In [1]:
```
#Import Libraries
import pandas as pd
import math
```

In [2]:
```
#Load Dataset ( already split by Reuters : train + test )

train = pd.read_csv("ModApte_train.csv")
test = pd.read_csv("ModApte_test.csv")

print("Train size:", len(train))
print("Test size:", len(test))

print(train.columns)
print(test.columns)
```

```
Train size: 9603
Test size: 3299
Index(['text', 'text_type', 'topics', 'lewis_split', 'cgis_split', 'old_id',
       'new_id', 'places', 'people', 'orgs', 'exchanges', 'date', 'title'],
      dtype='object')
Index(['text', 'text_type', 'topics', 'lewis_split', 'cgis_split', 'old_id',
       'new_id', 'places', 'people', 'orgs', 'exchanges', 'date', 'title'],
      dtype='object')
```

In [3]:
```
import re
def parse_topics(s) :
    if not isinstance(s, str):
        return []
    # Find all items inside single quotes
    return re.findall(r"'([^']+)'", s)

train["topics_list"] = train["topics"].apply(parse_topics)
test["topics_list"]  = test["topics"].apply(parse_topics)

print(train["topics"].head(10).to_list())
print(train["topics_list"].head(10).to_list())
```

```
["['cocoa']", "['grain' 'wheat' 'corn' 'barley' 'oat' 'sorghum']", "['veg-oil' 'lins
eed' 'lin-oil' 'soy-oil' 'sun-oil' 'soybean' 'oilseed'\n 'corn' 'sunseed' 'grain' 's
orghum' 'wheat']", '[]', "['earn']", "['acq']", "['earn']", "['earn' 'acq']", "['ear
n']", "['earn']"]
[['cocoa'], ['grain', 'wheat', 'corn', 'barley', 'oat', 'sorghum'], ['veg-oil', 'lin
seed', 'lin-oil', 'soy-oil', 'sun-oil', 'soybean', 'oilseed', 'corn', 'sunseed', 'gr
ain', 'sorghum', 'wheat'], [], ['earn'], ['acq'], ['earn'], ['earn', 'acq'], ['ear
n'], ['earn']]
```

In [4]:
```
# Keep only articles with exactly one topic
def keep_single_labeled(df):
    df = df.copy()
    df["topic_count"] = df["topics_list"].apply(len)
    df = df[df["topic_count"] == 1]   # keeps only rows with exactly 1 topic
    return df

train_f = keep_single_labeled(train)
```

```
test_f  = keep_single_labeled(test)

print("\nTrain size (after removing empty + multi-label topics):", len(train_f))
print("Test size (after removing empty + multi-label topics):", len(test_f))
```

```
Train size (after removing empty + multi-label topics): 6552
Test size (after removing empty + multi-label topics): 2581
```

In [5]:
```
def convert_to_binary(topics_list):
    # topics_list always has exactly one element after filtering
    if topics_list[0] == "earn":
        return 1   # earn
    else:
        return 0   # not-earn

train_f["binary_label"] = train_f["topics_list"].apply(convert_to_binary)
test_f["binary_label"]  = test_f["topics_list"].apply(convert_to_binary)

# Check how many earn vs not-earn articles we have
print("Training label distribution:")
print(train_f["binary_label"].value_counts())

print("\nTest label distribution:")
print(test_f["binary_label"].value_counts())
```

```
Training label distribution:
binary_label
0    3712
1    2840
Name: count, dtype: int64

Test label distribution:
binary_label
0    1498
1    1083
Name: count, dtype: int64
```

In [6]:
```
def preprocess(text):
    # Convert text to string and lowercase
    text = str(text).lower()

    # Remove everything except letters and spaces
    text = re.sub(r'[^a-z\s]', ' ', text)

    # Split text into words
    tokens = text.split()

    return tokens

# Apply preprocessing ONLY to the filtered datasets
train_f["tokens"] = train_f["text"].apply(preprocess)
test_f["tokens"]  = test_f["text"].apply(preprocess)

# Optional sanity check
print(train_f["tokens"].head(3))
```

```
0    [showers, continued, throughout, the, week, in...
4    [champion, products, inc, said, its, board, of...
5    [computer, terminal, systems, inc, said, it, h...
Name: tokens, dtype: object
```

In [7]:
```
def build_vocab(token_lists, min_freq=1):
    """
    Build vocabulary from training tokens only.
    Returns:
        vocab (word -> index)
    """

    word_counts = {}

    # Count word frequencies
    for tokens in token_lists:
        for w in tokens:
            word_counts[w] = word_counts.get(w, 0) + 1

    # Build vocabulary dictionary
    vocab = {}
    idx = 0

    for w, c in word_counts.items():
        if c >= min_freq:
            vocab[w] = idx
            idx += 1

    return vocab
```

In [8]:
```
vocab = build_vocab(train_f["tokens"].tolist())
print("Vocabulary size:", len(vocab))
```

```
Vocabulary size: 19591
```

In [9]:
```
# COunt words per class ( earn vs not-earn)


def count_words_by_class(token_lists, labels):
    # Dictionary for counting words in earn articles
    word_counts_earn = {}

    # Dictionary for counting words in not-earn articles
    word_counts_not_earn = {}

    # Total number of word tokens (including repeats) in each class
    total_words_earn = 0
    total_words_not_earn = 0

    # Go through each document (article) one by one
    for tokens, y in zip(token_lists, labels):

        # If this article is "earn"
        if y == 1:
            for w in tokens:
                word_counts_earn[w] = word_counts_earn.get(w, 0) + 1
                total_words_earn += 1

        # If this article is "not-earn"
        else:
            for w in tokens:
                word_counts_not_earn[w] = word_counts_not_earn.get(w, 0) + 1
                total_words_not_earn += 1

    return word_counts_earn, word_counts_not_earn, total_words_earn, total_words_no


# Run using training data
token_lists = train_f["tokens"].tolist()
labels = train_f["binary_label"].tolist()

word_counts_earn, word_counts_not_earn, total_words_earn, total_words_not_earn = co

print("Total words in earn articles:", total_words_earn)
print("Total words in not-earn articles:", total_words_not_earn)

print("Unique words seen in earn articles:", len(word_counts_earn))
print("Unique words seen in not-earn articles:", len(word_counts_not_earn))
```

```
Total words in earn articles: 178278
Total words in not-earn articles: 513721
Unique words seen in earn articles: 7534
Unique words seen in not-earn articles: 17594
```

In [10]:
```
#  Compute Naive Bayes probabilities with Laplace smoothing
#we want to answer questions like:
# If an article is earn, how likely is it to contain the word “profit”?
# That is written as: P(word | class)

# Output: We will compute:
# P(word | earn)
# P(word | not-earn)
# But we store them as log probabilities to avoid math underflow.

import math

def compute_word_log_probs(vocab,
                           word_counts_earn,
                           word_counts_not_earn,
                           total_words_earn,
                           total_words_not_earn):

    vocab_size = len(vocab)

    log_prob_earn = {}
    log_prob_not_earn = {}

    for word in vocab:
        # How many times this word appeared in each class
        count_earn = word_counts_earn.get(word, 0)
        count_not_earn = word_counts_not_earn.get(word, 0)
        # Laplace smoothing
        prob_word_given_earn = (count_earn + 1) / (total_words_earn + vocab_size)
        prob_word_given_not_earn = (count_not_earn + 1) / (total_words_not_earn + v

        # Store LOG probabilities
        log_prob_earn[word] = math.log(prob_word_given_earn)
        log_prob_not_earn[word] = math.log(prob_word_given_not_earn)

    return log_prob_earn, log_prob_not_earn


# Test the function using the counts we got from the training data
log_prob_earn, log_prob_not_earn = compute_word_log_probs(
    vocab,
    word_counts_earn,
    word_counts_not_earn,
    total_words_earn,
    total_words_not_earn
)

print("Example word probabilities:")
print("log P('profit' | earn):", log_prob_earn.get("profit"))
print("log P('profit' | not-earn):", log_prob_not_earn.get("profit"))
```

```
Example word probabilities:
log P('profit' | earn): -4.578584665852126
log P('profit' | not-earn): -8.997207155715452
```

In [11]:

```
# Compute class prior probabilities
num_docs = len(train_f)
num_earn = sum(train_f["binary_label"] == 1)
num_not_earn = sum(train_f["binary_label"] == 0)

log_prior_earn = math.log(num_earn / num_docs)
log_prior_not_earn = math.log(num_not_earn / num_docs)

print("log prior earn:", log_prior_earn)
print("log prior not-earn:", log_prior_not_earn)

# Predict class for ONE article

def predict_one(tokens,
                vocab,
                log_prob_earn,
                log_prob_not_earn,
                log_prior_earn,
                log_prior_not_earn):
    """
    tokens: list of words from one article
    Returns:
      1 -> earn
      0 -> not-earn
    """

    # Start with the class priors
    score_earn = log_prior_earn
    score_not_earn = log_prior_not_earn

    # Add word evidence
    for word in tokens:
        # Ignore words not in vocabulary
        if word in vocab:
            score_earn += log_prob_earn[word]
            score_not_earn += log_prob_not_earn[word]

    # Choose the class with the higher score
    if score_earn > score_not_earn:
        return 1
    else:
        return 0

# Pick one article from training data
sample_tokens = train_f.iloc[0]["tokens"]
true_label = train_f.iloc[0]["binary_label"]

predicted_label = predict_one(
    sample_tokens,
    vocab,
    log_prob_earn,
    log_prob_not_earn,
    log_prior_earn,
    log_prior_not_earn
)

print("True label:", true_label)
print("Predicted label:", predicted_label)
```

```
log prior earn: -0.8359662943776536
log prior not-earn: -0.5681995316268141
True label: 0
Predicted label: 0
```

In [12]:
```
#  Predict for MANY articles
def predict_many(token_lists,
                 vocab,
                 log_prob_earn,
                 log_prob_not,
                 log_prior_earn,
                 log_prior_not):

     """
    Predict labels for many documents.
    Returns a list of predicted labels.
    """

    predictions = []

    for tokens in token_lists:
        y_pred = predict_one(
            tokens,
            vocab,
            log_prob_earn,
            log_prob_not,
            log_prior_earn,
            log_prior_not
        )
        predictions.append(y_pred)

    return predictions
```

In [13]:
```
#  Create K folds (list of index lists)
def compute_accuracy(y_true, y_pred):
    correct = 0
    for t, p in zip(y_true, y_pred):
        if t == p:
            correct += 1
    return correct / len(y_true)
```

In [14]:
```
import random
import math

def make_k_folds(n, k=5, seed=42):

    indices = list(range(n))
    random.seed(seed)
    random.shuffle(indices)

    fold_size = n // k
    folds = []
    start = 0

    for i in range(k):
        end = start + fold_size if i < k - 1 else n
        folds.append(indices[start:end])
        start = end

    return folds
```

In [15]:
```
# Run 5-fold cross-validation for SCRATCH Naive Bayes

def run_5fold_cv_scratch(train_df, k=5, seed=42):

      """
    train_df must have:
      - 'tokens' (list of words)
      - 'binary_label' (1=earn, 0=not-earn)

    Returns:
      - list of fold accuracies
      - mean accuracy
      - std accuracy
    """

    n = len(train_df)
    folds = make_k_folds(n, k=k, seed=seed)

    fold_accuracies = []
    for fold_id in range(k):

        valid_idx = set(folds[fold_id])
        train_idx = [i for i in range(n) if i not in valid_idx]
     # Create fold-train and fold-valid datasets

        fold_train = train_df.iloc[train_idx]
        fold_valid = train_df.iloc[list(valid_idx)]
     # 1) Build vocabulary from fold_train only (NO leakage)
        vocab = build_vocab(fold_train["tokens"].tolist())
    # 2) Count words by class using fold_train only
        wc_earn, wc_not, total_earn, total_not = count_words_by_class(
            fold_train["tokens"].tolist(),
            fold_train["binary_label"].tolist()
        )
     # 3) Compute word log-probabilities (Laplace smoothing)
        log_prob_earn, log_prob_not = compute_word_log_probs(
            vocab, wc_earn, wc_not, total_earn, total_not
        )
     # 4) Compute class priors from fold_train only
        num_docs = len(fold_train)
        num_earn = sum(fold_train["binary_label"] == 1)
        num_not = num_docs - num_earn

        log_prior_earn = math.log(num_earn / num_docs)
        log_prior_not = math.log(num_not / num_docs)
    # 5) Predict on fold_valid and compute accuracy
        y_true = fold_valid["binary_label"].tolist()
        y_pred = predict_many(
            fold_valid["tokens"].tolist(),
            vocab,
            log_prob_earn,
            log_prob_not,
            log_prior_earn,
            log_prior_not
        )

        acc = compute_accuracy(y_true, y_pred)
        fold_accuracies.append(acc)

        print(f"Fold {fold_id+1}/{k} accuracy: {acc:.4f}")

    mean_acc = sum(fold_accuracies) / k
    variance = sum((a - mean_acc) ** 2 for a in fold_accuracies) / k
    std_acc = math.sqrt(variance)

    return fold_accuracies, mean_acc, std_acc
```

In [16]:

```
# RUN 5-FOLD CV for SCRATCH Naive Bayes

fold_accs, mean_acc, std_acc = run_5fold_cv_scratch(train_f, k=5, seed=42)

print("\nFold accuracies:", [round(a, 4) for a in fold_accs])
print("Mean accuracy:", round(mean_acc, 4))
print("Std (variation):", round(std_acc, 4))
```

```
Fold 1/5 accuracy: 0.9458
Fold 2/5 accuracy: 0.9405
Fold 3/5 accuracy: 0.9420
Fold 4/5 accuracy: 0.9267
Fold 5/5 accuracy: 0.9253

Fold accuracies: [0.9458, 0.9405, 0.942, 0.9267, 0.9253]
Mean accuracy: 0.9361
Std (variation): 0.0084
```

In [17]:
```
# ibrary baseline (CountVectorizer + MultinomialNB)
# - Same 5 folds
# - Compare mean accuracy with scratch


from sklearn.feature_extraction.text import CountVectorizer
from sklearn.naive_bayes import MultinomialNB

def tokens_to_text(token_lists):
    """
    Convert tokens back to text so CountVectorizer can read it.
    Example: ['profit','rose'] -> "profit rose"
    """
    return [" ".join(tokens) for tokens in token_lists]


def run_5fold_cv_library(train_df, k=5, seed=42):
    n = len(train_df)
    folds = make_k_folds(n, k=k, seed=seed)

    fold_accuracies = []

    for fold_id in range(k):
        valid_idx = set(folds[fold_id])
        train_idx = [i for i in range(n) if i not in valid_idx]

        fold_train = train_df.iloc[train_idx]
        fold_valid = train_df.iloc[list(valid_idx)]

        # Convert tokens to strings (library expects text)
        X_train_text = tokens_to_text(fold_train["tokens"].tolist())
        X_valid_text = tokens_to_text(fold_valid["tokens"].tolist())

        y_train = fold_train["binary_label"].tolist()
        y_valid = fold_valid["binary_label"].tolist()

        # Bag of Words (library)
        vectorizer = CountVectorizer()
        X_train_vec = vectorizer.fit_transform(X_train_text)
        X_valid_vec = vectorizer.transform(X_valid_text)

        # Naive Bayes (library)
        model = MultinomialNB(alpha=1.0)  # Laplace smoothing
        model.fit(X_train_vec, y_train)

        # Predict and evaluate
        y_pred = model.predict(X_valid_vec)
        acc = compute_accuracy(y_valid, y_pred)

        fold_accuracies.append(acc)
        print(f"[Library] Fold {fold_id+1}/{k} accuracy: {acc:.4f}")

    mean_acc = sum(fold_accuracies) / len(fold_accuracies)
    variance = sum((a - mean_acc) ** 2 for a in fold_accuracies) / len(fold_accurac
    std_acc = math.sqrt(variance)

    return fold_accuracies, mean_acc, std_acc


# RUN  (same k and seed as scratch for fair comparison)
lib_fold_accs, lib_mean_acc, lib_std_acc = run_5fold_cv_library(train_f, k=5, seed=

print("\n[Library] Fold accuracies:", [round(a, 4) for a in lib_fold_accs])
print("[Library] Mean accuracy:", round(lib_mean_acc, 4))
print("[Library] Std (variation):", round(lib_std_acc, 4))
```

```
[Library] Fold 1/5 accuracy: 0.9458
[Library] Fold 2/5 accuracy: 0.9389
[Library] Fold 3/5 accuracy: 0.9443
[Library] Fold 4/5 accuracy: 0.9290
[Library] Fold 5/5 accuracy: 0.9245

[Library] Fold accuracies: [0.9458, 0.9389, 0.9443, 0.929, 0.9245]
[Library] Mean accuracy: 0.9365
[Library] Std (variation): 0.0084
```

In [ ]:

In [ ]: