Part A

Count / Frequency based Naive Bayes Classifier

```
import os
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.feature_extraction.text import TfidfVectorizer, CountVectorizer
from sklearn.pipeline import Pipeline
from sklearn.metrics import (
    accuracy_score,
    classification_report,
    confusion_matrix,
    f1_score
)

# ========================================================
# TODO: Students must implement the following steps:
# 1. Complete the fit method in NaiveBayesClassifier (4 TODOs for log prior and
# 2. Complete the predict method in NaiveBayesClassifier (2 TODOs for log probab
# 3. Complete the data loading calls in Section 2.
# 4. Initialize CountVectorizer with proper parameters in Section 3a.
# 5. Complete the feature transformation (fit_transform and transform) in Sectio
# 6. Initialize and fit the custom nb_model in Section 3b.
# 7. Use the fitted nb_model to generate predictions in Section 4.
# ========================================================


# Data loading function (DO NOT CHANGE)
def load_pubmed_rct_file(filepath):
    """
    Reads a .txt file from the PubMed 20k RCT dataset.
    Returns a DataFrame with 'label' and 'sentence'.
    """
    labels, sentences = [], []
    with open(filepath, 'r', encoding='utf-8') as f:
        for line in f:
            line = line.strip()
            if not line or '\t' not in line:
                continue
            label, sent = line.split('\t', maxsplit=1)
            labels.append(label)
            sentences.append(sent)
    return pd.DataFrame({'label': labels, 'sentence': sentences})
```

```
# Implementing Multinomial Naive Bayes from scratch
class NaiveBayesClassifier:
    """
    Multinomial Naive Bayes Classifier implemented from scratch.
    It is suitable for both Count and TF-IDF features.
    """
    def __init__(self, alpha=1.0):
        self.alpha = alpha
        self.class_priors = {}
        self.feature_log_probs = {}
        self.classes = None
        self.vocabulary_size = 0

    def fit(self, X_counts, y):
        y_array = y.to_numpy()
        self.classes = np.unique(y_array)
        self.vocabulary_size = X_counts.shape[1]
        n_samples = X_counts.shape[0]

        for c in self.classes:
            X_c = X_counts[y_array == c]
            n_c_samples = X_c.shape[0]

            # // TODO: Calculate the log prior and store it in self.class_priors
            # Calculate Class Prior P(C): log(P(C))
            # P(C) = (Number of samples in class c) / (Total number of samples)
            self.class_priors[c] = np.log(n_c_samples / n_samples)

            feature_sum = X_c.sum(axis=0).A1
            total_mass = np.sum(feature_sum)

            # Apply Laplace smoothing (additive smoothing, alpha=1.0 default):
            # P(w_i | C) = (count(w_i, C) + alpha) / (total_words_in_C + alpha *

            # // TODO: Calculate the numerator (with Laplace smoothing)
            numerator = feature_sum + self.alpha

            # // TODO: Calculate the denominator (with Laplace smoothing)
            denominator = total_mass + (self.alpha * self.vocabulary_size)

            # // TODO: Calculate the log likelihood (log(numerator / denominator
            self.feature_log_probs[c] = np.log(numerator / denominator)

    def predict(self, X_counts):
        y_pred = []
        for i in range(X_counts.shape[0]):
            scores = {}

            x_i = X_counts.getrow(i)

            for c in self.classes:
```

```
                log_prob = self.class_priors[c]
                log_likelihoods = self.feature_log_probs[c]

                non_zero_indices = x_i.indices
                non_zero_data = x_i.data

                # // TODO: Complete the log probability calculation for the like
                # Add log likelihoods contribution (Log-Sum Trick):
                # log_prob += sum(count(w_i) * log(P(w_i|C)))

                log_prob += np.dot(non_zero_data, log_likelihoods[non_zero_indic
                scores[c] = log_prob

            # // TODO: Find the key (class label) with the maximum score
            predicted_class = max(scores, key=scores.get)

            y_pred.append(predicted_class)

        # // TODO: Return the final predictions array
        return np.array(y_pred)
```

```
# Load and Prepare Data (DO NOT CHANGE)
dir_path = './'
try:
    train_df = load_pubmed_rct_file(os.path.join(dir_path, 'train.txt'))
    dev_df   = load_pubmed_rct_file(os.path.join(dir_path, 'dev.txt'))
    test_df  = load_pubmed_rct_file(os.path.join(dir_path, 'test.txt'))



    print(f"Train samples: {len(train_df)}")
    print(f"Dev   samples: {len(dev_df)}")
    print(f"Test  samples: {len(test_df)}")

    X_train, y_train = train_df['sentence'], train_df['label']
    X_dev,   y_dev   = dev_df['sentence'],   dev_df['label']
    X_test,  y_test  = test_df['sentence'],  test_df['label']
    target_names = sorted(y_train.unique())
    print(f"Classes: {target_names}")

except FileNotFoundError as e:
    print(f"Error: Dataset file not found. Please ensure the files are uploadec
    X_train, y_train = pd.Series([]), pd.Series([])
    X_test, y_test = pd.Series([]), pd.Series([])
    target_names = []
```

```
Train samples: 83405
Dev   samples: 30212
Test  samples: 30135
Classes: ['BACKGROUND', 'CONCLUSIONS', 'METHODS', 'OBJECTIVE', 'RESULTS']
```

```
# Feature Extraction and Custom Model Training
if X_train is not None and len(X_train) > 0:

    # Initialize and fit the CountVectorizer for count-based features
    count_vectorizer = CountVectorizer(
        lowercase=True,
        strip_accents='unicode',
        stop_words='english',
        # // TODO: Set appropriate ngram_range
        ngram_range=(1,1),
        # // TODO: Set appropriate min_df
        min_df=5
    )

    print("Fitting Count Vectorizer and transforming training data...")
    # // TODO: Fit the vectorizer on X_train and transform
    X_train_counts = count_vectorizer.fit_transform(X_train)
    if X_train_counts is not None:
        print(f"Vocabulary size: {X_train_counts.shape[1]}")

    print("Transforming test data...")
    # // TODO: Transform X_test using the fitted vectorizer
    X_test_counts = count_vectorizer.transform(X_test)


    # Train Custom Naive Bayes Classifier
    print("\nTraining the Custom Naive Bayes Classifier (from scratch)...")

    # // TODO: Initialize the custom NaiveBayesClassifier
    nb_model = NaiveBayesClassifier(alpha=1.0)

    # // TODO: Fit the model using X_train_counts and y_train
    # nb_model.fit(...)
    nb_model.fit(X_train_counts, y_train)
    print("Training complete.")

else:
    print("Skipping feature extraction and training: Training data is empty or
```

```
Fitting Count Vectorizer and transforming training data...
Vocabulary size: 15523
Transforming test data...

Training the Custom Naive Bayes Classifier (from scratch)...
Training complete.
```

```
# Predict and evaluate on test set
print("\n=== Test Set Evaluation (Custom Count-Based Naive Bayes) ===")

# // TODO: Predict y_test_pred using X_test_counts
```

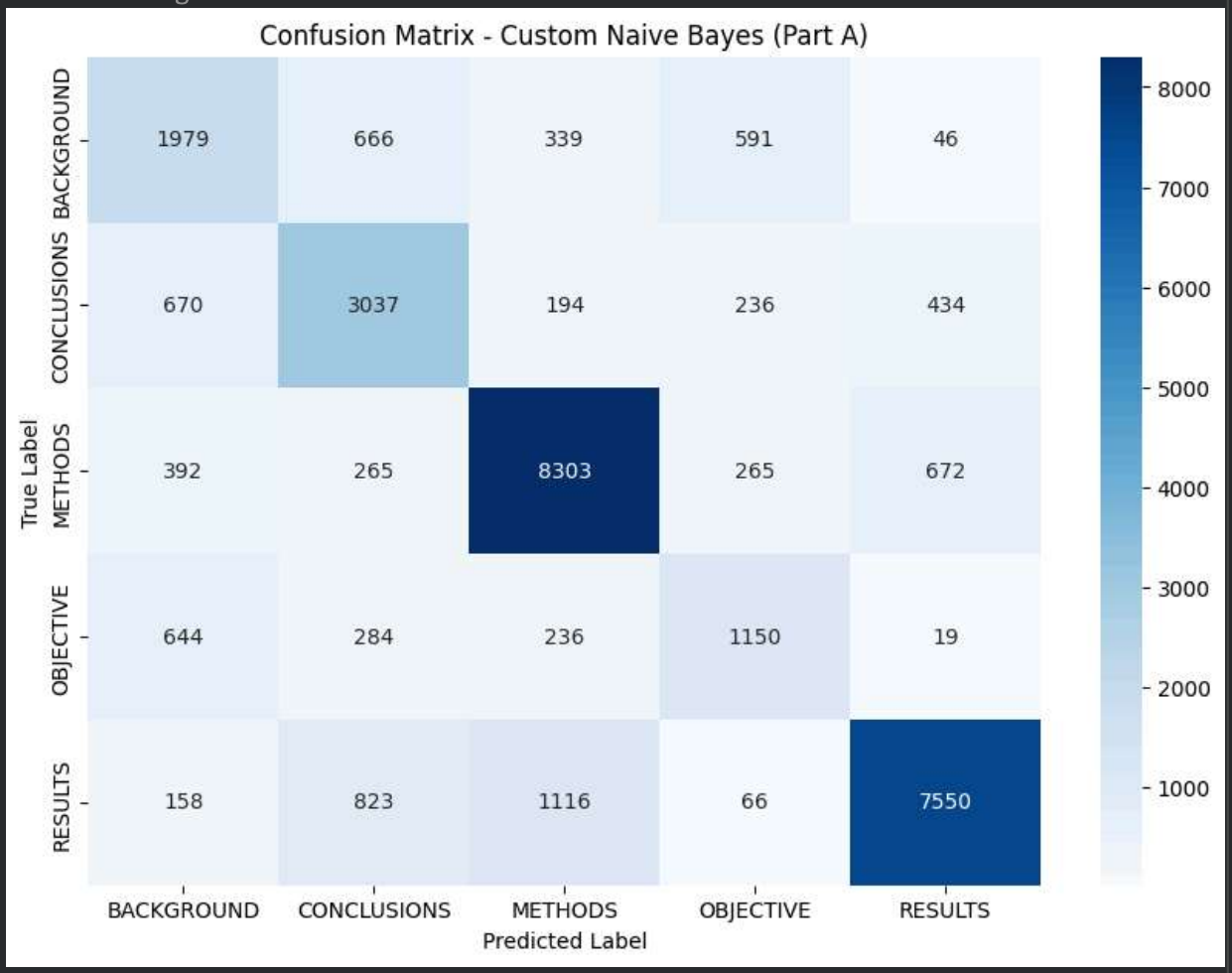
```
y_test_pred = nb_model.predict(X_test_counts)

if y_test_pred is not None:
    print(f"Accuracy: {accuracy_score(y_test, y_test_pred):.4f}")
    print(classification_report(y_test, y_test_pred, target_names=target_names))
    test_f1 = f1_score(y_test, y_test_pred, average='macro')
    print(f"Macro-averaged F1 score: {test_f1:.4f}")
else:
    print("Prediction step failed or incomplete.")
```

```
=== Test Set Evaluation (Custom Count-Based Naive Bayes) ===
Accuracy: 0.7307
              precision    recall  f1-score   support

  BACKGROUND       0.51      0.55      0.53      3621
 CONCLUSIONS       0.60      0.66      0.63      4571
     METHODS       0.81      0.84      0.83      9897
   OBJECTIVE       0.50      0.49      0.50      2333
     RESULTS       0.87      0.78      0.82      9713

    accuracy                           0.73     30135
   macro avg       0.66      0.66      0.66     30135
weighted avg       0.74      0.73      0.73     30135

Macro-averaged F1 score: 0.6603
```

```
# # Confusion Matrix on test set
#     # // TODO: Use the confusion_matrix, matplotlib, and seaborn libraries tc
#     # a visual confusion matrix (heatmap) for the predicted results.
#     # if y_test_pred is not None:
#     #     cm = confusion_matrix(...)
#     #     plt.figure(...)
#     #     sns.heatmap(...)
#     #     plt.show()


print("\n=== Test Set Evaluation (Custom Count-Based Naive Bayes) ===")

y_test_pred = nb_model.predict(X_test_counts)

if y_test_pred is not None and 'y_test' in locals() and len(y_test) > 0:
    print(f"Accuracy: {accuracy_score(y_test, y_test_pred):.4f}")
    print(classification_report(y_test, y_test_pred, target_names=target_names))
    test_f1 = f1_score(y_test, y_test_pred, average='macro')
    print(f"Macro-averaged F1 score: {test_f1:.4f}")

    # Confusion Matrix on test set
    # // TODO: Use the confusion_matrix, matplotlib, and seaborn libraries to
    # a visual confusion matrix (heatmap) for the predicted results.
    cm = confusion_matrix(y_test, y_test_pred, labels=target_names)
```

```
        plt.figure(figsize=(10, 7))
        sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',
                    xticklabels=target_names, yticklabels=target_names)
        plt.title('Confusion Matrix - Custom Naive Bayes (Part A)')
        plt.ylabel('True Label')
        plt.xlabel('Predicted Label')
        plt.show()

    else:
        print("Prediction or evaluation step failed or incomplete.")
```

```
=== Test Set Evaluation (Custom Count-Based Naive Bayes) ===
Accuracy: 0.7307
              precision    recall  f1-score   support

  BACKGROUND       0.51      0.55      0.53      3621
 CONCLUSIONS       0.60      0.66      0.63      4571
     METHODS       0.81      0.84      0.83      9897
   OBJECTIVE       0.50      0.49      0.50      2333
     RESULTS       0.87      0.78      0.82      9713

    accuracy                           0.73     30135
   macro avg       0.66      0.66      0.66     30135
weighted avg       0.74      0.73      0.73     30135

Macro-averaged F1 score: 0.6603
```



Part B

TF-IDF score based Classifier

```
import os
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.naive_bayes import MultinomialNB
from sklearn.pipeline import Pipeline
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import (
    accuracy_score,
    classification_report,
    confusion_matrix,
    f1_score
)

# =====================================================
# TODO: Implement the following steps:
# 1. Define the initial `pipeline` combining TfidfVectorizer and MultinomialNB
# 2. Train the initial pipeline on the training data (X_train, y_train).
# 3. Predict and evaluate the performance of the initial model on the test data
# 4. Define the `param_grid` for hyperparameter tuning.
# 5. Initialize `GridSearchCV` using the pipeline, parameter grid, and appropri
# 6. Fit the Grid Search object using the development data (X_dev, y_dev).
# 7. Print the `best_params_` and `best_score_` found by the grid search.
# =====================================================


# // TODO: Define a Pipeline named 'pipeline' using TfidfVectorizer and Multinc
# Use standard initial parameters
pipeline = Pipeline([
    ('tfidf', TfidfVectorizer()),
    ('nb', MultinomialNB())
])

# // TODO: Train the initial pipeline on the training set
print("Training initial Naive Bayes pipeline...")
if 'X_train' in locals() and len(X_train) > 0:
    pipeline.fit(X_train, y_train)
    print("Training complete.")
else:
    print("Skipping training: X_train not found.")


# Predict and evaluate on test set
# // TODO: Predict y_test_pred and calculate metrics
print("\n=== Test Set Evaluation (Initial Sklearn Model) ===")
y_test_pred = None
if 'pipeline' in locals() and 'X_test' in locals() and len(X_test) > 0:
    try:
        y_test_pred = pipeline.predict(X_test)
```

```
        except Exception as e:
            print(f"Prediction failed: {e}")

    if y_test_pred is not None and 'y_test' in locals() and len(y_test) > 0:
        print(f"Accuracy: {accuracy_score(y_test, y_test_pred):.4f}")
        print(classification_report(y_test, y_test_pred, target_names=target_names)
        print(f"Macro-averaged F1 score: {f1_score(y_test, y_test_pred, average='ma
        pass
    else:
        print("Initial model evaluation skipped: Predictions not available.")


    # Hyperparameter Tuning using GridSearchCV

    # // TODO: Define the parameter grid 'param_grid' to tune both TF-IDF and NB pa
    param_grid = {
        'tfidf__ngram_range': [(1, 1), (1, 2)],  # Test unigrams vs. unigrams+bigra
        'tfidf__use_idf': [True, False],         # Test with and without IDF weight
        'nb__alpha': [0.1, 0.5, 1.0]             # Test with different smoothing va
    }

    # // TODO: Initialize GridSearchCV using the pipeline and param_grid.
    # Ensure cv=3 and scoring='f1_macro' are used.
    grid = GridSearchCV(
        pipeline,
        param_grid,
        cv=3,
        scoring='f1_macro',
        n_jobs=-1,
        verbose=2
    )

    print("\nStarting Hyperparameter Tuning on Development Set...")
    # // TODO: Fit the GridSearchCV object using the development data.
    if 'grid' in locals() and 'X_dev' in locals() and len(X_dev) > 0:
        try:
            grid.fit(X_dev, y_dev)
            print("Grid search complete.")
        except Exception as e:
            print(f"GridSearchCV fitting failed: {e}")
    else:
        print("Skipping grid search: X_dev not found or grid not initialized.")


    if 'grid' in locals() and hasattr(grid, 'best_params_'):
        # // TODO: Print the best parameters and the corresponding best cross-valic
        print("\n--- Grid Search Results ---")
        print(f"Best Parameters found: {grid.best_params_}")
        print(f"Best F1 Macro Score (on dev set): {grid.best_score_:.4f}")
    else:
```

```
    print("\nHyperparameter tuning skipped or failed: Grid Search object not ir
```

```
Training initial Naive Bayes pipeline...
Training complete.

=== Test Set Evaluation (Initial Sklearn Model) ===
Accuracy: 0.7127
              precision    recall  f1-score   support

  BACKGROUND       0.66      0.36      0.47      3621
 CONCLUSIONS       0.61      0.58      0.59      4571
     METHODS       0.69      0.90      0.78      9897
   OBJECTIVE       0.73      0.06      0.11      2333
     RESULTS       0.79      0.87      0.83      9713

    accuracy                           0.71     30135
   macro avg       0.70      0.56      0.56     30135
weighted avg       0.71      0.71      0.68     30135

Macro-averaged F1 score: 0.5573

Starting Hyperparameter Tuning on Development Set...
Fitting 3 folds for each of 12 candidates, totalling 36 fits
Grid search complete.

--- Grid Search Results ---
Best Parameters found: {'nb__alpha': 0.1, 'tfidf__ngram_range': (1, 2), 'tfidf__
Best F1 Macro Score (on dev set): 0.6567
```

Part C

Bayes Optimal Classifier

Part C Draft

```
"""# **Part C**
Bayes Optimal Classifier

Part C Draft
"""

import os
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
from sklearn.ensemble import RandomForestClassifier, VotingClassifier
```

```
from sklearn.ensemble import RandomForestClassifier, VotingClassifier
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.tree import DecisionTreeClassifier
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import FunctionTransformer
from sklearn.naive_bayes import MultinomialNB
from sklearn.calibration import CalibratedClassifierCV
from sklearn.metrics import accuracy_score, f1_score, classification_report, con
from sklearn.model_selection import train_test_split # Used for P(h|D) calculati

# ====================================================
# TODO: Implement the following steps:
# 1. Train all five hypotheses on the sampled training data.
# 2. Compute the Posterior Weights P(h_i | D) using a validation split.
# 3. Fit the VotingClassifier using the sampled training data.
# 4. Make final predictions and evaluate the BOC performance on the test data.
# ====================================================

# Dynamic Data Sampling (DO NOT CHANGE)
BASE_SAMPLE_SIZE = 10000

# Prompt the user for their full SRN
FULL_SRN = input("Please enter your full SRN (e.g., PES1UG22CS345): ")

try:
    # Extract the last three characters and convert to integer
    if len(FULL_SRN) >= 3:
        srn_suffix_str = FULL_SRN[-3:]
        srn_value = int(srn_suffix_str)
    else:
        # Fallback if input is too short
        raise ValueError("SRN too short.")
except (ValueError, IndexError, TypeError):
    # Fallback if SRN is not entered or format is incorrect
    print("WARNING: SRN input failed or format is incorrect. Using 10000.")
    srn_value = 0

# Calculate the final sample size: 10000 + last three SRN digits
SAMPLE_SIZE = BASE_SAMPLE_SIZE + srn_value

print(f"Using dynamic sample size: {SAMPLE_SIZE}")

# Placeholder initialization in case data wasn't loaded in the environment
if 'X_train' not in locals() or len(X_train) == 0:
    print("Warning: Training data not found. Using small placeholder data.")
    X_train = pd.Series(["sample text one"] * 11000)
    y_train = pd.Series(["BACKGROUND"] * 5000 + ["METHODS"] * 6000)
    X_test = pd.Series(["test text one", "test text two"])
    y_test = pd.Series(["BACKGROUND", "METHODS"])
    target_names = ["BACKGROUND", "CONCLUSIONS", "METHODS", "OBJECTIVE", "RESULT

# Use .sample() to get a random subset, and .loc to align labels
```

```
if 'X_train' in locals() and len(X_train) >= SAMPLE_SIZE:
    X_train_sampled_indices = X_train.sample(n=SAMPLE_SIZE, random_state=42).ind
    X_train_sampled = X_train.loc[X_train_sampled_indices]
    y_train_sampled = y_train.loc[X_train_sampled_indices]
    effective_sample_size = SAMPLE_SIZE
else:
    # Handle cases where X_train is smaller than SAMPLE_SIZE
    X_train_sampled = X_train
    y_train_sampled = y_train
    effective_sample_size = len(X_train)

print(f"Actual sampled training set size used: {effective_sample_size}")


# Base TF-IDF parameters (DO NOT CHANGE)
tfidf_params = {
    'lowercase': True,
    'strip_accents': 'unicode',
    'stop_words': 'english',
    'ngram_range': (1, 1), # Using unigrams only to keep feature space small for
    'min_df': 5
}

# Define the five diverse hypotheses/pipelines (DO NOT CHANGE)

# H1: Multinomial Naive Bayes
h1_nb = Pipeline([
    ('tfidf', TfidfVectorizer(**tfidf_params)),
    ('clf', MultinomialNB(alpha=1.0, fit_prior=False))
])

# H2: Logistic Regression
h2_lr = Pipeline([
    ('tfidf', TfidfVectorizer(**tfidf_params)),
    ('clf', LogisticRegression(solver='liblinear', multi_class='auto', max_iter=
])

# H3: Random Forest Classifier
h3_rf = Pipeline([
    ('tfidf', TfidfVectorizer(**tfidf_params)),
    ('clf', CalibratedClassifierCV(
        RandomForestClassifier(n_estimators=50, max_depth=10, random_state=42, n
    ))
])

# H4: Decision Tree Classifier
h4_dt = Pipeline([
    ('tfidf', TfidfVectorizer(**tfidf_params)),
    ('clf', CalibratedClassifierCV(
        DecisionTreeClassifier(max_depth=10, random_state=42), cv=3, method='iso
    ))
])
```

```
])

# H5: K-Nearest Neighbors
h5_knn = Pipeline([
    ('tfidf', TfidfVectorizer(**tfidf_params)),
    ('clf', CalibratedClassifierCV(
        KNeighborsClassifier(n_neighbors=5, n_jobs=-1), cv=3, method='isotonic'
    ))
])

hypotheses = [h1_nb, h2_lr, h3_rf, h4_dt, h5_knn]
hypothesis_names = ['NaiveBayes', 'LogisticRegression', 'RandomForest', 'Decisio


# Training and BOC Implementation (STUDENT TASK)

# // TODO: Train all five hypotheses on X_train_sampled and y_train_sampled usin
print("\nTraining all base models...")
if 'X_train_sampled' in locals() and not X_train_sampled.empty:
    for name, model in zip(hypothesis_names, hypotheses):
        print(f"Training {name}...")
        model.fit(X_train_sampled, y_train_sampled)
    print("All base models trained.")
else:
    print("Skipping training: Sampled data is empty.")


# // TODO: Implement the Posterior Weight Calculation (P(h_i | D)).
# This requires splitting X_train_sampled into a small train_sub/val_sub set
# and calculating the validation log-likelihood for each model. Normalize these

# We will use F1-score on a validation split as an approximation for P(h|D)
try:
    # Create a sub-train and validation set from the sampled data
    X_train_sub, X_val_sub, y_train_sub, y_val_sub = train_test_split(
        X_train_sampled, y_train_sampled, test_size=0.25, random_state=42, strat
    )

    f1_scores = []

    # We need to re-train copies of the models on the sub-training set
    # Define new instances for evaluation
    h1_nb_eval = Pipeline([('tfidf', TfidfVectorizer(**tfidf_params)), ('clf', M
    h2_lr_eval = Pipeline([('tfidf', TfidfVectorizer(**tfidf_params)), ('clf', L
    h3_rf_eval = Pipeline([('tfidf', TfidfVectorizer(**tfidf_params)), ('clf', C
    h4_dt_eval = Pipeline([('tfidf', TfidfVectorizer(**tfidf_params)), ('clf', C
    h5_knn_eval = Pipeline([('tfidf', TfidfVectorizer(**tfidf_params)), ('clf', (

    eval_hypotheses = [h1_nb_eval, h2_lr_eval, h3_rf_eval, h4_dt_eval, h5_knn_

    print("\nCalculating posterior weights (based on validation F1 scores):")
    for name, model in zip(hypothesis_names, eval_hypotheses):
```

```
            print(f"  Evaluating {name}...")
            model.fit(X_train_sub, y_train_sub)
            y_val_pred = model.predict(X_val_sub)
            score = f1_score(y_val_sub, y_val_pred, average='macro')
            f1_scores.append(score)
            print(f"  {name} F1: {score:.4f}")

        # Normalize F1 scores to sum to 1
        total_f1 = sum(f1_scores)
        if total_f1 > 0:
            posterior_weights = [score / total_f1 for score in f1_scores]
        else:
            print("Warning: All models scored 0. Using equal weights.")
            posterior_weights = [0.2, 0.2, 0.2, 0.2, 0.2]

        print("\nCalculated Posterior Weights:")
        for name, weight in zip(hypothesis_names, posterior_weights):
            print(f"  {name}: {weight:.4f}")

    except Exception as e:
        print(f"Error during weight calculation: {e}. Defaulting to equal weights.")
        posterior_weights = [0.2, 0.2, 0.2, 0.2, 0.2]


    # Implement and Evaluate the Bayes Optimal Classifier
    # 'estimators' uses the original models (h1_nb, h2_lr, etc.)
    # which were trained on the FULL sampled dataset
    estimators = list(zip(hypothesis_names, hypotheses))

    # BOC is approximated using soft voting with posterior weights
    boc_soft_voter = VotingClassifier(
        estimators=estimators,
        voting='soft',
        weights=posterior_weights,
        n_jobs=-1
    )

    print("\nFitting the VotingClassifier (BOC approximation)...")
    # // TODO: Fit the VotingClassifier using the full sampled training data (X_trai
    # This .fit() call will re-fit the estimators on the provided data
    boc_soft_voter.fit(X_train_sampled, y_train_sampled)
    print("Fitting complete.")


    # Make the final BOC prediction on the test set
    print("\nPredicting on test set...")
    # // TODO: Predict y_pred using X_test, and then calculate and visualize evaluat
    if 'boc_soft_voter' in locals() and 'X_test' in locals() and not X_test.empty:
        y_pred = boc_soft_voter.predict(X_test)
    else:
        y_pred = None
```

```
# Final Evaluation (STUDENT TASK)
print("\n=== Final Evaluation: Bayes Optimal Classifier (Soft Voting) ===")

if y_pred is not None:
    # Example calculations:
    # // TODO: Generate and visualize the Confusion Matrix (heatmap) for the BOC

    # Calculate and print metrics
    accuracy = accuracy_score(y_test, y_pred)
    f1 = f1_score(y_test, y_pred, average='macro')
    print(f"Accuracy: {accuracy:.4f}")
    print(f"Macro F1 Score: {f1:.4f}")

    # Classification Report
    print("\nClassification Report (BOC):")
    print(classification_report(y_test, y_pred, target_names=target_names))

    # Confusion Matrix
    cm = confusion_matrix(y_test, y_pred, labels=target_names)
    plt.figure(figsize=(10, 7))
    sns.heatmap(cm, annot=True, fmt='d', cmap='Greens',
                xticklabels=target_names, yticklabels=target_names)
    plt.title('Confusion Matrix - Bayes Optimal Classifier (Part C)')
    plt.ylabel('True Label')
    plt.xlabel('Predicted Label')
    plt.show()

else:
    print("Evaluation skipped: Predictions not generated.")
```

```
Please enter your full SRN (e.g., PES1UG22CS345): PES1UG23CS325
Using dynamic sample size: 10325
Actual sampled training set size used: 10325

Training all base models...
Training NaiveBayes...
Training LogisticRegression...
/usr/local/lib/python3.12/dist-packages/sklearn/linear_model/_logistic.py:1247:
  warnings.warn(
Training RandomForest...
Training DecisionTree...
Training KNN...
All base models trained.

Calculating posterior weights (based on validation F1 scores):
  Evaluating NaiveBayes...
  NaiveBayes F1: 0.6288
  Evaluating LogisticRegression...
/usr/local/lib/python3.12/dist-packages/sklearn/linear_model/_logistic.py:1247:
  warnings.warn(
  LogisticRegression F1: 0.5964
  Evaluating RandomForest...
  RandomForest F1: 0.5177
  Evaluating DecisionTree...
  DecisionTree F1: 0.2767
  Evaluating KNN...
  KNN F1: 0.1692

Calculated Posterior Weights:
  NaiveBayes: 0.2873
  LogisticRegression: 0.2725
  RandomForest: 0.2365
  DecisionTree: 0.1264
  KNN: 0.0773

Fitting the VotingClassifier (BOC approximation)...
Fitting complete.

Predicting on test set...

=== Final Evaluation: Bayes Optimal Classifier (Soft Voting) ===
Accuracy: 0.7051
Macro F1 Score: 0.6104
```