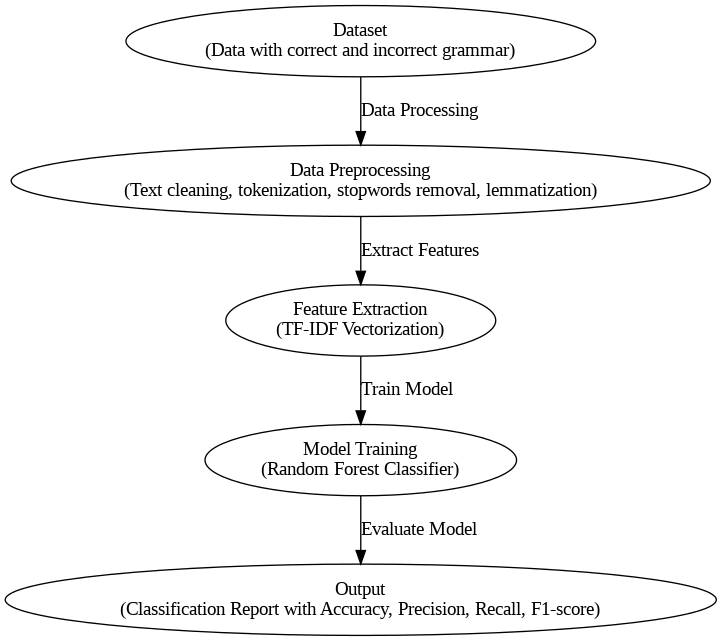
Grammatical Error Detection Using Natural Language Processing  
By: - Madhaesh P(126156079), Shravwan R(126156138), Janaranjan A (126156058)

# AIM:

To develop a Natural Language Processing (NLP) model that detects grammatical errors in text data.

# WORKFLOW:

  
Flowchart/Diagram:   
  
  
- Dataset: Collection of text data containing correct and incorrect grammar.  
- Method: Data preprocessing, tokenization, text cleaning, feature extraction using TF-IDF, model training with Best Model as RandomForestClassifier.  
- Output: Classification report showcasing the model's accuracy in detecting grammatical errors.

Explanation:  
  
**1. Dataset:** The dataset is preprocessed by cleaning the text, tokenizing, and removing stop words.  
**2. Method:**  
 **1. Data Exploration and Preprocessing**

* **Missing Values**: Check for missing or null values in your dataset. Missing values can lead to issues during model training, so consider either filling them in or removing them.
* **Text Normalization**: Normalize the text by converting it to lowercase, removing unnecessary punctuation, and handling special characters.
* **Tokenization**: Break down sentences into individual tokens (words). You can use libraries like nltk, spaCy, or transformers for this.
* **Stopword Removal**: Remove common words (like "and", "the", etc.) that may not contribute to grammatical error detection. However, be cautious as some stopwords may be necessary for grammatical context.
* **Lemmatization/Stemming**: Reduce words to their base or root form to standardize the text and reduce the vocabulary size.

**2. Feature Engineering**

* **POS Tagging**: Use Part-of-Speech (POS) tagging to understand the grammatical structure of sentences. This can help identify patterns of errors.
* **N-grams**: Consider using bi-grams or tri-grams to capture contextual information and common error patterns.
* **Error Labels**: If your dataset has both correct and incorrect sentences, ensure that labels are properly assigned to each sentence or token.

**3. Model Selection**

* **Baseline Model**: Start with a simple model like Logistic Regression or Naive Bayes to establish a baseline performance.

**4. Model Training and Evaluation**

* **Cross-Validation**: Use cross-validation to ensure that your model generalizes well to unseen data.
* **Evaluation Metrics**: Choose appropriate metrics like F1-score, Precision, Recall, and Accuracy to evaluate your model. Since grammatical error detection may involve imbalanced classes, F1-score might be more informative.

**3. Output:** The model is trained and evaluated using metrics such as accuracy and classification report.

# CODE:

**Data Preprocessing:-**

import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

import seaborn as sns

import nltk

import re

from nltk.corpus import stopwords

from nltk.tokenize import word\_tokenize

from nltk.stem import WordNetLemmatizer

from sklearn.model\_selection import train\_test\_split

from sklearn.feature\_extraction.text import TfidfVectorizer,CountVectorizer

from sklearn.naive\_bayes import MultinomialNB

from sklearn.metrics import accuracy\_score, classification\_report

from nltk import pos\_tag

from nltk.util import ngrams

from collections import Counter

nltk.download('punkt')

nltk.download('stopwords')

nltk.download('wordnet')

nltk.download('averaged\_perceptron\_tagger')

df=pd.read\_csv('/content/All\_train\_data.csv')

df.head()

#Checking for missing Values

print("\n Missing values in each column:")

print(df.isnull().sum())

#Basic Statistics

print("\n Basic Statistics:")

print(df.describe())

if 'labels' in df.columns:

    print("\n Class Distribution:")

    sns.countplot(x='labels', data=df)

    plt.title('Distribution of the target variable')

    plt.show()

def clean\_text(text):

  #Convert to lowercase

  text=text.lower()

  #remove punctuation and special characters

  text=re.sub(r'[^\w\s]','',text)

  #remove numbers

  text=re.sub(r'\d+','',text)

  #remove extra spaces

  text=re.sub(r'\s+',' ',text).strip()

  return text

#Text Cleaning

df['cleaned\_text']= df['input'].apply(clean\_text)

#Tokenization

df['tokens']=df['cleaned\_text'].apply(word\_tokenize)

#remove stopwords

stop\_words=set(stopwords.words('english'))

df['tokens']=df['tokens'].apply(lambda x: [word for word in x if word not in stop\_words])

#Lemmatization

lemmatizer=WordNetLemmatizer()

df['tokens']=df['tokens'].apply(lambda x: [lemmatizer.lemmatize(word) for word in x])

#Checking the processed data

print("\n Processed Data:")

print(df[['input','cleaned\_text','tokens']].head())

if 'labels' in df.columns:

    X = df['cleaned\_text']

    y = df['labels']

    X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

    print("\nTraining and testing data split complete:")

    print(f"Training set size: {len(X\_train)}")

    print(f"Testing set size: {len(X\_test)}")

df.to\_csv('preprocessed\_data.csv', index=False)

**Feature Engineering:-**

df1=pd.read\_csv('/content/preprocessed\_data.csv')

df1['tokens']=df1['tokens'].apply(eval)

#Part of Speech tagging

def pos\_tagging(tokens):

    return nltk.pos\_tag(tokens)

df1['pos\_tags'] = df1['tokens'].apply(pos\_tagging)

print("\nPOS tagging complete.")

print(df1[['tokens', 'pos\_tags']].head())

*#N-grams*

def generate\_ngrams(tokens, n=2):

    return list(ngrams(tokens, n))

*#Generate Bigrams*

df1['bigrams'] = df1['tokens'].apply(lambda x: generate\_ngrams(x, n=2))

*#Generate Trigrams*

df1['trigrams'] = df1['tokens'].apply(lambda x: generate\_ngrams(x, n=3))

print("\nN-grams generation complete.")

print(df1[['tokens', 'bigrams', 'trigrams']].head())

*#Term Frequency- Inverse Document Frequency*

df1['cleaned\_text']=df1['tokens'].apply(lambda x: ' '.join(x))

*#intialize TF-IDF*

tfidf\_vectorizer=TfidfVectorizer(max\_features=5000)

*# Fit and Transform the cleaned text*

tfidf\_matrix=tfidf\_vectorizer.fit\_transform(df1['cleaned\_text'])

*#Convert the TF-IDF matrix to a DataFrame*

tfidf\_df=pd.DataFrame(tfidf\_matrix.toarray(), columns=tfidf\_vectorizer.get\_feature\_names\_out())

print("\nTF-IDF feature generation complete.")

print(tfidf\_df.head())

*#Error Density (ratio of Incorrect words)*

def error\_density(row):

  error\_count=sum([1 for tokens in row['tokens'] if token in row['errors']])

df1.to\_csv('feature\_engineered\_data.csv', index=False)

print("\nFeature engineering complete. Dataset saved as 'feature\_engineered\_data.csv'.")

return error\_count/len(row['tokens']) if len(row['tokens'])>0 else 0

**Model Selection:-**

import pandas as pd

from sklearn.model\_selection import train\_test\_split, GridSearchCV

from sklearn.metrics import accuracy\_score, precision\_score, recall\_score, f1\_score,classification\_report

from sklearn.linear\_model import LogisticRegression

from sklearn.ensemble import RandomForestClassifier

from sklearn.svm import SVC

from sklearn.naive\_bayes import MultinomialNB

X= df2.drop(columns=['labels'])

y=df2['labels']

from sklearn.feature\_extraction.text import TfidfVectorizer

vectorizer = TfidfVectorizer()

X = vectorizer.fit\_transform(X['input'])

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

models = {

    'Logistic Regression': LogisticRegression(),

    'Random Forest': RandomForestClassifier(),

    'Support Vector Machine': SVC(),

    'Naive Bayes': MultinomialNB()

}

def evaluate\_model(model, X\_train, y\_train, X\_test, y\_test):

  model.fit(X\_train, y\_train)

  y\_pred = model.predict(X\_test)

  accuracy = accuracy\_score(y\_test, y\_pred)

  precision = precision\_score(y\_test, y\_pred, average='weighted')

  recall = recall\_score(y\_test, y\_pred, average='weighted')

  f1 = f1\_score(y\_test, y\_pred, average='weighted')

  return accuracy, precision, recall, f1

for model\_name, model in models.items():

  accuracy , precision, recall, f1 = evaluate\_model(model, X\_train, y\_train, X\_test, y\_test)

  print(f"{model\_name} Results:")

  print(f"Accuracy: {accuracy:.4f}")

  print(f"Precision: {precision:.4f}")

  print(f"Recall: {recall:.4f}")

  print(f"F1 Score: {f1:.4f}\n")

from sklearn.model\_selection import GridSearchCV

*# Define the parameter grid for GridSearchCV*

param\_grid = {

    'n\_estimators': [50, 100],

    'max\_depth': [None, 10, 20],

    'min\_samples\_split': [2, 5]

}

*# Initialize GridSearchCV with the RandomForestClassifier*

grid\_search = GridSearchCV(RandomForestClassifier(), param\_grid=param\_grid, cv=3, scoring='f1\_weighted')

*# Fit the model using the training data*

grid\_search.fit(X\_train, y\_train)

*# Get the best estimator (model) from the grid search*

best\_model = grid\_search.best\_estimator\_

from sklearn.naive\_bayes import MultinomialNB

from sklearn.model\_selection import GridSearchCV

from sklearn.metrics import accuracy\_score, precision\_score, recall\_score, f1\_score, classification\_report

*# Define the parameter grid for GridSearchCV (alpha is the main parameter for tuning)*

param\_grid = {

    'alpha': [0.1, 0.5, 1.0, 2.0, 5.0]

}

*# Initialize GridSearchCV with the MultinomialNB*

grid\_search\_nb = GridSearchCV(MultinomialNB(), param\_grid=param\_grid, cv=3, scoring='f1\_weighted')

*# Fit the model using the training data*

grid\_search\_nb.fit(X\_train, y\_train)

*# Get the best estimator (model) from the grid search*

best\_nb\_model = grid\_search\_nb.best\_estimator\_

*# Make predictions on the test set*

y\_pred\_nb = best\_nb\_model.predict(X\_test)

*# Calculate evaluation metrics*

accuracy\_nb = accuracy\_score(y\_test, y\_pred\_nb)

precision\_nb = precision\_score(y\_test, y\_pred\_nb, average='weighted')

recall\_nb = recall\_score(y\_test, y\_pred\_nb, average='weighted')

f1\_nb = f1\_score(y\_test, y\_pred\_nb, average='weighted')

*# Print evaluation results*

print("Best Multinomial Naive Bayes Model Results:")

print(f"Accuracy: {accuracy\_nb:.4f}")

print(f"Precision: {precision\_nb:.4f}")

print(f"Recall: {recall\_nb:.4f}")

print(f"F1-Score: {f1\_nb:.4f}")

print("\nClassification Report:\n", classification\_report(y\_test, y\_pred\_nb))

y\_pred\_best = best\_model.predict(X\_test)

print("Best Model (Random Forest) Results After Hyperparameter Tuning:")

print(f"Accuracy: {accuracy\_score(y\_test, y\_pred\_best):.4f}")

print(f"Precision: {precision\_score(y\_test, y\_pred\_best, average='weighted'):.4f}")

print(f"Recall: {recall\_score(y\_test, y\_pred\_best, average='weighted'):.4f}")

print(f"F1-Score: {f1\_score(y\_test, y\_pred\_best, average='weighted'):.4f}")

print("\nClassification Report:\n", classification\_report(y\_test, y\_pred\_best))

**Flowchart:-**

*# Install the necessary library if not already installed*

!pip install graphviz

from graphviz import Digraph

*# Create a new directed graph*

dot = Digraph()

*# Add nodes for each step in the workflow*

dot.node('A', 'Dataset\n(Data with correct and incorrect grammar)')

dot.node('B', 'Data Preprocessing\n(Text cleaning, tokenization, stopwords removal, lemmatization)')

dot.node('C', 'Feature Extraction\n(TF-IDF Vectorization)')

dot.node('D', 'Model Training\n(Random Forest Classifier)')

dot.node('E', 'Output\n(Classification Report with Accuracy, Precision, Recall, F1-score)')

*# Define the edges (connections) between the steps*

dot.edge('A', 'B', label='Data Processing')

dot.edge('B', 'C', label='Extract Features')

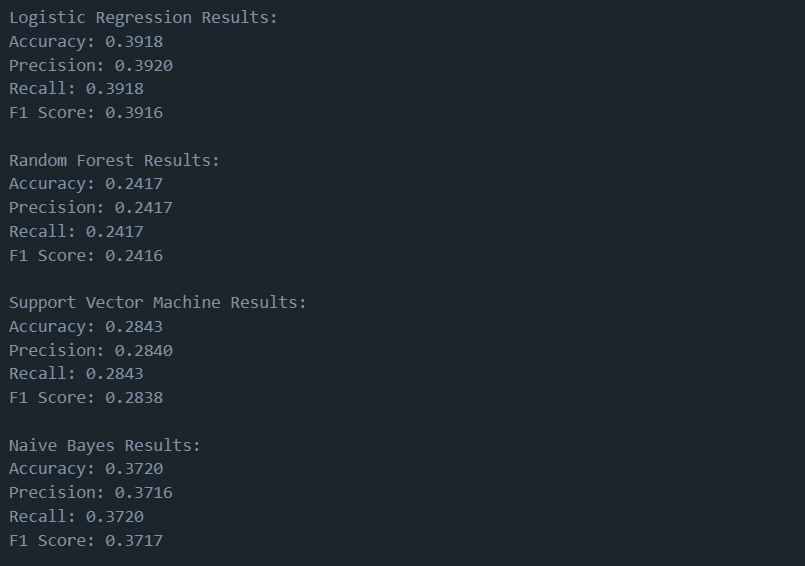
dot.edge('C', 'D', label='Train Model')

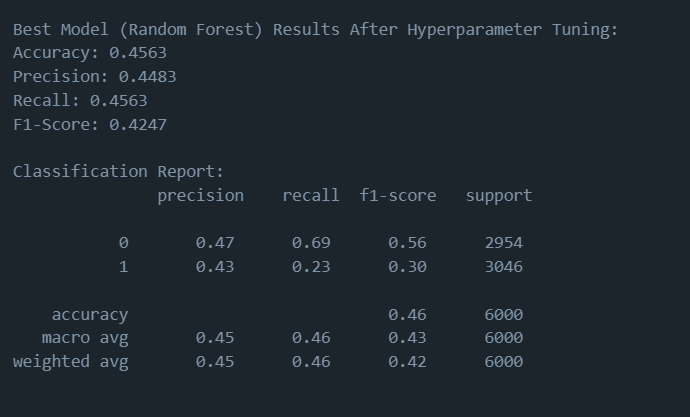
dot.edge('D', 'E', label='Evaluate Model')

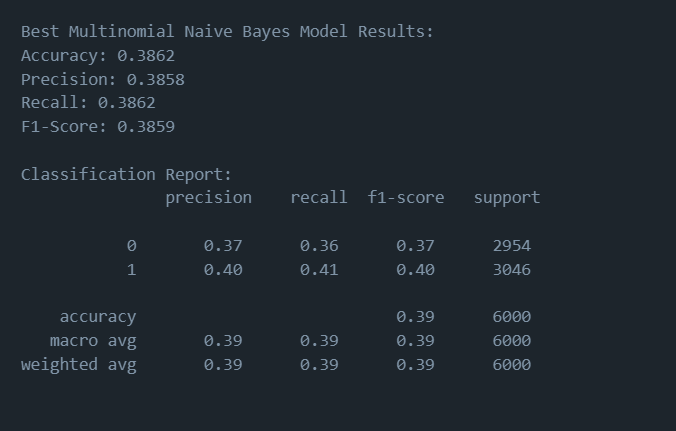
*# Render the flowchart and display it*

dot.render('nlp\_workflow\_1', format='png', view=True)

# RESULT:







# ATTACHMENTS:

1. 

2. <https://www.kaggle.com/datasets/vipin20/nlp-word-correction>