AI VIET NAM
@aivietnam.edu.vn

AI VIET NAM – AI COURSE 2025

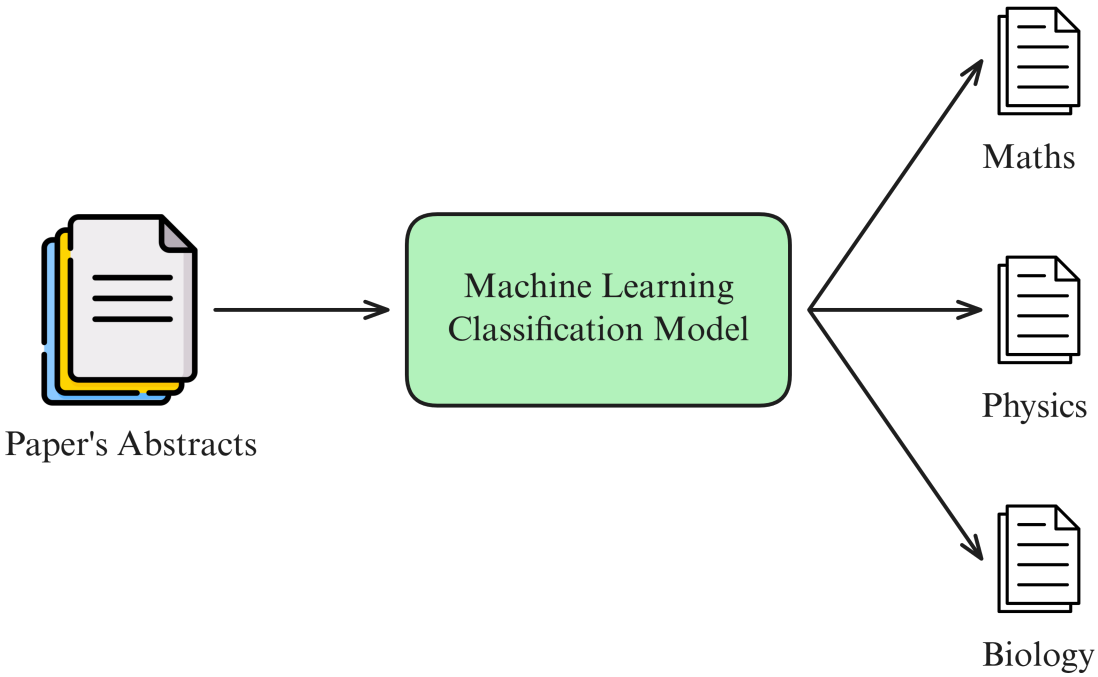
Phân loại chủ đề bài báo dựa vào kỹ thuật Ensemble Learning

Random Forest, AdaBoost, Gradient Boosting, XGBoost, LightGBM

Nguyễn Quốc Thái Trần Minh Nam Đinh Quang Vinh

I. Giới thiệu

Text Classification (Tạm dịch: Phân loại văn bản) là một trong những bài toán cơ bản và quan trọng trong lĩnh vực Xử lý ngôn ngữ tự nhiên (NLP). Mục tiêu của bài toán này là phân loại các đoạn văn bản vào các nhóm hoặc nhãn khác nhau dựa trên nội dung của chúng. Các ứng dụng phổ biến của Text Classification bao gồm phân loại email (spam vs. ham), phân loại tin tức, phân loại cảm xúc (sentiment analysis), và nhiều ứng dụng khác.



Hình 1: Minh họa ứng dụng phân loại topic của paper dựa trên abstract.

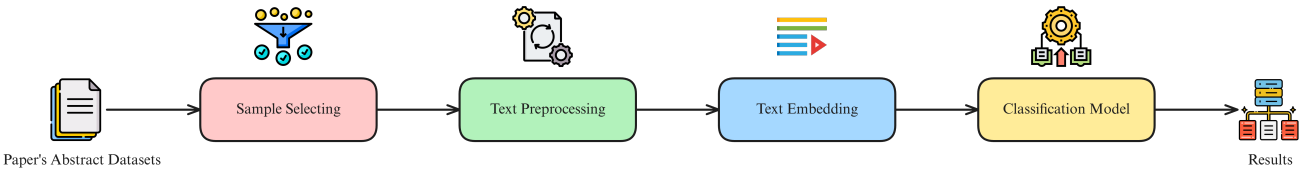
Trong project này, chúng ta sẽ xây dựng một chương trình Text Classification liên quan đến việc phân loại một abstract của publication (bài báo khoa học) thành các topic khác nhau. Chương trình sẽ được xây dựng trên nền tảng Python và sử dụng các thư viện học máy phổ biến như scikit-learn, numpy, v.v.
Theo đó, Input/Output chung của chương trình sẽ bao gồm:

- **Input:** Một abstract của publication (bài báo khoa học).
- **Output:** Topic của abstract đó (ví dụ: vật lý, toán học, khoa học máy tính, v.v.).

Mục lục

II. Xây dựng chương trình phân loại topic của publication abstract

Trong phần này, ta sẽ tiến hành cài đặt chương trình phân loại topic của publication abstract. Chương trình sẽ được xây dựng dựa trên ba phương pháp mã hóa văn bản khác nhau, bao gồm Bag-of-Words (BoW), TF-IDF và Sentence Embeddings. Mỗi phương pháp sẽ được áp dụng với một mô hình phân loại khác nhau, bao gồm Naive Bayes, KNN và Decision Tree.



Hình 2: Pipeline

II.1. Cài đặt và Import thư viện

Chúng ta import các thư viện cần thiết để thực hiện các bước tiền xử lý, trích xuất đặc trưng, huấn luyện mô hình và đánh giá mô hình. Các thư viện này bao gồm:

```python
from collections import Counter, defaultdict
from typing import List, Dict, Literal, Union

import re
import math
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

from datasets import load_dataset
from sentence_transformers import SentenceTransformer

from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix
from sklearn.feature_extraction.text import CountVectorizer, TfidfVectorizer
from sklearn.cluster import KMeans
from sklearn.neighbors import KNeighborsClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.naive_bayes import GaussianNB

import warnings

warnings.filterwarnings("ignore")

CACHE_DIR = "./cache"
```

Trước tiên, chúng ta cùng điểm qua một số thư viện chính được sử dụng trong bài:

- **re:** Thư viện để làm việc với biểu thức chính quy (regular expressions), giúp xử lý và phân tích chuỗi văn bản.
- **datasets:** Thư viện để tải và làm việc với các bộ dữ liệu từ Hugging Face.
- **sentence-transformers:** Thư viện để tạo và sử dụng các mô hình sentence embeddings, giúp chuyển đổi văn bản thành các vector ngữ nghĩa.
- **matplotlib.pyplot:** Thư viện để vẽ đồ thị và biểu đồ, hỗ trợ trực quan hóa dữ liệu.
- **seaborn:** Thư viện để vẽ đồ thị thống kê, cung cấp các hàm tiện ích để trực quan hóa dữ liệu một cách đẹp mắt.
- **numpy:** Cung cấp các đối tượng mảng đa chiều và các hàm toán học để làm việc với các mảng này.
- **scikit-learn:** Thư viện học máy phổ biến, giúp xây dựng và triển khai các mô hình học máy phức tạp một cách nhanh chóng.

II.2. Đọc và khám phá bộ dữ liệu

Trong bài này, chúng ta sẽ sử dụng bộ dữ liệu UniverseTBD/arxiv-abstracts-large được cung cấp trên HuggingFace. Bộ dữ liệu này bao gồm các abstract (tóm tắt) của các bài báo khoa học được đăng trên arXiv, một kho lưu trữ trực tuyến cho các bài báo khoa học trong nhiều lĩnh vực khác nhau. Bộ dữ liệu này có thể được sử dụng để phân loại các abstract theo các chủ đề khác nhau hoặc để phân tích nội dung của các bài báo khoa học, bao gồm các thông tin như tiêu đề, tác giả, ngày cập nhật, v.v. Bộ dữ liệu này có kích thước lớn với hơn 2 triệu bản ghi.
Chúng ta load bộ dữ liệu này bằng thư viện `datasets` với hàm `load_dataset` và in ra một số thông tin cơ bản về bộ dữ liệu như sau:

```
ds = load_dataset("UniverseTBD/arxiv-abstracts-large")
ds
```

Nội dung được in ra màn hình như sau:

```
DatasetDict(
train: Dataset(
features: ['id', 'submitter', 'authors', 'title', 'comments', 'journal-ref', 'doi', 'report-no',
'categories', 'license', 'abstract', 'versions', 'update_date', 'authors_parsed'],
num_rows: 2292057
)
)
```

Các fields của bộ dữ liệu bao gồm:

- **id:** ID của paper trên arXiv.

- **submitter:** Người nộp bài báo.
- **authors:** Danh sách các tác giả của bài báo.
- **title:** Tiêu đề của bài báo.
- **comments:** Các bình luận liên quan đến bài báo.
- **journal-ref:** Tham chiếu đến tạp chí nơi bài báo được xuất bản.
- **doi:** Digital Object Identifier, mã định danh duy nhất cho bài báo.
- **report-no:** Số báo cáo liên quan đến bài báo.
- **categories:** Các chủ đề hoặc lĩnh vực mà bài báo thuộc về.
- **license:** Giấy phép sử dụng của bài báo.
- **abstract:** Tóm tắt nội dung của bài báo (đây là field chúng ta sẽ sử dụng để phân loại).
- **versions:** Phiên bản của bài báo.
- **update_date:** Ngày cập nhật của bài báo.
- **authors_parsed:** Danh sách các tác giả đã được phân tích và chuẩn hóa.

Chúng ta sẽ sử dụng field `abstract` để làm input cho mô hình phân loại, và field `categories` để làm nhãn (label) cho mô hình.
Để quan sát dữ liệu, chúng ta in ra màn hình 3 bản ghi đầu tiên của bộ dữ liệu như sau:

```
# print three first examples
for i in range(3):
    print(f"Example {i+1}:")
    print(ds['train'][i]['abstract'])
    print(ds['train'][i]['categories'])
    print("---" * 20)
```

Dựa vào dữ liệu ta thấy dữ liệu abstract của chúng ta có chứa nhiều kí tự đặc biệt như dấu chấm, dấu phẩy, dấu ngoặc kép, v.v. Ngoài ra, các abstract này cũng có độ dài khác nhau, có thể từ vài câu đến vài đoạn văn. Các nhãn (categories) của các abstract này cũng rất đa dạng, bao gồm nhiều lĩnh vực khác nhau như vật lý, toán học, khoa học máy tính, v.v. Chúng ta in toàn bộ các categories của abstract để xem xét các nhãn này có phù hợp với bài toán phân loại hay không như sau:

```
all_categories = ds['train']['categories']
print(set(all_categories))
```

Categories của abstract tuân theo format `<primary category> <secondary category> ...`, ví dụ như `hep-ph` hay `math.CO cs.CG`. Chúng ta sẽ sử dụng field `abstract` để làm input cho

mô hình phân loại, và phần primary category (category đầu tiên) trong field `categories` để làm nhãn (label) cho mô hình.
Để quan sát số lượng primary categories trong bộ dữ liệu, chúng ta sẽ đếm số lượng các nhãn này và in ra như sau:

```
all_categories = ds['train']['categories']
category_set = set()

# Collect unique labels
for category in all_categories:
    parts = category.split(' ')
    for part in parts:
        topic = part.split('.')[0]
        category_set.add(topic)

# Sort the labels and print them
sorted_categories= sorted(list(category_set), key=lambda x: x.lower())
print(f'There are {len(sorted_categories)} unique primary categories in the dataset:')
for category in sorted_categories:
    print(category)
```

Kết quả in ra sẽ là danh sách các primary categories trong bộ dữ liệu như sau:

```
There are 38 unique primary categories in the dataset:
acc-phys
adap-org
alg-geom
ao-sci
astro-ph
atom-ph
bayes-an
chao-dyn
chem-ph
cmp-lg
comp-gas
cond-mat
cs
...
```

Chúng ta sẽ lấy 1000 samples, các samples này có primary là một trong 5 categories sau: `astro-ph`, `cond-mat`, `cs`, `math`, `physics`.

```
# load 1000 samples with single label belonging to specific categories
samples = []
CATEGORIES_TO_SELECT = ['astro-ph', 'cond-mat', 'cs', 'math', 'physics']
for s in ds['train']:
    if len(s['categories'].split(' ')) != 1:
```

```
        continue

    cur_category = s['categories'].strip().split('.')[0]
    if cur_category not in CATEGORIES_TO_SELECT:
        continue

    samples.append(s)

    if len(samples) >= 1000:
        break
print(f"Number of samples: {len(samples)}") # Number of samples: 1000
```

II.3. Tiền xử lý dữ liệu

Tiền xử lý dữ liệu đặc trưng: Nội dung của abstract có chứa nhiều kí tự đặc biệt (newline, trailing space, v.v.), viết hoa, cũng như dấu câu. Do đó, chúng ta cần thực hiện các bước tiền xử lý để chuẩn hóa dữ liệu.
Đối với mỗi sample trong danh sách 1000 samples chúng ta đã chọn ra, chúng ta sẽ thực hiện các bước tiền xử lý như sau:

- Loại bỏ các kí tự `\n` và khoảng trắng ở đầu và cuối chuỗi.
- Loại bỏ các kí tự đặc biệt (dấu câu, kí tự không phải chữ cái hoặc số).
- Loại bỏ các chữ số.
- Chuyển đổi tất cả các chữ cái thành chữ thường (lowercase).
- Lấy nhãn (label) là primary category (phần đầu tiên) trong field `categories`.

Các bước tiền xử lý trên tương ứng với đoạn code sau:

```
preprocessed_samples = []
for s in samples:
    abstract = s['abstract']

    # Remove \n characters in the middle and leading/trailing spaces
    abstract = abstract.strip().replace("\n", " ")

    # Remove special characters
    abstract = re.sub(r'[^\w\s]', '', abstract)

    # Remove digits
    abstract = re.sub(r'\d+', '', abstract)

    # Remove extra spaces
    abstract = re.sub(r'\s+', ' ', abstract).strip()

    # Convert to lower case
    abstract = abstract.lower()
```

```
    # for the label, we only keep the first part
    parts = s['categories'].split(' ')
    category = parts[0].split('.')[0]

    preprocessed_samples.append({
        "text": abstract,
        "label": category
    })
```

Tiếp theo, chúng ta cần tạo hai dictionaries để lưu trữ các primary categories dưới dạng số và ngược lại bằng đoạn code sau:

```
label_to_id = {label: i for i, label in enumerate(sorted_labels)}
id_to_label = {i: label for i, label in enumerate(sorted_labels)}

# Print label to ID mapping
print("Label to ID mapping:")
for label, id_ in label_to_id.items():
    print(f"{label} --> {id_}")
```

Kết quả in ra sẽ là danh sách các nhãn và ID tương ứng như sau:

```
Label to ID mapping:
astro-ph -> 0
cond-mat -> 1
cs -> 2
math -> 3
physics -> 4
```

Cuối cùng, chúng ta sẽ chia dữ liệu trên thành 2 tập: tập huấn luyện (train) và tập kiểm tra (test) với tỉ lệ 80% cho tập huấn luyện và 20% cho tập kiểm tra. Chúng ta sẽ sử dụng hàm `train_test_split` từ thư viện `sklearn` để thực hiện việc này.

```
X_full = [sample['text'] for sample in preprocessed_samples]
y_full = [label_to_id[sample['label']] for sample in preprocessed_samples]

X_train, X_test, y_train, y_test = train_test_split(X_full, y_full, test_size=0.2,
                                                    random_state=42, stratify=y_full)

print(f"Training samples: {len(X_train)}")
print(f"Test samples: {len(X_test)}")
```

Số lượng mẫu trong tập huấn luyện và tập kiểm tra sẽ được in ra như sau:

Training samples: 800
Test samples: 200

II.4. Mã hóa văn bản

Văn bản của chúng ta sau khi tiền xử lý sẽ được mã hóa thành các vector số để có thể sử dụng trong mô hình học máy. Có nhiều phương pháp để mã hóa văn bản, nhưng trong bài này, chúng ta sẽ sử dụng từng phương pháp trong ba phương pháp sau và so sánh chúng với nhau: *Bag-of-Words* (BoW), *TF-IDF* (Term Frequency-Inverse Document Frequency), và *Sentence Embeddings*.

II.4.1. Bag-of-Words (BoW)

Bag-of-Words (BoW) là một kỹ thuật biểu diễn văn bản đơn giản và thường được sử dụng. Nó chuyển đổi văn bản thành một vector có độ dài cố định bằng cách đếm số lần xuất hiện của mỗi từ trong văn bản. Phương pháp này bỏ qua ngữ pháp và thứ tự từ nhưng vẫn giữ lại tần suất của các từ.
Ví dụ về cách sử dụng BoW để mã hóa văn bản với class `CountVectorizer` từ thư viện `sklearn` như sau:

```python
docs = [
    "I am going to school to study for the final exam.",
    "The weather is nice today and I feel happy.",
    "I love programming in Python and exploring new libraries.",
    "Data science is an exciting field with many opportunities.",
]

bow = CountVectorizer()
vectors = bow.fit_transform(docs)

for i, vec in enumerate(vectors):
    print(f"Document {i+1}: {vec.toarray()}")
```

Đoạn code trên khai báo một danh sách các văn bản (docs) và sử dụng class `CountVectorizer` để mã hóa chúng thành các vector BoW với hàm `fit_transform`. Sau đó, nó in ra các vector tương ứng với từng văn bản như sau:

Vector BoW khi sử dụng CountVectorizer

Document 1: [[1 0 0 0 1 0 0 0 0 1 1 1 0 0 0 0 0 0 0 0 0 0 1 0 1 1 2 0 0 0]]
Document 2: [[0 0 1 0 0 0 0 1 0 0 0 0 1 0 1 0 0 0 0 1 0 0 0 0 0 0 1 0 1 1 0]]
Document 3: [[0 0 1 0 0 0 1 0 0 0 0 0 0 1 0 1 1 0 1 0 0 1 1 0 0 0 0 0 0 0 0]]
Document 4: [[0 1 0 1 0 1 0 0 1 0 0 0 0 0 1 0 0 1 0 0 1 0 0 0 1 0 0 0 0 0 1]]

Chúng ta thấy được mỗi văn bản được mã hóa thành một vector với kích thước bằng với số lượng từ trong từ điển. Mỗi phần tử trong vector tương ứng với tần suất xuất hiện của từ đó trong văn bản.

II.4.2. TF-IDF

TF-IDF (Term Frequency-Inverse Document Frequency) là một phương pháp mã hóa văn bản nâng cao hơn BoW, nó không chỉ tính tần suất của từ trong văn bản mà còn xem xét tần suất của từ trong toàn bộ tập dữ liệu. Điều này giúp giảm trọng số của các từ phổ biến và tăng trọng số của các từ hiếm gặp, từ đó cải thiện khả năng phân loại. Ví dụ về cách sử dụng TF-IDF để mã hóa văn bản với class `TfidfVectorizer` từ thư viện `sklearn` như sau:

```
docs = [
    "I am going to school to study for the final exam.",
    "The weather is nice today and I feel happy.",
    "I love programming in Python and exploring new libraries.",
    "Data science is an exciting field with many opportunities.",
]

vectorizer = TfidfVectorizer()
tfidf_vectors = vectorizer.fit_transform(docs)

for i, vec in enumerate(tfidf_vectors):
    print(f"TF-IDF for Document {i+1}:")
    print(vec.toarray())
```

Đoạn code trên khai báo một danh sách các văn bản (docs) và sử dụng class `TfidfVectorizer` để mã hóa chúng thành các vector TF-IDF với hàm `fit_transform`. Sau đó, nó in ra các vector tương ứng với từng văn bản như sau:

Vector TF-IDF khi sử dụng TfidfVectorizer

TF-IDF for Document 1:

[[0.29333722 0. 0. 0. 0.29333722 0. 0. 0. 0. 0.29333722 0.29333722 0.29333722 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.29333722 0. 0.29333722 0.23127044 0.58667444 0. 0. 0.]]

TF-IDF for Document 2:

[[0. 0. 0.30091213 0. 0. 0. 0. 0.38166888 0. 0. 0. 0. 0.38166888 0. 0.30091213 0. 0. 0. 0. 0.38166888 0. 0. 0. 0. 0. 0. 0.30091213 0. 0.38166888 0.38166888 0.]]

TF-IDF for Document 3:

[[0. 0. 0.2855815 0. 0. 0. 0.36222393 0. 0. 0. 0. 0. 0. 0.36222393 0. 0.36222393 0.36222393 0. 0.36222393 0. 0. 0.36222393 0.36222393 0. 0. 0. 0. 0. 0. 0. 0.]]

TF-IDF for Document 4:

[[0. 0.34056989 0. 0.34056989 0. 0.34056989 0. 0. 0.34056989 0. 0. 0. 0. 0. 0.26850921 0. 0. 0.34056989 0. 0. 0.34056989 0. 0. 0. 0.34056989 0. 0. 0. 0. 0. 0.34056989]]

Khác với BoW, các vector TF-IDF có trọng số khác nhau cho từng từ, do đó các vector này biểu diễn từng trọng số với kiểu số thực (float) thay vì số nguyên (int). Điều này giúp mô hình phân loại có thể nhận biết được tầm quan trọng của từng từ trong văn bản.

II.4.3. Sentence Embeddings

Sentence Embeddings là một phương pháp mã hóa văn bản nâng cao hơn, nó chuyển đổi toàn bộ câu hoặc đoạn văn thành một vector có kích thước cố định. Các vector này thường được huấn luyện trên các tập dữ liệu lớn và có thể nắm bắt được ngữ nghĩa của câu, từ đó cải thiện khả năng phân loại.
Chúng ta sẽ implement class `EmbeddingVectorizer` để mã hóa văn bản thành các vector embeddings. Class này sẽ sử dụng mô hình pre-trained từ thư viện `sentence-transformers` để chuyển đổi văn bản thành vector.

Xây dựng EmbeddingVectorizer để mã hóa văn bản

```python
class EmbeddingVectorizer:
    def __init__(
        self,
        model_name: str = 'intfloat/multilingual-e5-base',
        normalize: bool = True
    ):
        self.model = SentenceTransformer(model_name)
        self.normalize = normalize

    def _format_inputs(
        self,
        texts: List[str],
        mode: Literal['query', 'passage']
    ) -> List[str]:
        if mode not in {"query", "passage"}:
            raise ValueError("Mode must be either 'query' or 'passage'")
        return [f"{mode}: {text.strip()}" for text in texts]

    def transform(
        self,
        texts: List[str],
        mode: Literal['query', 'passage'] = 'query'
    ) -> List[List[float]]:
        if mode == 'raw':
            inputs = texts
        else:
            inputs = self._format_inputs(texts, mode)

        embeddings = self.model.encode(inputs, normalize_embeddings=self.normalize)
        return embeddings.tolist()

    def transform_numpy(
        self,
        texts: List[str],
        mode: Literal['query', 'passage'] = 'query'
    ) -> np.ndarray:
```

```
        return np.array(self.transform(texts, mode=mode))
```

Class `EmbeddingVectorizer` được xây dựng với các methods sau:

- `__init__`: Khởi tạo mô hình SentenceTransformer với tên mô hình và tùy chọn chuẩn hóa.
- `_format_inputs`: Định dạng đầu vào cho mô hình, thêm tiền tố "query"hoặc "passage"vào văn bản.
- `transform`: Chuyển đổi danh sách văn bản thành các vector embeddings.
- `transform_numpy`: Chuyển đổi danh sách văn bản thành các vector embeddings dưới dạng numpy array.

Tương tự hai phương pháp trên, chúng ta sẽ sử dụng class `EmbeddingVectorizer` để mã hóa văn bản trong bộ dữ liệu của mình. Đoạn code sau sẽ thực hiện việc này:

```
docs = [
    "I am going to school to study for the final exam.",
    "The weather is nice today and I feel happy.",
    "I love programming in Python and exploring new libraries.",
    "Data science is an exciting field with many opportunities.",
]

vectorizer = EmbeddingVectorizer()
embeddings = vectorizer.transform(docs)

for i, emb in enumerate(embeddings):
    print(f"Embedding for Document {i+1}:")
    print(emb[:3])  # Print first 10 dimensions for brevity
```

Đoạn code trên sẽ mã hóa các văn bản trong danh sách `docs` thành các vector embeddings và in ra 10 chiều đầu tiên của mỗi vector như sau:

Vector Embeddings khi sử dụng EmbeddingVectorizer

Embedding for Document 1:
-0.014805682934820652, 0.031276583671569824, -0.01615864224731922
Embedding for Document 2:
0.011917386204004288, 0.03327357769012451, -0.025739021599292755
Embedding for Document 3:
0.012662884779274464, 0.03936118632555008, -0.024181174114346504
Embedding for Document 4:
0.0063935937359929085, 0.04922199621796608, -0.028402965515851974

Các vectors này tương tự như các vectors TF-IDF, nhưng chúng có kích thước cố định và có thể nắm bắt được ngữ nghĩa của câu. Bên cạnh đó, có rất ít từ trong các vectors này có giá trị bằng 0, điều này cho thấy các vectors embeddings có thể biểu diễn tốt hơn ngữ nghĩa của văn bản so với các phương pháp mã hóa khác.

II.4.4. Khai báo và khởi tạo các vectorizers

Chúng ta sẽ khai báo và khởi tạo các vectorizers và huấn luyện chúng trên tập dữ liệu của chúng ta cho từng phương pháp mã hóa văn bản đã đề cập ở trên. Đoạn code sau sẽ thực hiện việc này:

```
bow_vectorizer = CountVectorizer()
X_train_bow = # Your code here
X_test_bow = bow_vectorizer.transform(X_test)

tfidf_vectorizer = TfidfVectorizer()
X_train_tfidf = # Your code here
X_test_tfidf = tfidf_vectorizer.transform(X_test)

embedding_vectorizer = EmbeddingVectorizer()
X_train_embeddings = # Your code here
X_test_embeddings = embedding_vectorizer.transform(X_test)

# convert all to numpy arrays for consistency
X_train_bow, X_test_bow = np.array(X_train_bow.toarray()), np.array(X_test_bow.toarray
                                                    ())
X_train_tfidf, X_test_tfidf = np.array(X_train_tfidf.toarray()), np.array(X_test_tfidf.
                                                   toarray())
X_train_embeddings, X_test_embeddings = np.array(X_train_embeddings), np.array(
                                                   X_test_embeddings)

# Print shapes of the transformed datasets
print(f"Shape of X_train_bow: {X_train_bow.shape}")
print(f"Shape of X_test_bow: {X_test_bow.shape}\n")
print(f"Shape of X_train_tfidf: {X_train_tfidf.shape}")
print(f"Shape of X_test_tfidf: {X_test_tfidf.shape}\n")
print(f"Shape of X_train_embeddings: {X_train_embeddings.shape}")
print(f"Shape of X_test_embeddings: {X_test_embeddings.shape}\n")
```

Kết quả in ra màn hình lần lượt sẽ là kích thước của các tập dữ liệu đã được mã hóa:

Kích thước của các tập dữ liệu đã được mã hóa

Shape of X_train_bow: (800, 10373)
Shape of X_test_bow: (200, 10373)

Shape of X_train_tfidf: (800, 10373)
Shape of X_test_tfidf: (200, 10373)

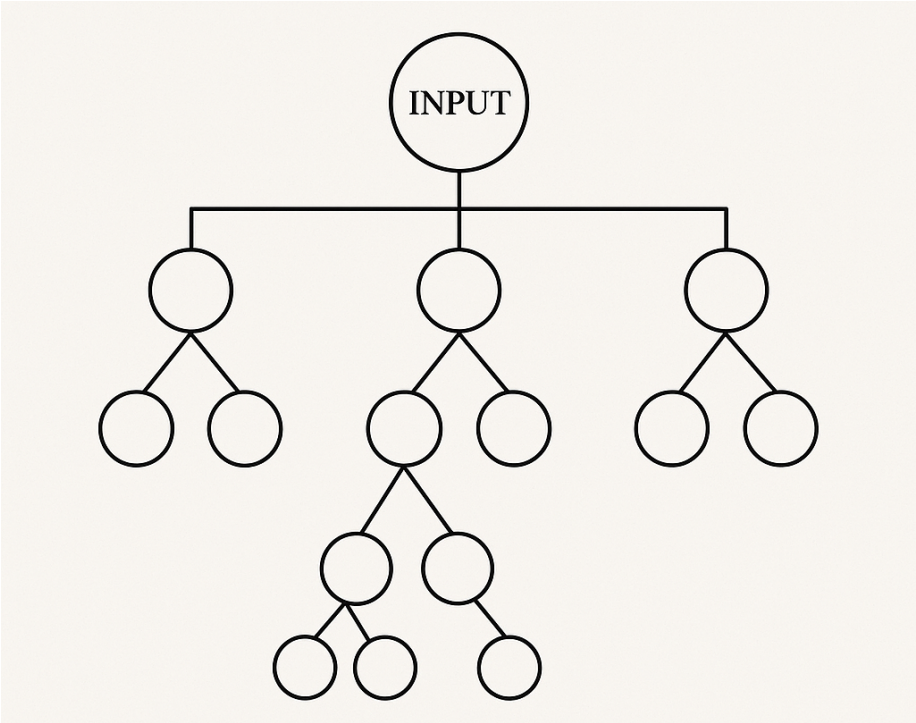
Shape of X_train_embeddings: (800, 768)
Shape of X_test_embeddings: (200, 768)

Chúng ta thấy rằng các vector BoW và TF-IDF có kích thước bằng nhau, tương ứng với số lượng từ trong từ điển (10373 từ), trong khi các vector embeddings có kích thước cố định là 768, cho thấy chúng được mã hóa thành các vector có kích thước nhỏ hơn nhiều so với BoW và TF-IDF.

II.5. Huấn luyện và đánh giá mô hình phân loại

II.5.1. Random Forest

Random Forest là một thuật toán học máy mạnh mẽ, nó cải thiện độ chính xác dự đoán bằng cách kết hợp đầu ra của nhiều decision trees độc lập từ các mẫu ngẫu nhiên của dữ liệu huấn luyện thông qua quá trình bootstrapping. Khi đưa ra dự đoán, Random Forest sẽ tổng hợp kết quả của các decision trees này - bằng majority vote cho các bài toán phân loại hoặc bằng cách lấy trung bình cho các bài toán hồi quy.



Hình 3: Random Forest Algorithm

Tương tự như các mô hình trước, chúng ta sẽ thiết kế một hàm `train_and_test_random_forest` để huấn luyện mô hình Random Forest và in ra các kết quả phân loại như sau:

Hàm `train_and_test_random_forest` để huấn luyện mô hình Random Forest

```python
def train_and_test_random_forest(X_train, y_train, X_test, y_test, n_estimators: int =
                                 100):
    rf = RandomForestClassifier(n_estimators=n_estimators, random_state=42)
    rf.fit(X_train, y_train)

    # Predict on the test set
    y_pred = rf.predict(X_test)

    # Calculate accuracy and classification report
    accuracy = accuracy_score(y_test, y_pred)
    report = classification_report(y_test, y_pred, target_names=sorted_labels,
                                   output_dict=True)
```

```
    return y_pred, accuracy, report
```

Sau đó, chúng ta sẽ chạy hàm `train_and_test_random_forest` cho từng phương pháp mã hóa và in ra kết quả như sau:

Huấn luyện mô hình Random Forest cho các phương pháp mã hóa

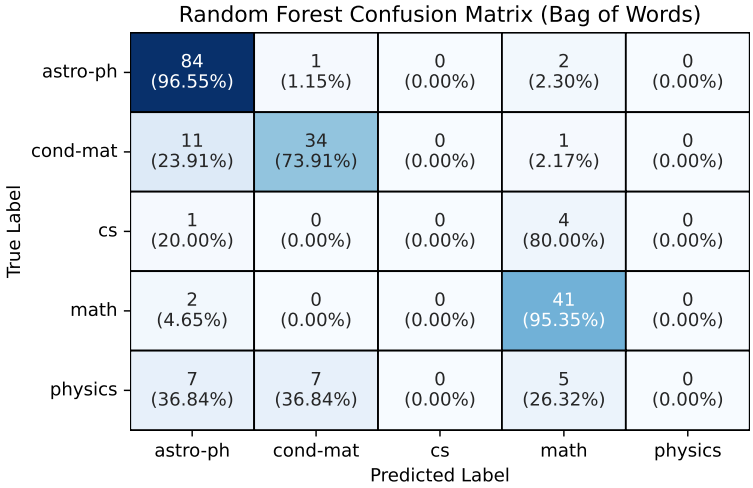
```
rf_bow_labels, rf_bow_accuracy, rf_bow_report = train_and_test_random_forest(
                                                X_train_bow, y_train, X_test_bow, y_test)
rf_tfidf_labels, rf_tfidf_accuracy, rf_tfidf_report = train_and_test_random_forest(
                                                X_train_tfidf, y_train, X_test_tfidf,
                                                y_test)
rf_embeddings_labels, rf_embeddings_accuracy, rf_embeddings_report =
                                                train_and_test_random_forest(
                                                X_train_embeddings, y_train,
                                                X_test_embeddings, y_test)

# Print Random Forest results
print("Accuracies for Random Forest:")
print(f"Bag of Words: {rf_bow_accuracy:.4f}")
print(f"TF-IDF: {rf_tfidf_accuracy:.4f}")
print(f"Embeddings: {rf_embeddings_accuracy:.4f}")
```

Và đây là kết quả độ chính xác của mô hình Random Forest cho từng phương pháp mã hóa:

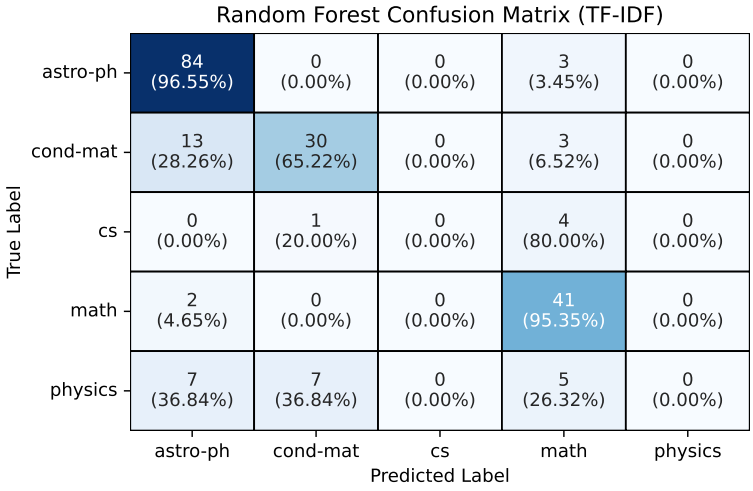
Độ chính xác của mô hình Random Forest cho từng phương pháp mã hóa

Accuracies for Random Forest:
Bag of Words: 0.7950
TF-IDF: 0.7750
Embeddings: 0.8550

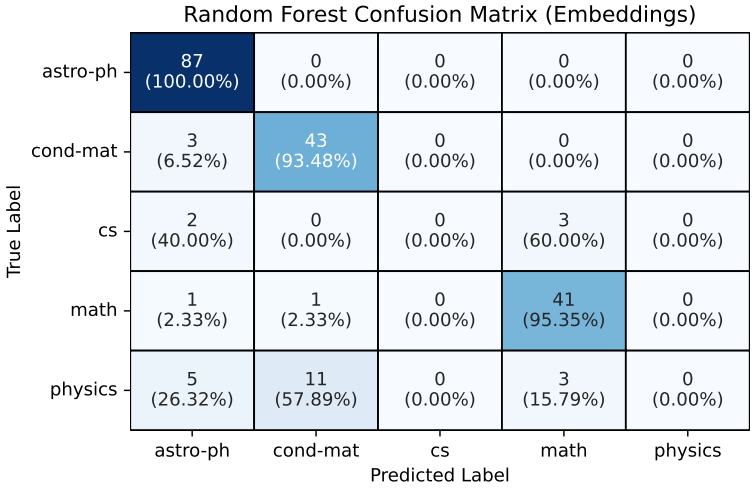
Trong đó, mô hình Random Forest với phương pháp mã hóa embeddings đạt độ chính xác cao nhất là 85.5%, trong khi phương pháp BoW và TF-IDF đạt độ chính xác lần lượt là 79.5% và 77.5%. Điều này cho thấy mô hình Random Forest có thể hoạt động tốt với các phương pháp mã hóa khác nhau, nhưng vẫn có xu hướng hoạt động tốt hơn với phương pháp mã hóa embeddings (giống với các mô hình đã thí nghiệm phía trên).
Cuối cùng, chúng ta có confusion matrices cho từng phương pháp mã hóa trong Hình 4.



(a) Confusion Matrix Random Forest với BoW



(b) Confusion Matrix Random Forest với TF-IDF



(c) Confusion Matrix Random Forest với Embeddings

Hình 4: Confusion Matrix cho từng phương pháp mã hóa với mô hình Random Forest

II.5.2. AdaBoost

AdaBoost, hay Adaptive Boosting, là một thuật toán học máy kết hợp nhiều mô hình học đơn giản và "yếu"để tạo ra một mô hình "mạnh"có độ chính xác cao. Nó hoạt động bằng cách huấn luyện lặp đi lặp lại các mô hình yếu này, với mỗi mô hình tiếp theo tập trung nhiều hơn vào các ví dụ bị phân loại sai từ các mô hình trước đó bằng cách tăng trọng số của chúng. Dự đoán cuối cùng là trung bình có trọng số của các dự đoán từ tất cả các mô hình yếu.
Chúng ta xây dựng hàm huấn luyện và kiểm thử mô hình AdaBoost trên bộ dữ liệu của chúng ta như sau:

Hàm `train_and_test_adaboost` để huấn luyện mô hình AdaBoost

```python
def train_and_test_adaboost(X_train, y_train, X_test, y_test, n_estimators: int = 50):
    ada = AdaBoostClassifier(n_estimators=n_estimators, random_state=42)
    ada.fit(X_train, y_train)

    # Predict on the test set
    y_pred = ada.predict(X_test)

    # Calculate accuracy and classification report
    accuracy = accuracy_score(y_test, y_pred)
    report = classification_report(y_test, y_pred, target_names=sorted_labels,
                                   output_dict=True)

    return y_pred, accuracy, report
```

Sau đó, chúng ta sẽ chạy hàm `train_and_test_adaboost` cho từng phương pháp mã hóa và in ra kết quả như sau:

Huấn luyện mô hình AdaBoost cho các phương pháp mã hóa

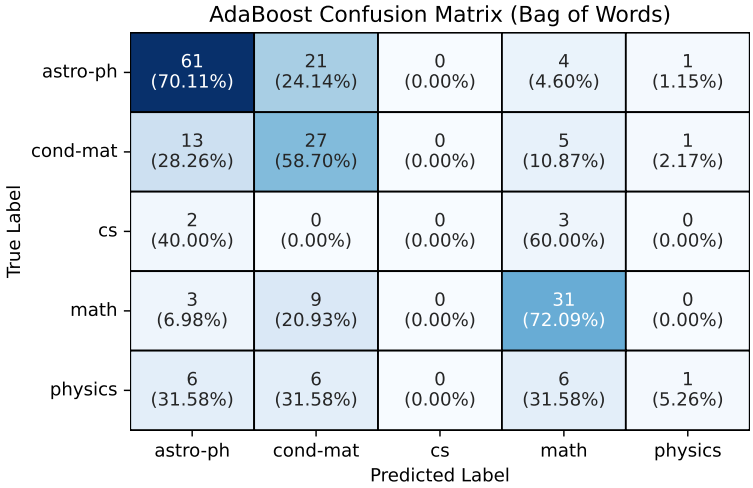
```
ada_bow_labels, ada_bow_accuracy, ada_bow_report = train_and_test_adaboost(X_train_bow,
                                                    y_train, X_test_bow, y_test)
ada_tfidf_labels, ada_tfidf_accuracy, ada_tfidf_report = train_and_test_adaboost(
                                                   X_train_tfidf, y_train, X_test_tfidf,
                                                   y_test)
ada_embeddings_labels, ada_embeddings_accuracy, ada_embeddings_report =
                                                   train_and_test_adaboost(X_train_embeddings,
                                                    y_train, X_test_embeddings, y_test)

# Print AdaBoost results
print("Accuracies for AdaBoost:")
print(f"Bag of Words: {ada_bow_accuracy:.4f}")
print(f"TF-IDF: {ada_tfidf_accuracy:.4f}")
print(f"Embeddings: {ada_embeddings_accuracy:.4f}")
```

Và đây là kết quả độ chính xác của mô hình AdaBoost cho từng phương pháp mã hóa:

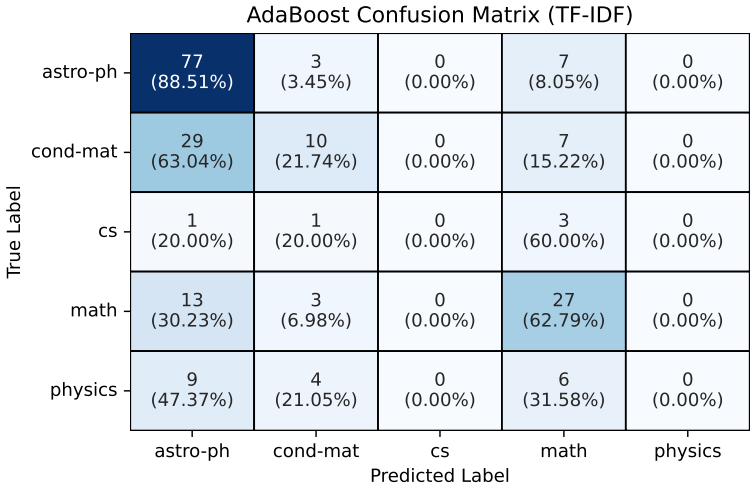
Độ chính xác của mô hình AdaBoost cho từng phương pháp mã hóa

Accuracies for AdaBoost:
Bag of Words: 0.6000
TF-IDF: 0.5700
Embeddings: 0.6200

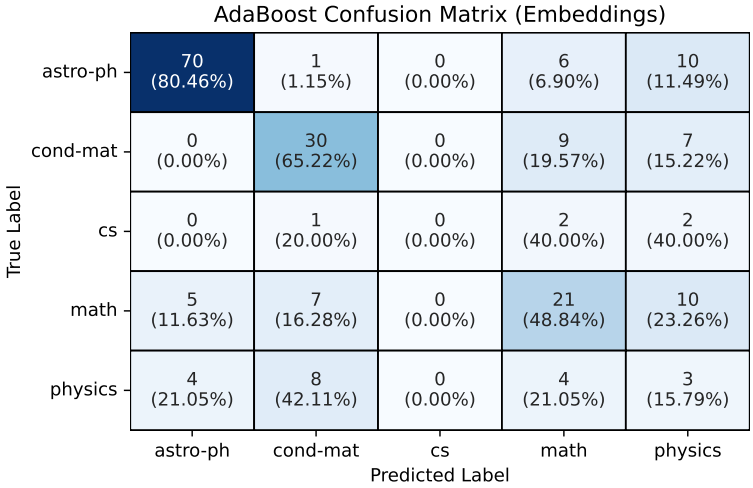
Chúng ta có thể thấy kết quả đánh giá mô hình AdaBoost không được như mong đợi, với độ chính xác chỉ đạt khoảng 60% cho phương pháp BoW và embeddings, và thấp hơn một chút với TF-IDF là 57%. Kết quả này thấp hơn các mô hình machine learning trước đó như Decision Tree, Naive Bayes và SVM. Để cải thiện, chúng ta có thể tối ưu các hyperparameters của mô hình AdaBoost hoặc thử các thuật toán boosting khác như Gradient Boosting hoặc XGBoost. Cuối cùng, chúng ta có confusion matrices cho từng phương pháp mã hóa trong Hình 5.



(a) Confusion Matrix AdaBoost với BoW



(b) Confusion Matrix AdaBoost với TF-IDF



(c) Confusion Matrix AdaBoost với Embeddings

Hình 5: Confusion Matrix cho từng phương pháp mã hóa với mô hình AdaBoost

II.5.3. Gradient Boosting

Gradient boosting là một kỹ thuật học máy tập hợp, xây dựng các cây quyết định (hoặc các mô hình yếu khác) theo trình tự, với mỗi cây mới sửa lỗi của các cây trước đó để tạo thành một mô hình dự đoán mạnh mẽ tổng thể. Nó hoạt động bằng cách lặp đi lặp lại việc tối thiểu hóa một hàm mất mát, sử dụng gradient để hướng dẫn việc huấn luyện mỗi mô hình con mới. Việc sử dụng gradient boosting có thể tốn kém về mặt tính toán và dễ bị overfitting, đòi hỏi việc điều chỉnh hyperparameter cẩn thận.
Chúng ta xây dựng hàm huấn luyện và kiểm thử mô hình Gradient Boosting trên bộ dữ liệu của chúng ta như sau:

Hàm `train_and_test_gradient_boosting` để huấn luyện mô hình Gradient Boosting

```
def train_and_test_gradient_boosting(X_train, y_train, X_test, y_test, n_estimators:
                                     int = 100):
    gb = GradientBoostingClassifier(n_estimators=n_estimators, random_state=42)
    gb.fit(X_train, y_train)

    # Predict on the test set
    y_pred = gb.predict(X_test)

    # Calculate accuracy and classification report
    accuracy = accuracy_score(y_test, y_pred)
    report = classification_report(y_test, y_pred, target_names=sorted_labels,
                                   output_dict=True)

    return y_pred, accuracy, report
```

Sau đó, chúng ta sẽ chạy hàm `train_and_test_gradient_boosting` cho từng phương pháp mã hóa và in ra kết quả như sau:

Huấn luyện mô hình Gradient Boosting cho các phương pháp mã hóa

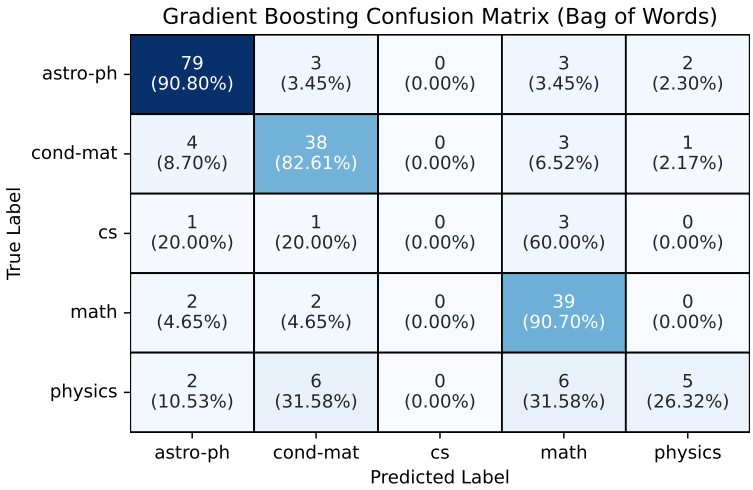
```
gb_bow_labels, gb_bow_accuracy, gb_bow_report = train_and_test_gradient_boosting(
                                                X_train_bow, y_train, X_test_bow, y_test)
gb_tfidf_labels, gb_tfidf_accuracy, gb_tfidf_report = train_and_test_gradient_boosting(
                                                X_train_tfidf, y_train, X_test_tfidf,
                                                y_test)
gb_embeddings_labels, gb_embeddings_accuracy, gb_embeddings_report =
                                                train_and_test_gradient_boosting(
                                                X_train_embeddings, y_train,
                                                X_test_embeddings, y_test)

# Print Gradient Boosting results
print("Accuracies for Gradient Boosting:")
print(f"Bag of Words: {gb_bow_accuracy:.4f}")
print(f"TF-IDF: {gb_tfidf_accuracy:.4f}")
print(f"Embeddings: {gb_embeddings_accuracy:.4f}")
```

Và đây là kết quả độ chính xác của mô hình Gradient Boosting cho từng phương pháp mã hóa:

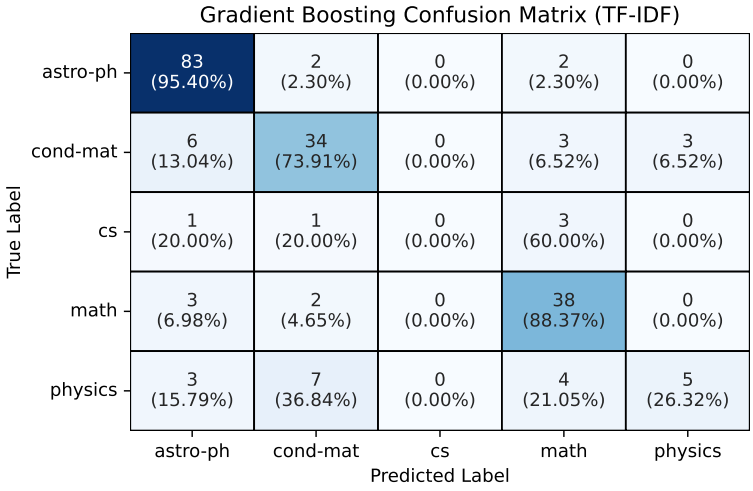
Độ chính xác của mô hình Gradient Boosting cho từng phương pháp mã hóa

```
Accuracies for Gradient Boosting:
Bag of Words: 0.8050
TF-IDF: 0.8000
Embeddings: 0.8650
```

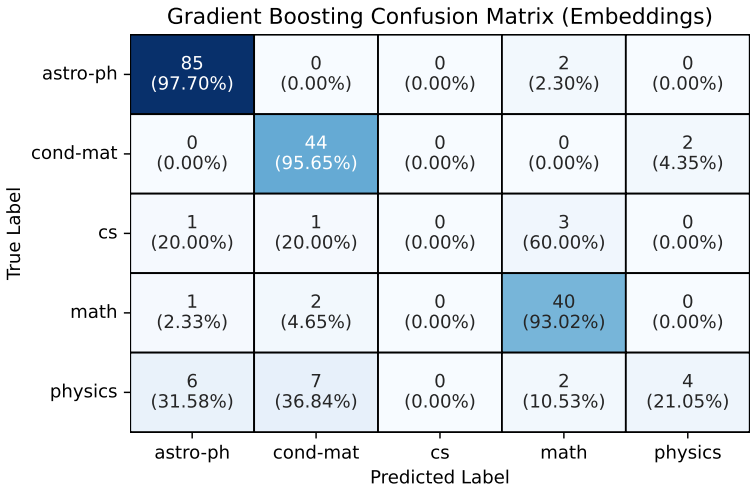
Kết quả đánh giá mô hình Gradient Boosting cho thấy độ chính xác đạt khoảng 80.5% với phương pháp BoW, 80% với TF-IDF và cao nhất là 86.5% với embeddings. Kết quả này khá tương đồng với các mô hình machine learning trước đó như Decision Tree và SVM, đặc biệt là khi sử dụng embeddings. Để cải thiện hơn nữa, chúng ta có thể tối ưu các hyperparameters của mô hình Gradient Boosting hoặc thử các thuật toán boosting khác như XGBoost hoặc LightGBM.
Cuối cùng, chúng ta có confusion matrices cho từng phương pháp mã hóa trong Hình 6.



(a) Confusion Matrix Gradient Boosting với BoW



(b) Confusion Matrix Gradient Boosting với TF-IDF



(c) Confusion Matrix Gradient Boosting với Embeddings

Hình 6: Confusion Matrix cho từng phương pháp mã hóa với mô hình Gradient Boosting

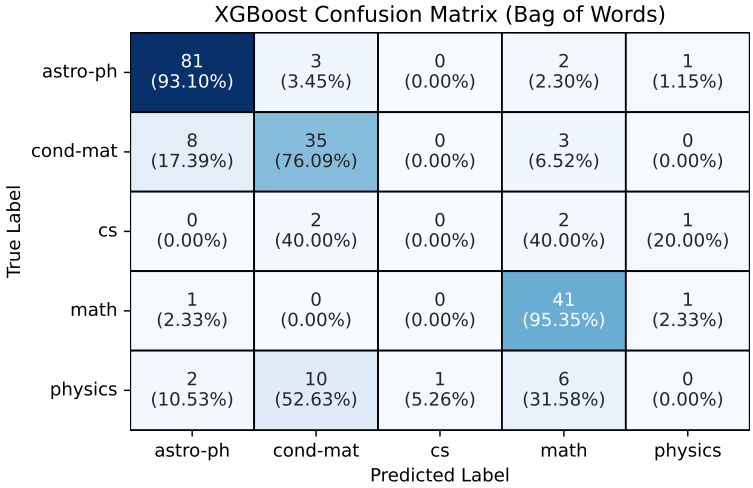
II.5.4. XGBoost

XGBoost là một thư viện học máy mã nguồn mở, được sử dụng rộng rãi vì cách triển khai tối ưu hóa và mở rộng của thuật toán gradient boosting cho các tác vụ như phân loại, hồi quy và xếp hạng. XGBoost nổi tiếng nhờ tốc độ, độ chính xác, cũng như khả năng xử lý các tập dữ liệu lớn thông qua xử lý song song và phân tán.
Chúng ta xây dựng hàm huấn luyện và kiểm thử mô hình XGBoost như sau:

Hàm `train_and_test_xgboost` để huấn luyện mô hình XGBoost

```
def train_and_test_xgboost(X_train, y_train, X_test, y_test, n_estimators: int = 100):
    xgb = XGBClassifier(
        n_estimators=n_estimators, use_label_encoder=False, eval_metric='mlogloss',
                                            random_state=42
    )
    xgb.fit(X_train, y_train)

    # Predict on the test set
    y_pred = xgb.predict(X_test)

    # Calculate accuracy and classification report
    accuracy = accuracy_score(y_test, y_pred)
    report = classification_report(y_test, y_pred, target_names=sorted_labels,
                                            output_dict=True)

    return y_pred, accuracy, report
```

Tiếp theo, chúng ta sẽ chạy hàm `train_and_test_xgboost` cho từng phương pháp mã hóa và in ra kết quả như sau:



(a) Confusion Matrix XGBoost với BoW

Huấn luyện mô hình XGBoost cho các phương pháp mã hóa

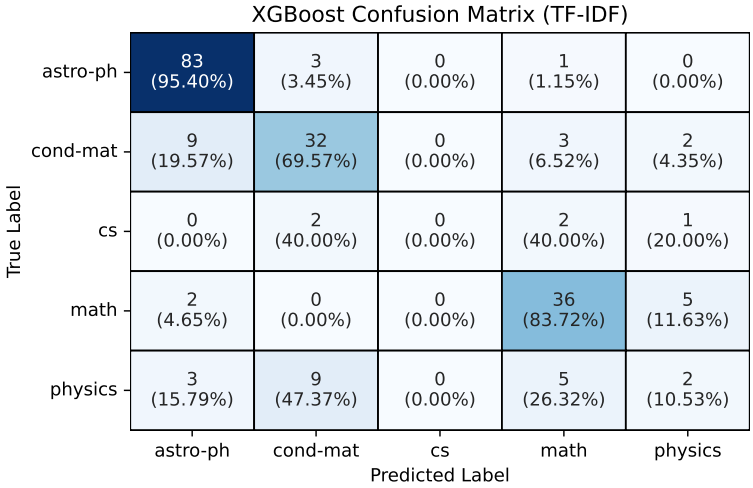
```
xgb_bow_labels, xgb_bow_accuracy, xgb_bow_report = train_and_test_xgboost(X_train_bow,
                                                   y_train, X_test_bow, y_test)
xgb_tfidf_labels, xgb_tfidf_accuracy, xgb_tfidf_report = train_and_test_xgboost(
                                                   X_train_tfidf, y_train, X_test_tfidf,
                                                   y_test)
xgb_embeddings_labels, xgb_embeddings_accuracy, xgb_embeddings_report =
                                                   train_and_test_xgboost(X_train_embeddings,
                                                   y_train, X_test_embeddings, y_test)

# Print XGBoost results
print("Accuracies for XGBoost:")
print(f"Bag of Words: {xgb_bow_accuracy:.4f}")
print(f"TF-IDF: {xgb_tfidf_accuracy:.4f}")
print(f"Embeddings: {xgb_embeddings_accuracy:.4f}")
```

Và đây là kết quả độ chính xác của mô hình XGBoost cho từng phương pháp mã hóa:

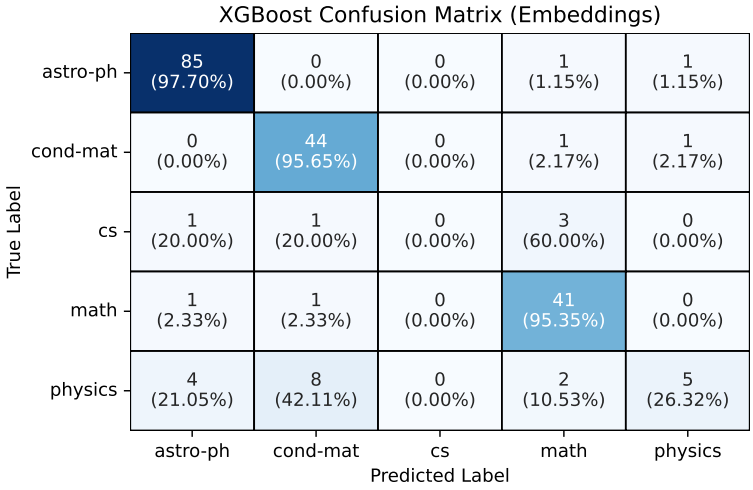
Độ chính xác của mô hình XGBoost cho từng phương pháp mã hóa

Accuracies for XGBoost:
Bag of Words: 0.7850
TF-IDF: 0.7650
Embeddings: 0.8750

Kết quả đánh giá mô hình XGBoost cho thấy độ chính xác đạt khoảng 78.5% với phương pháp BoW, 76.5% với TF-IDF và cao nhất là 87.5% với embeddings. Kết quả này cho thấy XGBoost hoạt động rất tốt với phương pháp mã hóa embeddings, đạt độ chính xác trong top những mô hình hoạt động tốt nhất trong các mô hình đã thử nghiệm. Tuy nhiên, kết quả với BoW và TF-IDF thấp hơn một chút so với các mô hình khác như SVM và Gradient Boosting.
Cuối cùng, chúng ta có confusion matrices cho từng phương pháp mã hóa trong Hình 7.



(b) Confusion Matrix XGBoost với TF-IDF



(c) Confusion Matrix XGBoost với Embeddings

Hình 7: Confusion Matrix cho từng phương pháp mã hóa với mô hình XGBoost

II.5.5. LightGBM

LightGBM (Light Gradient Boosting Machine) được xây dựng dựa trên thuật toán Gradient Boosting Decision Tree (GBDT). Đây là một framework học tăng cường theo gradient, tận dụng các thuật toán học dựa trên cây, đặc biệt là decision tree đã được tối ưu hóa.
Đặc điểm và tối ưu hóa nổi bật của LightGBM được liệt kê như sau:

- Phát triển cây theo lá (Leaf-wise, Best-first): Khác với nhiều phương pháp triển khai GBDT khác thường mở rộng cây theo từng tầng (level-wise), LightGBM mở rộng cây theo lá. Điều này có nghĩa là nó sẽ chọn lá có độ giảm tổn thất (delta loss) lớn nhất để chia, giúp mô hình hội tụ nhanh hơn và tiềm năng đạt độ chính xác cao hơn.

- Thuật toán dựa trên histogram: LightGBM sử dụng thuật toán dựa trên histogram để tìm điểm chia tối ưu, từ đó rút ngắn đáng kể thời gian huấn luyện, đặc biệt với các tập dữ liệu lớn.
- Tiết kiệm bộ nhớ: Thiết kế hướng đến hiệu quả sử dụng bộ nhớ, phù hợp để xử lý dữ liệu ở quy mô lớn.
- Hỗ trợ học song song và phân tán: LightGBM hỗ trợ nhiều chiến lược huấn luyện song song và phân tán, giúp mở rộng khả năng xử lý cho các bài toán Big Data.

Đầu tiên, chúng ta xây dựng hàm huấn luyện và kiểm thử mô hình LightGBM như sau:

Hàm `train_and_test_lightgbm` để huấn luyện mô hình LightGBM

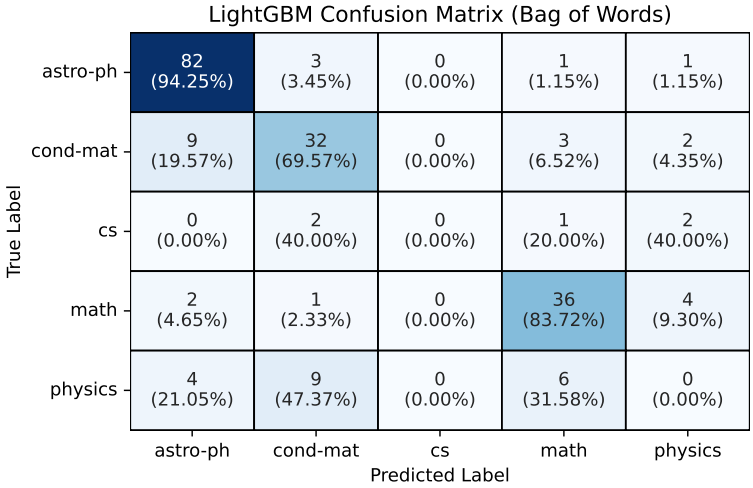
```
def train_and_test_lightgbm(X_train, y_train, X_test, y_test, n_estimators: int = 100):
    lgbm = lgb.LGBMClassifier(boosting_type='goss', n_estimators=n_estimators,
                                            random_state=42)
    lgbm.fit(X_train, y_train)

    # Predict on the test set
    y_pred = lgbm.predict(X_test)

    # Calculate accuracy and classification report
    accuracy = accuracy_score(y_test, y_pred)
    report = classification_report(y_test, y_pred, target_names=sorted_labels,
                                            output_dict=True)

    return y_pred, accuracy, report
```

Tiếp theo, chúng ta sẽ chạy hàm `train_and_test_lightgbm` cho từng phương pháp mã hóa và in ra kết quả như sau:

Huấn luyện mô hình LightGBM cho các phương pháp mã hóa

```
lgbm_bow_labels, lgbm_bow_accuracy, lgbm_bow_report = train_and_test_lightgbm(
                                            X_train_bow, y_train, X_test_bow, y_test)
lgbm_tfidf_labels, lgbm_tfidf_accuracy, lgbm_tfidf_report = train_and_test_lightgbm(
                                            X_train_tfidf, y_train, X_test_tfidf,
                                            y_test)
lgbm_embeddings_labels, lgbm_embeddings_accuracy, lgbm_embeddings_report =
                                            train_and_test_lightgbm(X_train_embeddings,
                                             y_train, X_test_embeddings, y_test)

# Print LightGBM results
print("Accuracies for LightGBM:")
print(f"Bag of Words: {lgbm_bow_accuracy:.4f}")
print(f"TF-IDF: {lgbm_tfidf_accuracy:.4f}")
print(f"Embeddings: {lgbm_embeddings_accuracy:.4f}")
```

Và đây là kết quả độ chính xác của mô hình LightGBM cho từng phương pháp mã hóa:

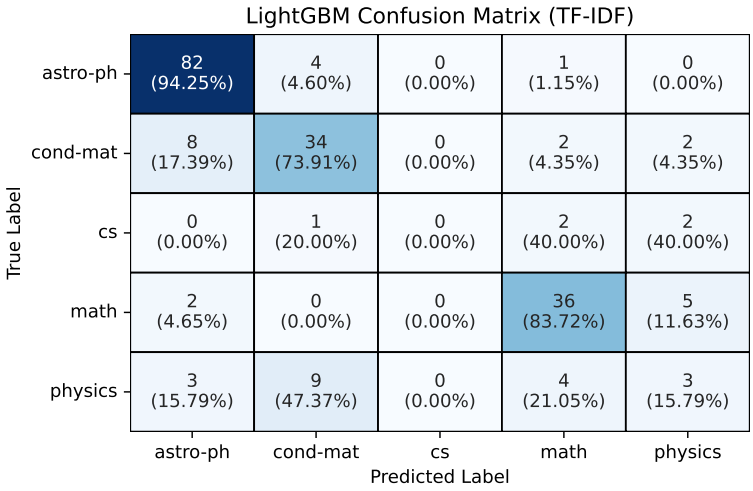


(a) Confusion Matrix LightGBM với BoW

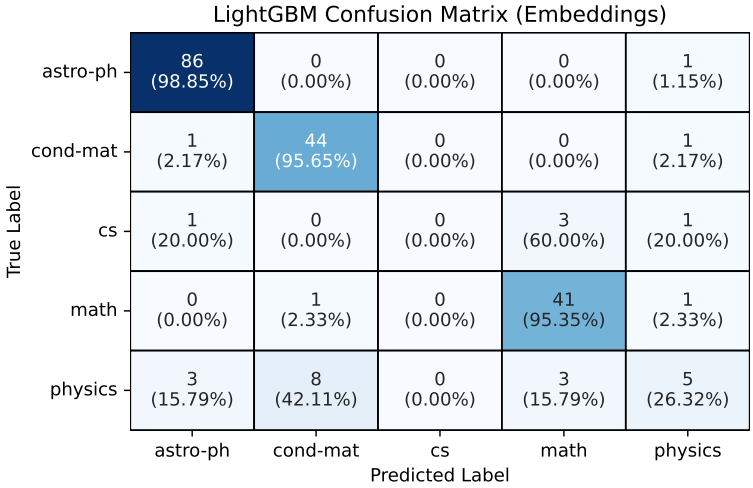
Độ chính xác của mô hình LightGBM cho từng phương pháp mã hóa

Accuracies for LightGBM:
Bag of Words: 0.7500
TF-IDF: 0.7750
Embeddings: 0.8800

Kết quả in ra màn hình cho thấy độ chính xác của mô hình LightGBM đạt khoảng 75% với phương pháp BoW, 77.5% với TF-IDF và cao nhất là 88% với embeddings. Tương tự với hầu hết các mô hình khác, mô hình LightGBM hoạt động tốt nhất với phương pháp mã hóa embeddings, đạt độ chính xác trong top những mô hình hoạt động tốt nhất trong các mô hình đã thử nghiệm. Kết quả với BoW và TF-IDF thấp hơn một chút so với các mô hình khác như SVM và Gradient Boosting.
Cuối cùng, chúng ta có confusion matrices cho từng phương pháp mã hóa trong Hình 8.



(b) Confusion Matrix LightGBM với TF-IDF



(c) Confusion Matrix LightGBM với Embeddings

Hình 8: Confusion Matrix cho từng phương pháp mã hóa với mô hình LightGBM

III. Câu hỏi trắc nghiệm

1. Mục tiêu chính của bài toán Text Classification là gì?
   (*a*) Dịch văn bản sang ngôn ngữ khác
   (*b*) Phân loại đoạn văn bản vào các nhóm/nhãn dựa trên nội dung
   (*c*) Tóm tắt văn bản thành một câu
   (*d*) Tạo văn bản mới từ dữ liệu huấn luyện

2. Bước tiền xử lý nào **không** được nêu trong tài liệu?
   (*a*) Chuyển chữ thường (lowercase)
   (*b*) Loại bỏ ký tự đặc biệt và chữ số
   (*c*) Loại bỏ ký tự xuống dòng và khoảng trắng dư
   (*d*) Stemming/Lemmatization

3. Phát biểu nào đúng về Bag-of-Words (BoW)?
   (*a*) Mã hoá dựa trên mô hình ngôn ngữ tiền huấn luyện
   (*b*) Bảo toàn thứ tự từ trong câu
   (*c*) Biểu diễn bằng đếm tần suất từ, bỏ qua trật tự
   (*d*) Luôn cho vector có kích thước 768

4. Vai trò của thành phần IDF trong TF-IDF là gì?
   (*a*) Tăng trọng số cho các từ xuất hiện thường xuyên trong mọi tài liệu
   (*b*) Giảm trọng số của các từ phổ biến trong toàn bộ tập dữ liệu
   (*c*) Chuẩn hoá chiều dài tài liệu về cùng độ dài
   (*d*) Loại bỏ hoàn toàn các từ dừng (stopwords)

5. Điểm khác biệt chính giữa BoW/TF-IDF và Sentence Embeddings là gì?
   (*a*) Embeddings cho vector dày (dense), kích thước cố định, nắm bắt ngữ nghĩa tốt hơn
   (*b*) Embeddings tạo vector rời rạc (sparse) có kích thước bằng kích thước từ vựng
   (*c*) Embeddings yêu cầu thủ công chọn đặc trưng
   (*d*) Embeddings chỉ hoạt động với văn bản tiếng Anh

6. Khi dùng `train_test_split(..., stratify=y)` với argument `stratify`, ta đang đảm bảo điều gì?
   (*a*) Kích thước vector đầu ra bằng nhau
   (*b*) Tỉ lệ nhãn giữa tập train/test được bảo toàn
   (*c*) Huấn luyện nhanh hơn
   (*d*) Tự động chuẩn hoá dữ liệu

7. Phát biểu nào đúng về Random Forest trong phân loại?
   (*a*) Mô hình chỉ chứa một cây rất sâu duy nhất để giảm bias
   (*b*) Tập hợp nhiều cây quyết định huấn luyện trên mẫu bootstrap, bỏ phiếu đa số
   (*c*) Huấn luyện tuần tự để giảm sai số của cây trước
   (*d*) Chỉ phù hợp với dữ liệu ảnh

8. Đặc trưng cốt lõi của AdaBoost là gì?
   (*a*) Chọn ngẫu nhiên đặc trưng tại mỗi nút chia
   (*b*) Huấn luyện song song nhiều mô hình yếu độc lập
   (*c*) Tăng trọng số các mẫu bị phân loại sai ở vòng trước để học tuần tự
   (*d*) Dùng tìm kiếm theo histogram cho điểm chia

9. Ý tưởng chính của Gradient Boosting là gì?
   (*a*) Lấy trung bình dự đoán của nhiều cây độc lập để giảm phương sai
   (*b*) Xây nhiều mô hình tuần tự, mỗi mô hình mới học theo hướng gradient của hàm loss
   (*c*) Nén từ vựng để giảm chiều
   (*d*) Tối ưu trực tiếp độ chính xác thay vì dùng hàm loss

10. LightGBM khác các triển khai GBDT thông thường ở điểm nào?
    (*a*) Phát triển cây theo lá (leaf-wise, best-first) và dùng thuật toán histogram để tìm điểm chia
    (*b*) Luôn bắt buộc độ sâu cây cố định bằng 1
    (*c*) Không hỗ trợ huấn luyện song song
    (*d*) Chỉ hoạt động với dữ liệu ảnh độ phân giải cao

Phụ lục

1. **Hint:** Các file code gợi ý và dữ liệu được lưu trong thư mục có thể được tải tại đây.

2. **Solution:** Các file code cài đặt hoàn chỉnh và phần trả lời nội dung trắc nghiệm có thể được tải tại đây (Lưu ý: Sáng thứ 3 khi hết deadline phần project, ad mới copy các nội dung bài giải nêu trên vào đường dẫn).

3. **Kiến thức cần cho project:**

   Để hoàn thành và hiểu rõ phần project một cách hiệu quả nhất, các học viên cần nắm rõ các kiến thức được trang bị

4. **Đề xuất phương pháp cải tiến project:**

   Project tiếp cận bài toán phân tích chủ đề văn bản (topic modeling) theo hướng sử dụng các phương pháp biểu diễn vector như Bag of Words (BoW), TF-IDF, S-BERT kết hợp với các thuật toán KNN, K-Means clustering và Decision Tree. Để tiếp tục cải tiến, tối ưu hệ thống; nhóm tác biên soạn gợi ý một số phương pháp như sau:

   - **Tiền xử lý nâng cao:** Sử dụng các kỹ thuật tiền xử lý chuyên sâu hơn như loại bỏ từ ngữ ít xuất hiện (rare words), lemmatization thay vì stemming, phát hiện và xử lý stopwords theo ngữ cảnh, và phát hiện cụm từ (phrase detection) để tăng chất lượng vector biểu diễn.
   - **Thử nghiệm nhiều phương pháp biểu diễn:**
     - So sánh hiệu quả giữa BoW, TF-IDF và mô hình embedding hiện đại như S-BERT.
     - Kết hợp nhiều phương pháp biểu diễn (feature fusion) để tận dụng ưu điểm của từng phương pháp.
   - **Tối ưu các thuật toán cây/ensemble:** bằng cách tinh chỉnh các siêu tham số mô hình
   - **Diễn giải mô hình (Model interpretability):**
     - Feature importance (split/gain/Permutation), SHAP cho cả cục bộ và toàn cục.
     - Partial Dependence / ICE để kiểm tra hiệu ứng đơn biến; *monotonic constraints* (XGB/LGBM) để áp điều kiện nghiệp vụ.
   - **Tăng cường dữ liệu (Data Augmentation):** Áp dụng các phương pháp như thay thế từ đồng nghĩa, paraphrasing hoặc back-translation để làm phong phú tập dữ liệu huấn luyện.

   Trên đây là một số gợi ý, giúp học viên có thể cải tiến thêm hệ thống để đạt hiểu quả tốt hơn về cả tốc độ xử lý và độ chính xác.

5. Rubric:

Mục	Kiến Thức	Đánh Giá
II.	- Lý thuyết về bài toán Topic Modeling trong Natural Language Processing. - Lý thuyết về các phương pháp biểu diễn văn bản: Bag of Words (BoW), TF-IDF, S-BERT và các thuật toán K-Means, KNN, Decision Tree.	- Hiểu được khái niệm topic modeling và các phương pháp biểu diễn văn bản. - Có khả năng sử dụng Python để triển khai các thuật toán K-Means, KNN và Decision Tree cho bài toán topic modeling.
III.	- Ứng dụng K-Means clustering để phân cụm chủ đề và Decision Tree để gán nhãn chủ đề. - So sánh hiệu quả các phương pháp biểu diễn văn bản (BoW, TF-IDF, S-BERT) trong bài toán topic modeling.	- Thực hiện phân cụm chủ đề và đánh giá mô hình bằng các chỉ số như Silhouette Score. - Đánh giá, so sánh và chọn phương pháp biểu diễn văn bản phù hợp cho dữ liệu thực tế.