PHASE – 4

(Data & Model Analysis, Feature Extraction and Build & Train the Model)

NAME	**HEMANATH M D**
REGISTER NUMBER	**61772221L01**
NM_ID	**aut2221l301**
PROJECT TITLE	**BUILDING A SMARTER AI-POWERED SPAM CLASSIFIER**

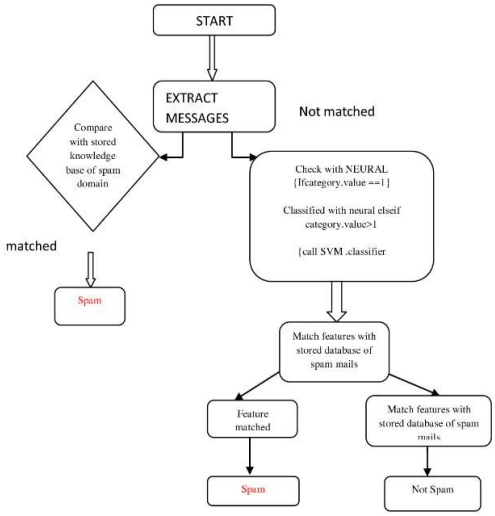
DATA ANALYSIS:

Model : Neural Networks

Data analysis using neural networks is like teaching a computer to recognize and understand patterns in data, just like how we humans recognize things.

- **Data Input:** Imagine we have a lot of emails - some are spam (unwanted junk emails) and some are not. Each email is like a piece of data.
- **Learning:** we use a neural network to teach our computer to recognize patterns in these emails. For instance, it learns that spam emails often contain words like "buy now," "free," or "win."
- **Recognition:** After learning from the data, the computer can look at a new email and decide whether it's spam or not. If it sees lots of words like "buy now" and "free," it might say, "This is likely spam."
- **Improvement:** The more emails we give it, the better it gets at recognizing spam. It learns to spot more patterns and becomes better at avoiding false alarms (marking non-spam as spam) or missing real spam.
- **Applications:** we can use this spam classifier to automatically filter out spam emails from our inbox, keeping our email safe and organized.

The purpose of this data analysis with a neural network is to save our time by automatically identifying and filtering out annoying and potentially harmful spam emails. It's like having a smart assistant who's really good at spotting and getting rid of junk mail for us.



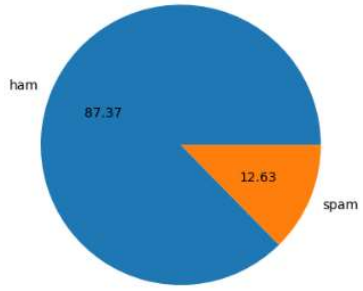
The dataset shall be analysed through the following processes:

- Exploratory Data Analysis (EDA).
- Length of the Message
  - Num_characters
  - Num_words
  - Num_sentence
- Word Frequency Analysis.
- Message length Analysis.

Exploratory Data Analysis (EDA):

Checking ham and spam count:

```
In [54]: # Count the values of 'Type' column
         df['Type'].value_counts()

Out[54]: Type
         0    4516
         1     653
         Name: count, dtype: int64

In [55]: # Plot a pie chart for 'Type' column

         plt.pie(df['Type'].value_counts(),labels=['ham','spam'],autopct="%0.2f")

Out[55]: ([<matplotlib.patches.Wedge at 0x11592292bd0>,
           <matplotlib.patches.Wedge at 0x11591ec45d0>],
          [Text(-1.0144997251399075, 0.4251944351600247, 'ham'),
           Text(1.014499764949479, -0.4251943401757036, 'spam')],
          [Text(-0.5533634864399495, 0.23192423736001344, '87.37'),
           Text(0.5533635081542612, -0.23192418555038377, '12.63')])
```



Code Explanation:

- df['Type'].value_counts(): This line is like counting how many times we find different things in a drawer labeled " Type." In this case, there are 4,516 emails labeled "ham" (non-spam) and 653 emails labeled "spam."
- # EDA: This comment suggests that we're moving into Exploratory Data Analysis (EDA). EDA is like taking a closer look at your data to understand it better.
- !pip install matplotlib: This line installs a library called Matplotlib, which is used for creating visualizations like charts and graphs.
- plt.pie(df['Type'].value_counts(), labels=['ham', 'spam'], autopct="%0.2f"): This code creates a pie chart. It takes the counts we found in step 1 and represents them in a pie chart. The "ham" slice is larger because there are more "ham" emails. The "autopct" part adds percentage labels to the chart.

Finding Length of the Message:

1. **Checking number of characters:**

```
In [56]: # Count the number of characters in a message
         df['num_of_characters']=df['Message'].apply(len)
```

2. **Checking number of Words:**

```
In [58]: # Count the number of words in a message
         df['num_of_words']=df['Message'].apply(lambda x:len(nltk.word_tokenize(x)))
```

3. **Checking number of Sentence:**

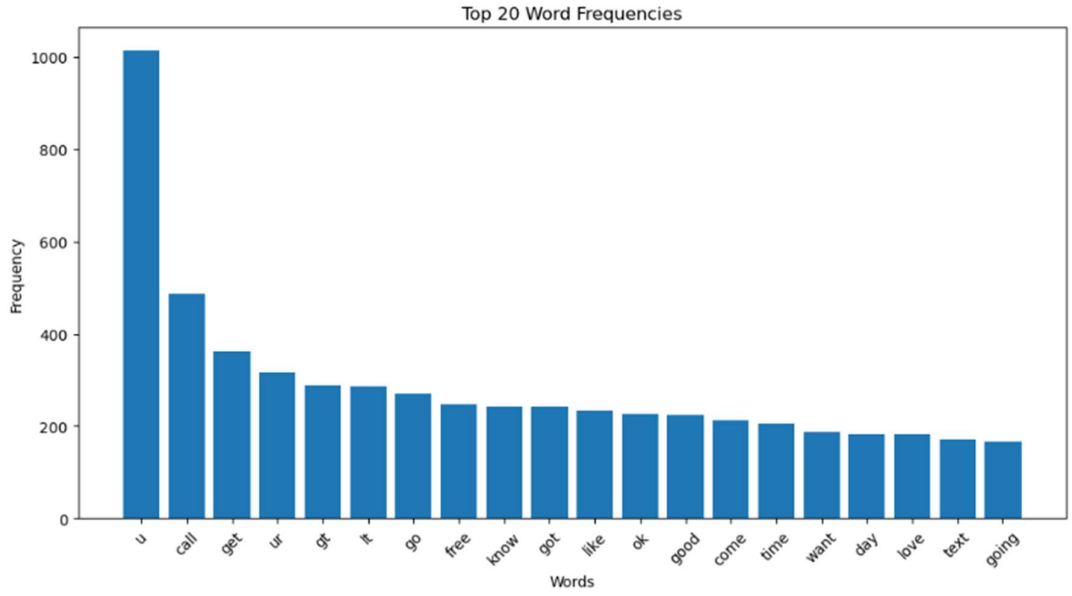
```
In [61]: # Count the number of sentences in a message
         df['num_of_sentences']=df['Message'].apply(lambda x:len(nltk.sent_tokenize(x)))
```

Out[62]:

	Type	Message	num_of_characters	num_of_words	num_of_sentences
0	0	Go until jurong point, crazy.. Available only ...	111	24	2
1	0	Ok lar... Joking wif u oni...	29	8	2
2	1	Free entry in 2 a wkly comp to win FA Cup fina...	155	37	2
3	0	U dun say so early hor... U c already then say...	49	13	1
4	0	Nah I don't think he goes to usf, he lives aro...	61	15	1
...	...	...	...	...	...
5567	1	This is the 2nd time we have tried 2 contact u...	161	35	4
5568	0	Will ì_ b going to esplanade fr home?	37	9	1
5569	0	Pity, * was in mood for that. So...any other s...	57	15	2
5570	0	The guy did some bitching but I acted like i'd...	125	27	1
5571	0	Rofl. Its true to its name	26	7	2

5169 rows × 5 columns

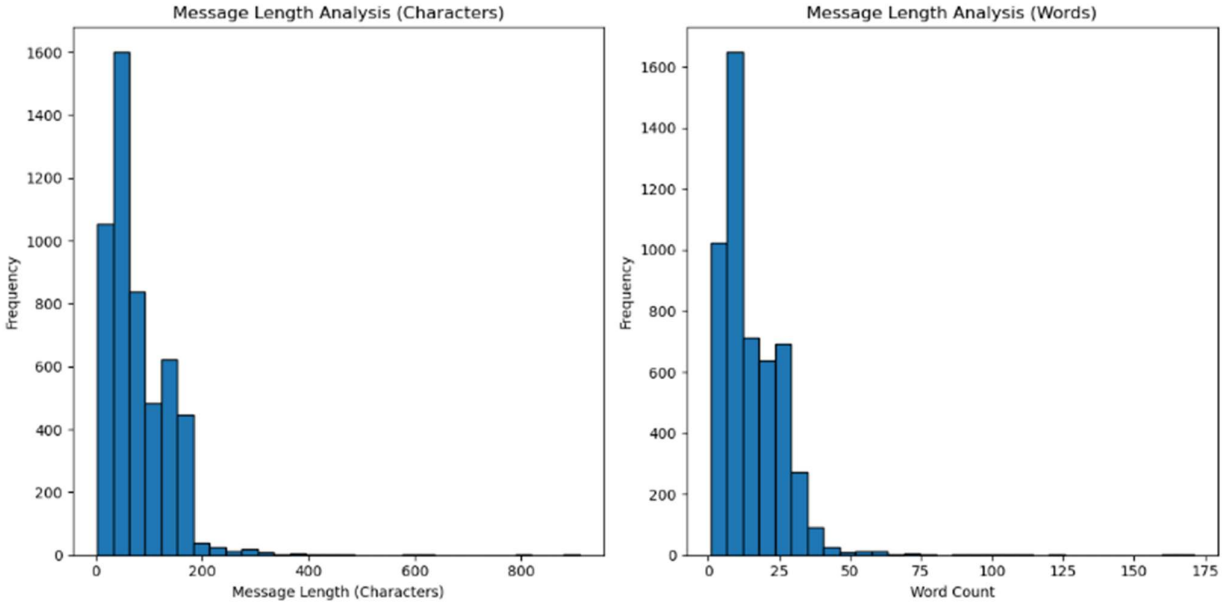
WORD FREQUENCY ANALYSIS:



In [117]:
```
# Word Frequency Analysis

import pandas as pd
import nltk
from nltk.corpus import stopwords
from collections import Counter
import matplotlib.pyplot as plt

# Download NLTK stopwords dataset
nltk.download('stopwords')

# Combine all text into a single string
text = ' '.join(df['Message'])

# Tokenize the text
words = nltk.word_tokenize(text)

# Convert to lowercase and remove stopwords
stop_words = set(stopwords.words('english'))
filtered_words = [word.lower() for word in words if word.isalpha() and word.lower() not in stop_words]

# Count word frequencies
word_freq = Counter(filtered_words)

# Get the most common words and their frequencies
most_common_words = word_freq.most_common(20)  # Change the number to get the top N words

# Plot the word frequencies
plt.figure(figsize=(12, 6))
words, frequencies = zip(*most_common_words)
plt.bar(words, frequencies)
plt.xlabel('Words')
plt.ylabel('Frequency')
plt.title('Top 20 Word Frequencies')
plt.xticks(rotation=45)
plt.show()
```

MESSAGE LENGTH ANALYSIS:

```
In [118]: # message length Analysis

          import pandas as pd
          import matplotlib.pyplot as plt

          # Calculate message length in characters
          df['Message_Length'] = df['Message'].str.len()

          # Calculate message length in words
          df['Word_Count'] = df['Message'].str.split().apply(len)

          # Plot the distribution of message lengths
          plt.figure(figsize=(12, 6))

          # Histogram of message length in characters
          plt.subplot(1, 2, 1)
          plt.hist(df['Message_Length'], bins=30, edgecolor='black')
          plt.xlabel('Message Length (Characters)')
          plt.ylabel('Frequency')
          plt.title('Message Length Analysis (Characters)')

          # Histogram of message length in words
          plt.subplot(1, 2, 2)
          plt.hist(df['Word_Count'], bins=30, edgecolor='black')
          plt.xlabel('Word Count')
          plt.ylabel('Frequency')
          plt.title('Message Length Analysis (Words)')

          plt.tight_layout()  # Ensures proper spacing of subplots
          plt.show()
```



Code Explanation:

Import Libraries: The code begins by importing two important libraries: pandas for data handling and matplotlib.pyplot for creating visual plots and charts.

Load Data: It assumes we have a dataset in a CSV file (replace 'your_data.csv' with our actual file name) and uses pd.read_csv to load this data into a DataFrame called 'df.'

Calculate Message Length:

1. It creates two new columns in the DataFrame: 'Message_Length' and 'Word_Count.'
2. 'Message_Length' stores the number of characters in each text message.
3. 'Word_Count' stores the number of words in each text message.

Plot Distribution:

- The code sets up a plot area using plt.figure(figsize=(12, 6)), which is like preparing a canvas for drawing.
- It creates two histograms (bar charts) side by side to analyze the distribution of message lengths.
- The first histogram shows the distribution of message lengths in characters.
- The second histogram shows the distribution of message lengths in words.

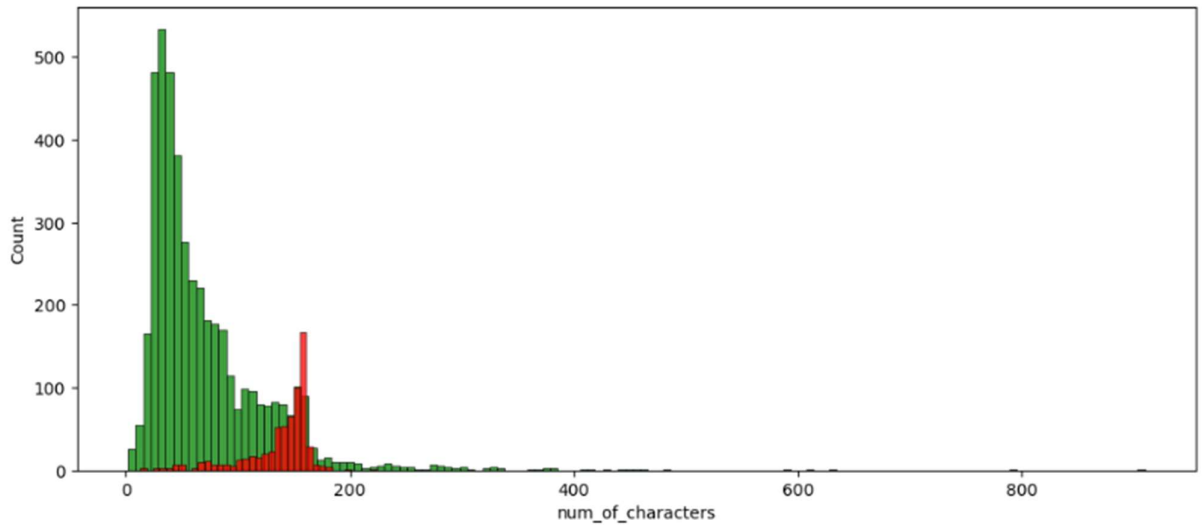
Customizing Plots:

- For each histogram, it specifies the number of bins (divisions) and the color of the edges of the bars to make the plot visually appealing.
- It labels the X and Y axes and gives each histogram a title.
- Show the Plots: plt.tight_layout() ensures that the two plots have proper spacing between them, and plt.show() displays the plots on the screen.

Histogram Representation:

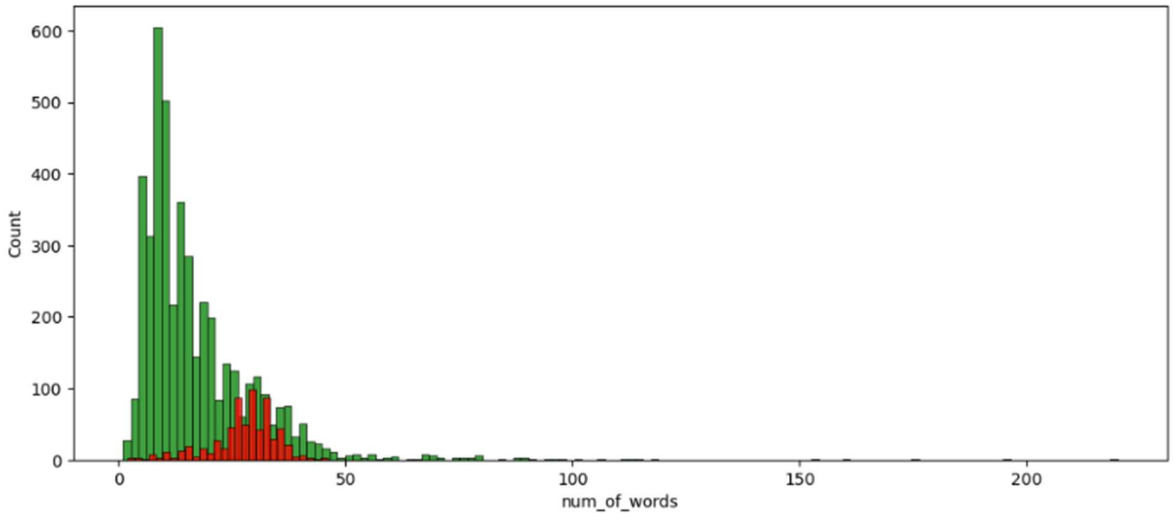
```
In [66]: # Plot a histogram for 'num_of_characters' column

         plt.figure(figsize=(12,5))
         sns.histplot(df[df['Type']==0]['num_of_characters'],color='green')
         sns.histplot(df[df['Type']==1]['num_of_characters'],color='red')
```

```
Out[66]: <Axes: xlabel='num_of_characters', ylabel='Count'>
```




```
In [67]: # Plot a histogram for 'num_of_words' column

         plt.figure(figsize=(12,5))
         sns.histplot(df[df['Type']==0]['num_of_words'],color='green')
         sns.histplot(df[df['Type']==1]['num_of_words'],color='red')
```

```
Out[67]: <Axes: xlabel='num_of_words', ylabel='Count'>
```



Plot Histogram for 'num_of_characters' column:

- A histogram is created to visualize the distribution of the 'num_of_characters' column in the DataFrame.
- The 'ham' messages are represented in green, and the 'spam' messages are represented in red.

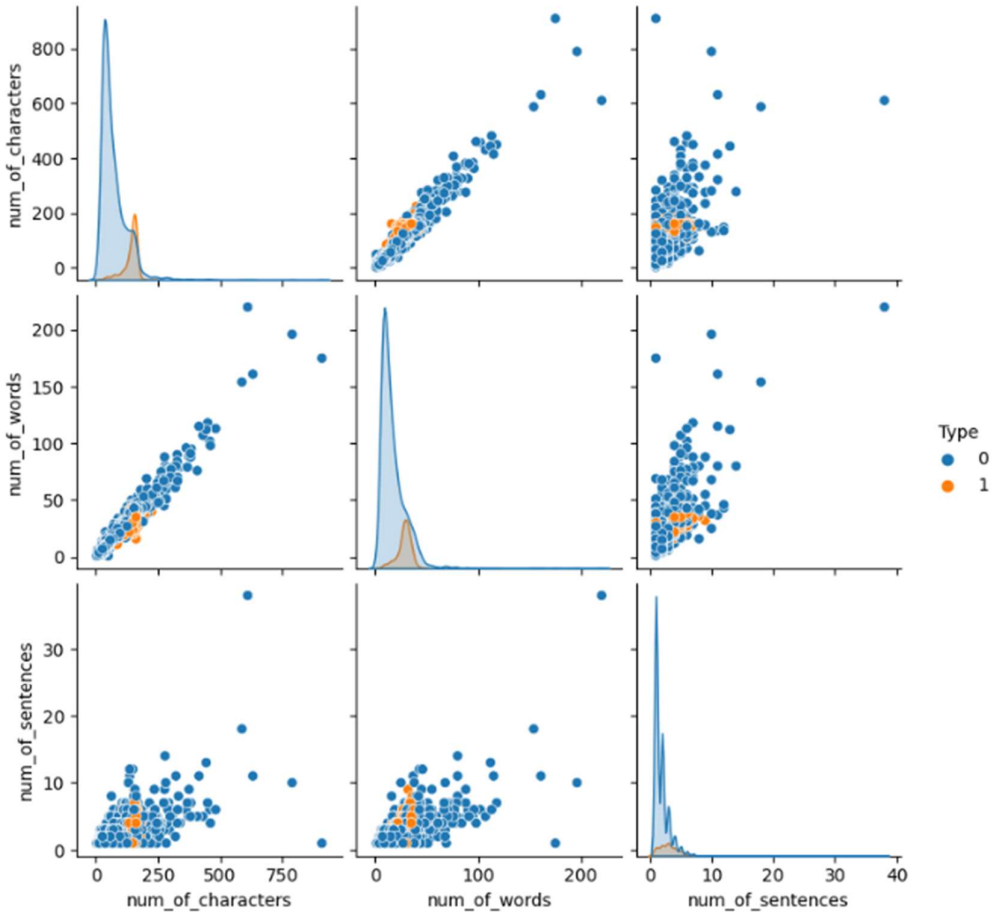
- The first plt.figure(figsize=(12,5)) statement specifies the size of the plot figure for the 'num_of_characters' histogram.

Plot Histogram for 'num_of_words' column:

- Another histogram is plotted, this time for the 'num_of_words' column in the DataFrame.
- Just like before, 'ham' messages are shown in green, and 'spam' messages are in red.
- The second plt.figure(figsize=(12,5)) statement sets the size of the plot figure for the 'num_of_words' histogram.

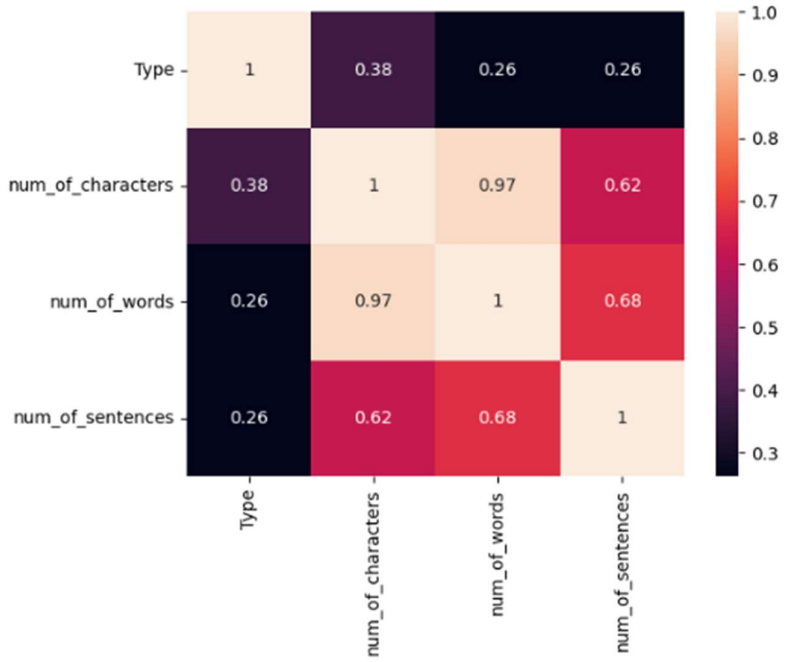
```
In [68]: # Plot a pair plot to visualize the relationships between the variables in the dataset by pairing them in a grid
         sns.pairplot(df,hue='Type')
```

```
Out[68]: <seaborn.axisgrid.PairGrid at 0x11592516910>
```



HEAT MAP:

```
In [69]: # Plot a heat map to visualize the correlation matrix of values
         # Exclude non-numeric columns (e.g., 'text' is a non-numeric column)
         import numpy as np
         numeric_df = df.select_dtypes(include=[np.number])

         # Plot the correlation matrix for numeric columns
         sns.heatmap(numeric_df.corr(), annot=True)
Out[69]: <Axes: >
```



- **Filter Numeric Columns:**

The code starts by selecting the numeric columns from the DataFrame df and creates a new DataFrame called numeric_df. Non-numeric columns (e.g., 'text') are excluded.

- **Plot a Heat Map:**
  - ✓ A heat map is generated to visually represent the correlation matrix of values in the numeric_df.
  - ✓ The sns.heatmap function from the Seaborn library is used for this purpose.
  - ✓ Correlation values are annotated and displayed within the heatmap.

MOST USED WORDS IN SPAM MESSAGES:

In [75]:
```
# Most used words in spam messages

spam_corpus = []
for msg in df[df['Type'] == 1]['transformed_text'].tolist():
    for word in msg.split():
        spam_corpus.append(word)
```

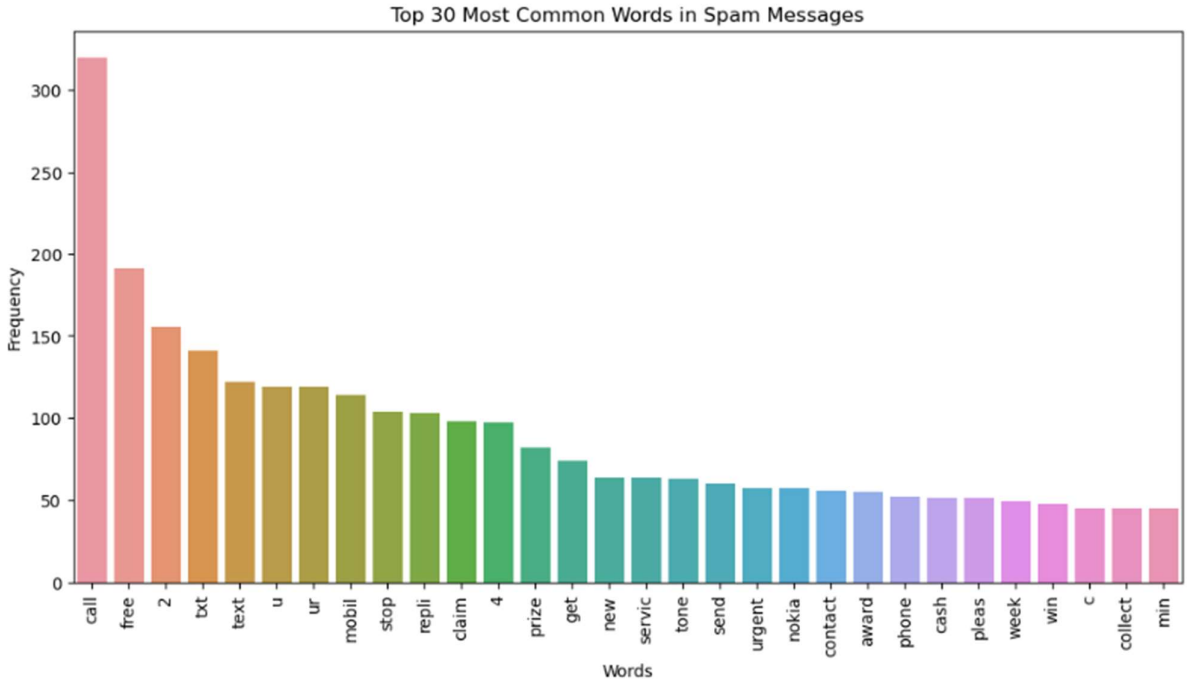
In [76]:
```
# number of most used words

len(spam_corpus)
```

Out[76]: 9939

In [77]:
```
spam_corpus
```

Out[77]:
```
['free',
 'entri',
 '2',
 'wkli',
 'comp',
 'win',
 'fa',
 'cup',
 'final',
 'tkt',
 '21st',
 'may',
 'text',
 'fa',
 '87121',
 'receiv',
 'entri',
 'question',
 'std',
```

In [78]:
```
# Top 30 Most Common Words in Spam Messages
word_counts = Counter(spam_corpus)
common_words = dict(word_counts.most_common(30))
words = list(common_words.keys())
counts = list(common_words.values())

plt.figure(figsize=(12, 6))
sns.barplot(x=words, y=counts)
plt.xticks(rotation='vertical')
plt.xlabel('Words')
plt.ylabel('Frequency')
plt.title('Top 30 Most Common Words in Spam Messages')
plt.show()
```



MOST USED WORDS IN HAM MESSAGES:

In [79]:
```
# Most used words in ham messages

ham_corpus = []
for msg in df[df['Type'] == 0]['transformed_text'].tolist():
    for word in msg.split():
        ham_corpus.append(word)
```

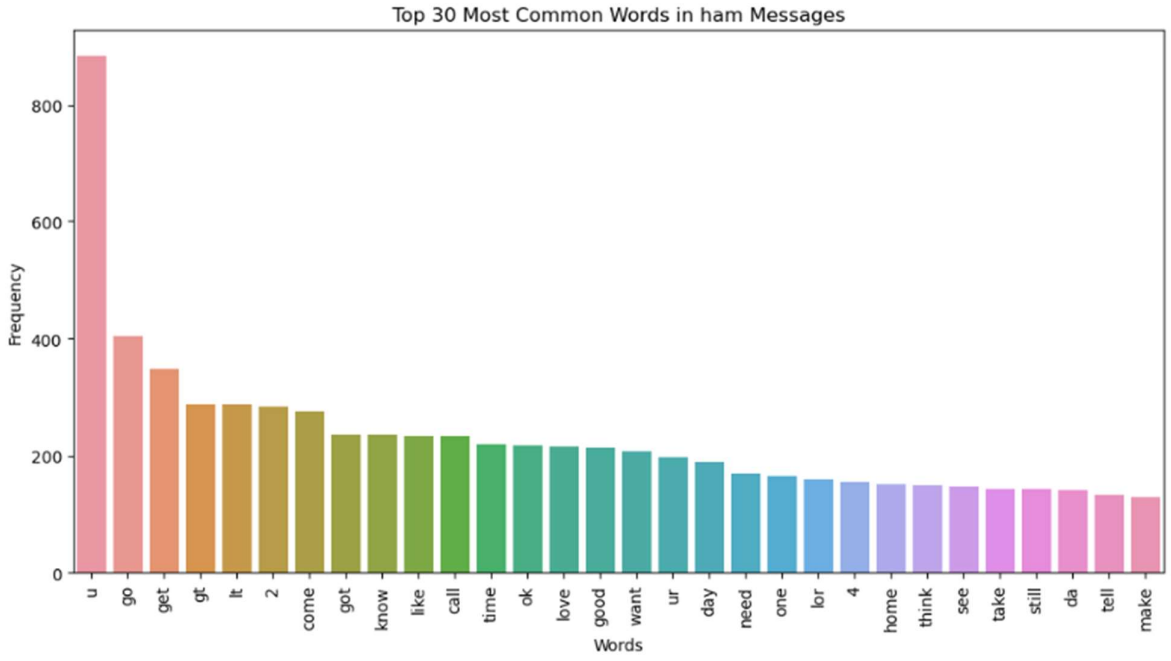
In [80]:
```
# number of most used words

len(ham_corpus)
```

Out[80]: 35404

In [81]:
```
ham_corpus
```

Out[81]:
```
['go',
 'jurong',
 'point',
 'crazi',
 'avail',
 'bugi',
 'n',
 'great',
 'world',
 'la',
 'e',
 'buffet',
 'cine',
 'got',
 'amor',
 'wat',
 'ok',
 'lar',
 'joke',
```

In [82]:
```
# Top 30 Most Common Words in ham Messages
word_counts = Counter(ham_corpus)
common_words = dict(word_counts.most_common(30))
words = list(common_words.keys())
counts = list(common_words.values())

plt.figure(figsize=(12, 6))
sns.barplot(x=words, y=counts)
plt.xticks(rotation='vertical')
plt.xlabel('Words')
plt.ylabel('Frequency')
plt.title('Top 30 Most Common Words in ham Messages')
plt.show()
```



FEATURE EXTRACTION:

Feature extraction is a dimensionality reduction technique used in machine learning and data analysis. It involves transforming or selecting the most relevant information (features) from a dataset while discarding less important or redundant data. Feature extraction is a crucial step in the data preprocessing pipeline and is important for several reasons:

Dimensionality Reduction: Many datasets have a large number of features, which can lead to the curse of dimensionality. This can result in increased computation time, overfitting, and reduced model performance. Feature extraction helps reduce the number of features while retaining essential information.

Improved Model Performance: Feature extraction can lead to better model performance. By selecting the most informative features, models can focus on the relevant patterns in the data, resulting in more accurate predictions and generalization to new data.

Reduced Overfitting: Feature extraction reduces the risk of overfitting, where a model becomes too specialized to the training data and fails to generalize to new data. By reducing the number of features and focusing on the most relevant ones, overfitting can be mitigated.

Enhanced Interpretability: Simplifying the dataset through feature extraction can make it easier to interpret the model's results. It becomes more straightforward to understand the factors influencing the model's decisions.

Computational Efficiency: Feature extraction reduces the computational load, as models process a smaller set of features, leading to faster training and inference times.

Noise Reduction: Some features in a dataset may contain noise or be irrelevant to the task at hand. Feature extraction can help filter out these noisy features, improving the quality of the data used for modeling.

Improved Data Visualization: Feature extraction can make data visualization more effective. By reducing the data to a manageable number of dimensions, it becomes easier to create meaningful visualizations.

Domain-Specific Knowledge Integration: Domain experts can often guide feature extraction by selecting or engineering features that are known to be relevant for a particular problem. This can lead to more effective models.

Enhanced Data Preprocessing: Feature extraction is a critical component of data preprocessing, which is essential for preparing data for machine learning. It helps clean and refine the data before feeding it to models.

```
In [84]: # Import necessary dependencies for feature extraction

         from sklearn.feature_extraction.text import TfidfVectorizer

In [85]: # Perform feature extraction

         tfidf=TfidfVectorizer()

         # x is the input feature
         x=tfidf.fit_transform(df['transformed_text']).toarray()

         # y is the ouput feature
         y=df['Type'].values

In [86]: # Input feature

         x

Out[86]: array([[0., 0., 0., ..., 0., 0., 0.],
                [0., 0., 0., ..., 0., 0., 0.],
                [0., 0., 0., ..., 0., 0., 0.],
                ...,
                [0., 0., 0., ..., 0., 0., 0.],
                [0., 0., 0., ..., 0., 0., 0.],
                [0., 0., 0., ..., 0., 0., 0.]])

In [87]: x.shape

Out[87]: (5169, 6708)

In [88]: # Output feature

         y

Out[88]: array([0, 0, 1, ..., 0, 0, 0])

In [89]: y.shape

Out[89]: (5169,)
```

CODE EXPLANATION:

Import Dependencies:

- The code imports the necessary dependencies, specifically the TfidfVectorizer from scikit-learn (sklearn), which is used for feature extraction from text data.

Initialize TF-IDF Vectorizer:

- It creates a TF-IDF vectorizer object called tfidf. TF-IDF stands for Term Frequency-Inverse Document Frequency, a method for converting text data into numerical feature vectors.

Perform Feature Extraction:

- The code applies feature extraction to the 'transformed_text' column of the DataFrame df using the TF-IDF vectorizer.
- The result is stored in variable x, representing the input features.

Input Feature (x) Explanation:

- The x variable contains the transformed text data as numerical feature vectors after TF-IDF feature extraction.
- x.shape is used to determine the shape of the x array, which reveals the number of samples and features.

Output Feature (y) Preparation:

- The code creates the output feature y by extracting the 'Type' column from the DataFrame, which likely represents the classification labels.

Output Feature (y) Explanation:

- The y variable contains the output feature, which typically represents the class labels (e.g., 'ham' or 'spam').
- y.shape is used to determine the shape of the y array, revealing the number of samples.

BUILDING MODEL FOR SPAM CLASSIFIER:

```
In [90]: # import the dependency for spliting train and test sets
         from sklearn.model_selection import train_test_split

In [91]: # Split the input and output features into train and test sets
         x_train,x_test,y_train,y_test = train_test_split(x,y,test_size=0.2,random_state=2)
```

Import Dependency for Train-Test Split:

- The code imports the necessary dependency, train_test_split, from scikit-learn (sklearn). This function is used to split the dataset into training and testing subsets.

Split the Data into Train and Test Sets:

- The code performs the actual data split.
- x represents the input features, and y represents the output features or class labels.
- The data is split into training and testing sets using train_test_split.
- x_train and y_train represent the training input and output features, respectively.
- x_test and y_test represent the testing input and output features, respectively.
- The test_size parameter specifies the proportion of data to allocate for the test set (in this case, 20% of the data).
- The random_state parameter ensures reproducibility by setting a specific random seed.

```
In [1]: !pip install tensorflow
```

```
In [95]: # Import the necessary dependencies for creating a neural network

from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.optimizers import Adam
from sklearn.metrics import accuracy_score
```

```
In [96]: # Create a neural network with a input layer, three hidden layers and a ouput layer
# No.of neurons in input layer is determined by input shape and input layer uses 'relu' activation function
# 500 neurons in 1st hidden layer and it uses 'relu' activation function
# 500 neurons in 2nd hidden layer and it uses 'relu' activation function
# 100 neurons in 3rd hidden layer and it uses 'relu' activation function
# 100 neurons in 4th hidden layer and it uses 'relu' activation function
# 1 neuron in output  layer and it uses 'sigmoid' activation function

model=Sequential()
model.add(Dense(units=500,input_shape=(6708,),activation='relu'))
model.add(Dense(units=500,activation='relu'))
model.add(Dense(units=100,activation='relu'))
model.add(Dense(units=100,activation='relu'))
model.add(Dense(units=1,activation='sigmoid'))
```

```
In [98]: # Compile the model
# Use binary_crossentropy loss function
# Use adam optimizer
# Use accuracy metrics for monitoring

model.compile(loss='binary_crossentropy' ,optimizer='adam',metrics='accuracy')
```

```
In [99]: # Train the model on the  dataset

model.fit(x_train,y_train,epochs=10,batch_size=64)
```

```
Epoch 1/10
65/65 [==============================] - 7s 52ms/step - loss: 0.2721 - accuracy: 0.8672
Epoch 2/10
65/65 [==============================] - 3s 49ms/step - loss: 0.0975 - accuracy: 0.9775
Epoch 3/10
65/65 [==============================] - 3s 49ms/step - loss: 0.0139 - accuracy: 0.9981
Epoch 4/10
65/65 [==============================] - 3s 48ms/step - loss: 7.9564e-04 - accuracy: 0.9998
Epoch 5/10
65/65 [==============================] - 3s 49ms/step - loss: 1.5202e-04 - accuracy: 1.0000
Epoch 6/10
65/65 [==============================] - 3s 47ms/step - loss: 6.1699e-05 - accuracy: 1.0000
```

Install TensorFlow:

The code begins by installing the TensorFlow library using the !pip install tensorflow command. TensorFlow is a deep learning framework.

Import Dependencies for Neural Network:

It imports the necessary dependencies for creating a neural network using TensorFlow and some functions for model evaluation from scikit-learn.

Create a Neural Network:

- A neural network is defined with the following architecture:
  - ✓ Input layer with a ReLU activation function.
  - ✓ Three hidden layers with 500, 500, and 100 neurons, each using ReLU activation.
  - ✓ An output layer with 1 neuron using a sigmoid activation function.

Compile the Model:

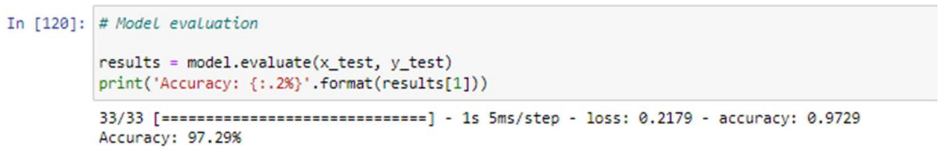
- The model is compiled with the following settings:
  - ✓ Loss function: Binary cross-entropy (commonly used for binary classification tasks).
  - ✓ Optimizer: Adam optimizer (a popular optimization algorithm).
  - ✓ Metric for monitoring: Accuracy.

Train the Model:

The model is trained on the training data (x_train and y_train) for 10 epochs (training iterations), with a batch size of 64.

Model Evaluation:

- ✓ The model's performance is evaluated on the test data (x_test and y_test).
- ✓ The evaluation results are stored in the results variable.
- ✓ The code then prints the model's accuracy, displaying it as a percentage.



Model:

```
In [121]: # Use the model on user input

def transform_text(text):
    # Convert the message to lower case
    text=text.lower()

    # Tokenize the message
    text=nltk.word_tokenize(text)

    # Remove the special characters in the message
    y=[]
    for i in text:
        if i.isalnum():
            y.append(i)
    text=y[:]
    y.clear()

    # Remove the stop words and punctuations in the message
    for i in text:
        if i not in stopwords.words('english') and i not in string.punctuation:
            y.append(i)
    text=y[:]
    y.clear()

    # Stemming
    for i in text:
        y.append(ps.stem(i))

    return " ".join(y)

user_text = input('Input the text: ')
preprocessed_text=transform_text(user_text)
vectorized_text=tfidf.transform([preprocessed_text]).toarray()

prediction = model.predict(vectorized_text)

print('Spam level: {:.2%}'.format(prediction[0][0]))

if prediction > 0.8:
    print('Spam!')
else:
    print('Not spam!')
```

Input the Message:

```
Input the text: freemsg hey there darling it s been 3 week s now
```

Output:

```
Input the text: freemsg hey there darling it s been 3 week s now and no word back  i d like some fun you up for it still  tb ok
xxx std chgs to send    1 50 to rcv,freemsg hey darl 3 week word back like fun still tb ok xxx std chg send 1 50 rcv
1/1 [==============================] - 0s 33ms/step
Spam level: 100.00%
Spam!
```

CODE EXPLANATION:

Text Preprocessing (transform_text function):

- ✓ The transform_text function takes a user's input text as its argument.
- ✓ It starts by converting the input text to lowercase to ensure consistent handling of text.
- ✓ The text is tokenized, meaning it's split into individual words or tokens using the NLTK library's word_tokenize function.

- ✓ Special characters in the text are removed, and the remaining words are collected in a list called y. This step helps clean the text.
- ✓ Stop words (common words like "the," "and," "is") and punctuation are removed from the text to keep only meaningful words.
- ✓ The text goes through stemming using the NLTK Porter Stemmer (ps.stem(i)) to reduce words to their root form.
- ✓ Finally, the processed words are joined to create a clean, preprocessed text.

User Input:

- ✓ The code prompts the user to input text: user_text.

Preprocess User Input:

- ✓ The transform_text function is called to preprocess the user's input text (user_text), making it suitable for model input.
- ✓ The preprocessed text is stored in the variable preprocessed_text.

Vectorize the Text:

- ✓ The preprocessed text is transformed into numerical feature vectors using the TF-IDF vectorizer (tfidf.transform). This step converts the text into a format the model can understand.
- ✓ The resulting feature vector is stored in the variable vectorized_text.

Make a Prediction:

- ✓ The model is used to predict whether the text is spam or not. The prediction is based on the provided feature vector (vectorized_text).
- ✓ The prediction result is stored in the variable prediction.

Display the Result:

- ✓ The code displays the "Spam level," which is the model's prediction score as a percentage.
- ✓ If the prediction score is greater than 80% (0.8), it's classified as "Spam!"; otherwise, it's classified as "Not spam!"