Importing Libraries

In [1]:
```
import numpy as np
from sklearn.pipeline import Pipeline
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.naive_bayes import MultinomialNB
from sklearn.model_selection import ParameterGrid
from sklearn.metrics import precision_score, recall_score, f1_score
import re
```

Problem 1- Reading Data

In [2]:
```
#Importing Positive Data
positiveFilePath = "clickbait.txt"
positiveList = []
with open(positiveFilePath, "r") as file:
    for line in file:
        stripped_line = line.strip()
        positiveList.append(stripped_line)
#print(positiveList)
#print(len(positiveList))

#Importing Negative Data
negativeFilePath = "not-clickbait.txt"
negativeList = []
with open(negativeFilePath, "r") as file:
    for line in file:
        stripped_line = line.strip()
        negativeList.append(stripped_line)
#print(negativeList)
#print(len(negativeList))
```

In [3]:
```
#Shuffling the 2 Lists Together
temp = positiveList + negativeList
label1 = 1
label2 = 0
merged_labels = [label1 if idx < len(positiveList) else label2 for idx in range(len(temp))]

combined_lists = list(zip(merged_labels, temp))
np.random.shuffle(combined_lists)
shuffledLabels, shuffledData = zip(*combined_lists)
size = len(temp)
#print(size)

#Splitting the Data into Train, Test, and Val
trainRatio = 0.8
testRatio = 0.1
valRatio = 0.1

trainSize = int(size * trainRatio)
testSize = int(size * testRatio)
valSize = int(size * valRatio)

trainData = shuffledData[:trainSize]
testData = shuffledData[trainSize:trainSize + testSize]
valData = shuffledData[trainSize + testSize:]
trainLabel = shuffledLabels[:trainSize]
testLabel = shuffledLabels[trainSize:trainSize + testSize]
valLabel = shuffledLabels[trainSize + testSize:]

trainingDataSize = len(trainData)
testingDataSize = len(testData)
valDataSize = len(valData)
positiveListSize = len(positiveList)
negativeListSize = len(negativeList)
wholeDataSize = len(positiveList) + len(negativeList)

print(f"Training data target: ", trainingDataSize)
print(f"Testing data size: ", testingDataSize)
print(f"Validation data size: ", valDataSize)
print(f"Training data target rate: ", round((positiveListSize / wholeDataSize) * trainingDataSize))
print(f"Testing data target rate: ", round((positiveListSize / wholeDataSize) * testingDataSize))
print(f"Validation data target rate: ", round((positiveListSize / wholeDataSize) * valDataSize))
```

```
Training data target:  1910
Testing data size:  238
Validation data size:  240
Training data target rate:  651
Testing data target rate:  81
Validation data target rate:  82
```

PROBLEM 2 – Baseline Performance (10 pts – Answer in Blackboard) Assume you have a trivial baseline classifier that flags every text presented to it as clickbait. What is the precision, recall, and F1-score of such a classifier on your test set? Do you think there is another good baseline classifier that would give you higher F-1 score?

Precision=((TP)/(TP+FP))=((651)/(651+1259))=0.341 because there are 651 is the number of True Positive and 1259 is the number of False Positives.

Recall=((TP)/(TP+FN))=((651)/(651+0))=1 because 651 is the number of True Positives and 0 is the number of False Negatives.

F-1 Score=((2xPxR)/(P+R))=((2x0.341x1)/(0.341+1))=0.501 because Precision is 0.341 and Recall is 1.

In [4]:
```
p = (((651) / (651 + 1259)))
#print(p)
f = ((2 * p * 1)/ (p + 1))
#print(f)
```

PROBLEM 3 – Training a single Bag-of-Words (BOW) Text Classifier

In [5]:
```
#yes1 for clickbait and yn0 for non-clickbait

# Create a pipeline with CountVectorizer and MultinomialNB
pipeline = Pipeline([
    ('vectorizer', CountVectorizer(ngram_range=(1, 2))),
    ('classifier', MultinomialNB())
])

# Fit the pipeline on the training set
pipeline.fit(trainData, trainLabel)

# Make predictions on the training and validation sets
predTrainLabels = pipeline.predict(trainData)
predValLabels = pipeline.predict(valData)

# Compute precision, recall, and F1-score on the training set
trainPrecision = precision_score(trainLabel, predTrainLabels, average='binary')
trainRecall = recall_score(trainLabel, predTrainLabels, average='binary')
trainFiScore = f1_score(trainLabel, predTrainLabels, average='binary')

# Compute precision, recall, and F1-score on the validation set
valPrecision = precision_score(valLabel, predValLabels, average='binary')
valRecall = recall_score(valLabel, predValLabels, average='binary')
valF1Score = f1_score(valLabel, predValLabels, average='binary')

# Print the results
print(f"Train Precision: {trainPrecision:.2f}")
print(f"Train Recall: {trainRecall:.2f}")
print(f"Train F1-Score: {trainFiScore:.2f}")

print(f"Validation Precision: {valPrecision:.2f}")
print(f"Validation Recall: {valRecall:.2f}")
print(f"Validation F1-Score: {valF1Score:.2f}")
```

```
Train Precision: 0.99
Train Recall: 1.00
Train F1-Score: 0.99
Validation Precision: 0.90
Validation Recall: 0.86
Validation F1-Score: 0.88
```

PROBLEM 4 – Hyperparameter Tuning

In [6]:
```
# Define the parameter grid for grid search
parameterGrid = {
    'vectorizer__max_df': [0.5, 0.75, 1.0],
    'classifier__alpha': [0.1, 1.0, 10.0],
    'vectorizer__ngram_range': [(1, 1), (1, 2)]
}
results = []

# Iterate over parameter combinations
for parameters in ParameterGrid(parameterGrid):
    # Create a pipeline with CountVectorizer and MultinomialNB
    pipeline = Pipeline([
        ('vectorizer', CountVectorizer(max_df = parameters['vectorizer__max_df'], ngram_range = parameters['vectorizer__ngram_range'])),
        ('classifier', MultinomialNB(alpha = parameters['classifier__alpha']))
    ])

    # Fit the pipeline on the training set
    pipeline.fit(trainData, trainLabel)

    # Make predictions on the validation set
    predValLabels = pipeline.predict(valData)

    # Calculate precision, recall, and F1-score
    precision = precision_score(valLabel, predValLabels)
    recall = recall_score(valLabel, predValLabels)
    f1 = f1_score(valLabel, predValLabels)

    results.append({
        'params': parameters,
        'precision': precision,
        'recall': recall,
        'f1_score': f1
    })

# Sort the results by F1-score in descending order
results.sort(key=lambda x: x['f1_score'], reverse=True)

# Print the top and bottom results
num_results_to_show = min(5, len(results))
print(f"Top {num_results_to_show} Results:")
for i in range(num_results_to_show):
    print(f"Parameters: {results[i]['params']}")
    print(f"Precision: {results[i]['precision']:.2f}")
    print(f"Recall: {results[i]['recall']:.2f}")
    print(f"F1-Score: {results[i]['f1_score']:.2f}")
    print()

if len(results) > 5:
    print("Bottom Results:")
    for i in range(-5, 0):
        print(f"Parameters: {results[i]['params']}")
        print(f"Precision: {results[i]['precision']:.2f}")
        print(f"Recall: {results[i]['recall']:.2f}")
        print(f"F1-Score: {results[i]['f1_score']:.2f}")
        print()
```

```
Top 5 Results:
Parameters: {'classifier__alpha': 1.0, 'vectorizer__max_df': 0.5, 'vectorizer__ngram_range': (1, 2)}
Precision: 0.90
Recall: 0.86
F1-Score: 0.88

Parameters: {'classifier__alpha': 1.0, 'vectorizer__max_df': 0.75, 'vectorizer__ngram_range': (1, 2)}
Precision: 0.90
Recall: 0.86
F1-Score: 0.88

Parameters: {'classifier__alpha': 1.0, 'vectorizer__max_df': 1.0, 'vectorizer__ngram_range': (1, 2)}
Precision: 0.90
Recall: 0.86
F1-Score: 0.88

Parameters: {'classifier__alpha': 1.0, 'vectorizer__max_df': 0.5, 'vectorizer__ngram_range': (1, 1)}
Precision: 0.88
Recall: 0.86
F1-Score: 0.87

Parameters: {'classifier__alpha': 1.0, 'vectorizer__max_df': 0.75, 'vectorizer__ngram_range': (1, 1)}
Precision: 0.88
Recall: 0.86
F1-Score: 0.87

Bottom Results:
Parameters: {'classifier__alpha': 10.0, 'vectorizer__max_df': 0.75, 'vectorizer__ngram_range': (1, 1)}
Precision: 0.93
Recall: 0.71
F1-Score: 0.81

Parameters: {'classifier__alpha': 10.0, 'vectorizer__max_df': 1.0, 'vectorizer__ngram_range': (1, 1)}
Precision: 0.93
Recall: 0.71
F1-Score: 0.81

Parameters: {'classifier__alpha': 10.0, 'vectorizer__max_df': 0.5, 'vectorizer__ngram_range': (1, 2)}
Precision: 0.94
Recall: 0.70
F1-Score: 0.80

Parameters: {'classifier__alpha': 10.0, 'vectorizer__max_df': 0.75, 'vectorizer__ngram_range': (1, 2)}
Precision: 0.94
Recall: 0.70
F1-Score: 0.80

Parameters: {'classifier__alpha': 10.0, 'vectorizer__max_df': 1.0, 'vectorizer__ngram_range': (1, 2)}
Precision: 0.94
Recall: 0.70
F1-Score: 0.80
```

PROBLEM 5 – Model selection

In [7]:
```
temp1 = results[0]['params']['vectorizer__max_df']
temp2 = results[0]['params']['vectorizer__ngram_range']
temp3 = results[0]['params']['classifier__alpha']
#print(temp1, temp2, temp3)

pipeline = Pipeline([
    ('vectorizer', CountVectorizer(max_df = results[0]['params']['vectorizer__max_df'], ngram_range = results[0]['params']['vectorizer__ngram_range'])),
    ('classifier', MultinomialNB(alpha = results[0]['params']['classifier__alpha']))
])

# Fit the pipeline on the training set
pipeline.fit(trainData, trainLabel)

# Apply the selected model to the test set
predTestLabels = pipeline.predict(testData)

# Compute precision, recall, and F1-score on the test set
testPrecision = precision_score(testLabel, predTestLabels)
testRecall = recall_score(testLabel, predTestLabels)
testFiScore = f1_score(testLabel, predTestLabels)

# Print the results on the test set
print(f"Test Precision: {testPrecision:.2f}")
print(f"Test Recall: {testRecall:.2f}")
print(f"Test F1-Score: {testFiScore:.2f}")
```

```
Test Precision: 0.92
Test Recall: 0.85
Test F1-Score: 0.88
```

PROBLEM 6 – Key Indicators

In [8]:
```
# Get the MultinomialNB classifier from the selected model
classifier = pipeline.named_steps['classifier']

# Get the feature log probabilities (log likelihoods)
featureLogProbabilities = classifier.feature_log_prob_

# Get the vocabulary from the CountVectorizer
vectorizer = pipeline.named_steps['vectorizer']
vocabulary = vectorizer.get_feature_names_out()

# Calculate the log odds for each feature (word)
logOdds = featureLogProbabilities[1] - featureLogProbabilities[0]

# Create a dictionary to store words and their log odds
wordLogOdds = {word: logOdd for word, logOdd in zip(vocabulary, logOdds)}

# Sort words by log odds in descending order
sortedWords = sorted(wordLogOdds.items(), key=lambda x: x[1], reverse=True)

# Select the top 5 words as strong clickbait indicators
clickbaitIndicators = [word for word, _ in sortedWords[:5]]

# Print the top 5 clickbait indicators
print("Top 5 Clickbait Indicators:")
for word in clickbaitIndicators:
    print(word)
```

```
Top 5 Clickbait Indicators:
won believe
you won
you ll
ll never
believe what
```

PROBLEM 7 – Regular expressions

In [9]:
```
top_keywords = clickbaitIndicators

# Create a regular expression pattern to match any of the keywords with word boundaries
pattern = r'\b(?:' + '|'.join(re.escape(keyword) for keyword in top_keywords) + r')\b'

# Function to check if the text contains any of the top keywords
def checkForClickbait(text):
    return re.search(pattern, text, flags=re.IGNORECASE) is not None

# Apply the function to your test set and store the results
testResults = [checkForClickbait(text) for text in testData]

# Calculate the number of true positives, false positives, true negatives, and false negatives
tp = sum(1 for pred, actual in zip(testResults, testLabel) if pred and actual == 1)
fp = sum(1 for pred, actual in zip(testResults, testLabel) if pred and actual == 0)
tn = sum(1 for pred, actual in zip(testResults, testLabel) if not pred and actual == 0)
fn = sum(1 for pred, actual in zip(testResults, testLabel) if not pred and actual == 1)

# Calculate precision and recall
precision = tp / (tp + fp)
recall = tp / (tp + fn)

# Print the precision and recall
print("Precision:", precision)
print("Recall:", recall)
```

```
Precision: 1.0
Recall: 0.07692307692307693
```

PROBLEM 8 – Comparing results (15pts – Answer in Blackboard) Compare your rules-based classifier from the previous problem with your machine-learning solution. Which classifier showed the best model metrics? Why do you think it performed the best? How did both compare to your trivial baseline (Problem 2)? If you had more time to try to improve this clickbait detection solution, what would you explore? (There is no single correct answer to this question; review your results and come up with your own ideas)

The classifier that showed the best model metrics was the machine-learning solution. I think it performed the best because looking at the metric results for the rules-based classifier, the classifier barely predicted anything which resulted in it correctly predicting the Positive it did predict on but because it barely predicted, there were a lot of False Negatives. The classifier from the machine-learning solutions did significantly better compared to the baseline. We can see this by the machine-learning solutions' Precision, Recall, and F1-Score being higher than that of the baseline. The rules-based classifier did similar compared to the baseline. We can see this by the rules-based having a higher Precision but a lower Recall. unremly, the clickbait detection solution barely predicts the phrase to be clickbait. This can be shown by the Precision being 1 or near 1 and then Recall being near 0. The current method predicts the words that have the highest probability to be clickbait and lowest probability to be non clickbait. Because of this, the words being selected are very unique words that don't appear that often resulting in the testing phrases to rarely containing them If I had more time to try to improve this clickbait detection solution, I would explore try to determine a method that not only finds the words with the highest probability to be clickbait and lowest probability to be non clickbait, but also appear a certain amount of times. Currently if a word appears once in the clickbait and 0 times in non clickbait, it would have a 100% probability of being clickbait however this is not a good identifier. If I could determine a minimum number of times a word appears to be considered an indicator, then the words chosen as indicators would not only have good probabilities of being clickbait and not non clickbait, but they would also appear more frequently than the current method.