using the defined grammar.
3. **parser.parse(sentence)**: Parses a sentence according to the grammar.
4. **tree.pretty\_print()**: Displays the parse tree in a readable format.

### **Pseudocode:**

1. **Define Grammar**:
   * Use CFG.fromstring() to create grammar rules.
2. **Create Parser**:
   * Initialize ChartParser with the defined grammar.
3. **Prepare Sentence**:
   * Tokenize the example sentence.
4. **Parse Sentence**:
   * For each parse tree generated by parser.parse(sentence):
     + Print the tree.
     + Print the tree in a pretty format using tree.pretty\_print().

Code:

| **import nltk**  **from** nltk **import** CFG **from** nltk.parse.chart **import** ChartParser  # Define a simple context-free grammar grammar = CFG.fromstring(""" S -> NP VP NP -> Det N | Det N PP PP -> P NP VP -> V NP | V NP PP Det -> 'the' | 'a' N -> 'dog' | 'cat' | 'park' | 'telescope' V -> 'saw' | 'ate' P -> 'in' | 'on' | 'with' """)  # Create a parser with the defined grammar parser = ChartParser(grammar)  # Example sentence sentence = "the dog saw the cat in the park".split()  # Parse the sentence **for** tree **in** parser.parse(sentence):  print(tree)  tree.pretty\_print() |
| --- |

**Practical 9: Write a program to Implement dependency parsing of a given text**

### **Main Functions and Modules to Remember:**

1. **spacy.load()** - Loads the language model (from spacy module).
2. **nlp()** - Processes the sentence using the loaded model.
3. **displacy.serve()** - Visualizes the dependency tree (from spacy.displacy).

### **Pseudocode:**

1. **Import** necessary SpaCy modules.
2. **Load pre-trained English model** using spacy.load().
3. **Parse sentence** using nlp().
4. **Print syntactic dependency** of each token in the parsed sentence.
5. **Visualize parse tree** using displacy.serve().

Code:

| **import** spacy **from** spacy **import** displacy  # Load the pre-trained English model nlp = spacy.load('en\_core\_web\_sm')  # Example sentence sentence = "The dog saw the cat in the park."  # Parse the sentence doc = nlp(sentence)  # Print the syntactic dependency information **for** token **in** doc:  print(f'{token.text: <12} {token.dep\_: <10} {token.head.text}')  # Visualize the parse tree using displacy displacy.serve(doc, style='dep') |
| --- |

Practical 10:

**Practical 11:Consider a scenario of Online Review and demonstrate the concept of sentiment analysis and emotion mining by applying various approaches like lexicon-based approach and rule-based approaches.**

### **Main Functions and Key Points to Remember:**

1. **VADER Sentiment Analysis**:
   * Utilizes the VADER (Valence Aware Dictionary and sEntiment Reasoner) library to analyze the sentiment of a given review text.
   * The polarity\_scores() method provides a sentiment score that includes positive, negative, neutral, and compound scores.
2. **Rule-Based Sentiment Analysis**:
   * A custom function (rule\_based\_sentiment()) analyzes sentiment based on predefined lists of positive and negative words. It assigns a score based on word presence and returns "Positive," "Negative," or "Neutral."
3. **Emotion Extraction**:
   * Uses a lexicon-based approach to extract emotions from the review text. It maps specific words to their corresponding emotions defined in the emotion\_lexicon dictionary.
4. **Detecting Emotions**:
   * A simple function (detect\_emotion()) detects emotions based on the presence of specific keywords, returning an emotion type.

### **Pseudocode:**

1. **VADER Sentiment Analysis**:
   * Initialize VADER sentiment analyzer.
   * Analyze sentiment using polarity\_scores().
2. **Rule-Based Sentiment Analysis**:
   * Define positive and negative words.
   * Create a function that calculates a score based on word presence and returns sentiment label.
3. **Emotion Extraction**:
   * Define an emotion lexicon mapping words to emotions.
   * Create a function that checks for words in the lexicon and returns corresponding emotions.
4. **Detect Emotions**:
   * Create a function that detects emotions based on specific keywords.

Code:

| #pip install vaderSentiment **from** vaderSentiment.vaderSentiment **import** SentimentIntensityAnalyzer  # Initialize VADER sentiment analyzer analyzer = SentimentIntensityAnalyzer()  # Review text review\_text = "Had a wonderful experience at the new restaurant. The food was absolutely delicious, and the service was fantastic. However, the ambiance could have been better, and the prices were a bit high."  # Analyze sentiment sentiment = analyzer.polarity\_scores(review\_text) print("Sentiment Analysis using VADER:") print(sentiment)  **def** **rule\_based\_sentiment**(text):  positive\_words = ['wonderful', 'delicious', 'fantastic']  negative\_words = ['high', 'worse', 'bad']   score = 0  **for** word **in** positive\_words:  **if** word **in** text:  score += 1  **for** word **in** negative\_words:  **if** word **in** text:  score -= 1  **if** score > 0:  **return** 'Positive'  **elif** score < 0:  **return** 'Negative'  **else**:  **return** 'Neutral'  # Analyze sentiment sentiment = rule\_based\_sentiment(review\_text) print("Sentiment Analysis using Rule-Based Approach:") print(sentiment)  emotion\_lexicon = {  'happy': 'joy',  'wonderful': 'joy',  'delicious': 'joy',  'fantastic': 'joy',  'high': 'anger',  'worse': 'sadness',  'bad': 'sadness' }  **def** **extract\_emotions**(text):  words = text.lower().split()  emotions = []  **for** word **in** words:  **if** word **in** emotion\_lexicon:  emotions.append(emotion\_lexicon[word])  **return** emotions  # Extract emotions emotions = extract\_emotions(review\_text) print("Emotion Mining using Lexicon-Based Approach:") print(emotions)  **def** **detect\_emotion**(text):  **if** "wonderful" **in** text **or** "delicious" **in** text **or** "fantastic" **in** text:  **return** 'joy'  **elif** "high" **in** text:  **return** 'anger'  **else**:  **return** 'neutral'  # Detect emotions emotion = detect\_emotion(review\_text) print("Emotion Mining using Rule-Based Approach:") print(emotion) |
| --- |

**Practical 12: Write a program to Implement Skip-gram.**

### **Main Functions and Key Points to Remember:**

1. **simple\_preprocess()**: Tokenizes and cleans the text.
2. **Word2Vec(sentences, vector\_size, window, sg, min\_count)**: Initializes and trains the Word2Vec model.
   * **sentences**: Tokenized and cleaned text.
   * **vector\_size**: Dimensionality of word vectors.
   * **window**: Context window size.
   * **sg**: 1 for Skip-gram, 0 for CBOW.
   * **min\_count**: Minimum frequency of words to consider.
3. **model.save(filename)**: Saves the trained model.
4. **Word2Vec.load(filename)**: Loads the saved model.
5. **model.wv[word]**: Retrieves the vector for a specific word.
6. **model.wv.most\_similar(word)**: Finds similar words to the given word.

### **Pseudocode:**

1. **Preprocess Text**:
   * Use simple\_preprocess() to tokenize and clean the corpus.
2. **Create and Train Model**:
   * Initialize Word2Vec with:
     + sentences: Tokenized corpus.
     + vector\_size: Dimensionality.
     + window: Context size.
     + sg: Skip-gram (1).
     + min\_count: Minimum word frequency.
   * Train the model.
3. **Save and Load Model**:
   * Save with model.save().
   * Load with Word2Vec.load().
4. **Retrieve Word Vector**:
   * Use model.wv[word] to get the vector for a word.
5. **Find Similar Words**:
   * Use model.wv.most\_similar(word) to find and print similar words.

Code:

| **from** gensim.models **import** Word2Vec **from** gensim.utils **import** simple\_preprocess  # Sample text corpus corpus = [  "She is a great dancer.",  "He is a wonderful musician.",  "They are excellent at their craft." ]  # Preprocess the text: Tokenize and clean processed\_corpus = [simple\_preprocess(sentence) **for** sentence **in** corpus]  # Create and train the Skip-gram model model = Word2Vec(sentences=processed\_corpus, vector\_size=100, window=2, sg=1, min\_count=1)  # Save and load the model model.save("skipgram\_model.bin") model = Word2Vec.load("skipgram\_model.bin")  # Get vector for a word word\_vector = model.wv['great'] print(f"Vector for 'great': {word\_vector}")  # Find similar words similar\_words = model.wv.most\_similar('great') print("Most similar words to 'great':") **for** word, similarity **in** similar\_words:  print(f"{word}: {similarity:.4f}") |
| --- |

**Practical 13: Write a program to Implement SMS Fraud Detection.**

### **Main Functions and Key Points to Remember:**

1. **Data Handling**:
   * **pd.DataFrame(data)**: Creates a DataFrame from the provided data.
2. **Text Processing and Model Training**:
   * **train\_test\_split(X, y, test\_size, random\_state)**: Splits data into training and test sets.
   * **make\_pipeline(TfidfVectorizer(), MultinomialNB())**: Creates a pipeline with TF-IDF vectorizer and Naive Bayes classifier.
   * **model.fit(X\_train, y\_train)**: Trains the model.
3. **Prediction**:
   * **model.predict(new\_sms)**: Predicts labels for new SMS messages.
4. **Sentiment Analysis**:
   * **SentimentIntensityAnalyzer()**: Initializes the sentiment analyzer.
   * **analyzer.polarity\_scores(text)**: Analyzes the sentiment of the given text.

### **Pseudocode:**

1. **Prepare Data**:
   * Create a DataFrame from SMS data.
2. **Train Model**:
   * Split data into training and test sets with train\_test\_split().
   * Create a pipeline with TfidfVectorizer and MultinomialNB.
   * Train with model.fit().
3. **Predict**:
   * Use model.predict() on new SMS messages to get predicted labels.
4. **Perform Sentiment Analysis**:
   * Initialize SentimentIntensityAnalyzer.
   * Analyze sentiment using analyzer.polarity\_scores().

Code:

| **import** pandas **as** pd **from** sklearn.feature\_extraction.text **import** TfidfVectorizer **from** sklearn.model\_selection **import** train\_test\_split **from** sklearn.naive\_bayes **import** MultinomialNB **from** sklearn.pipeline **import** make\_pipeline **from** vaderSentiment.vaderSentiment **import** SentimentIntensityAnalyzer  # Sample SMS data data = {  'sms\_text': [  "Your account has been credited with $560",  "Call us immediately! Your account is at risk",  "Your loan application has been approved",  "Win a $1000 gift card! Click here",  "Reminder: Your payment is due tomorrow"  ],  'label': ['transaction', 'fraud\_alert', 'transaction', 'promotion', 'reminder'] }  df = pd.DataFrame(data)  # Text processing and model training X\_train, X\_test, y\_train, y\_test = train\_test\_split(df['sms\_text'], df['label'], test\_size=0.2, random\_state=42) model = make\_pipeline(TfidfVectorizer(), MultinomialNB()) model.fit(X\_train, y\_train)  # Predict on new SMS new\_sms = ["Congratulations! You've won $1000. Claim now"] predicted\_label = model.predict(new\_sms) print(f"Predicted Label: {predicted\_label[0]}")  # Sentiment Analysis analyzer = SentimentIntensityAnalyzer() sentiment = analyzer.polarity\_scores(new\_sms[0]) print(f"Sentiment Analysis: {sentiment}") |
| --- |