Lab 05. Класифікація тексту

---

1. Мета роботи

Навчитися працювати з методами класифікації тексту у Python, використовувати векторизацію тексту (TF-IDF, CountVectorizer, HashingVectorizer), реалізовувати та оцінювати моделі класифікації (Naive Bayes, Logistic Regression, Random Forest), а також аналізувати якість моделей за допомогою метрик (accuracy, ROC-AUC тощо).

Очікуваний результат:
Після виконання лабораторної роботи студент уміє:

- перетворювати текст у числові вектори;
- будувати класифікатори для текстових даних;
- аналізувати якість моделей;
- покривати код юніт-тестами з використанням `pytest`.

2. Завдання

Для кожного з наведених фрагментів чужого коду необхідно:

1. **Проаналізувати** логіку роботи, знайти помилки та недоліки.
2. **Переписати код** у вигляді зрозумілих функцій або класів із док-рядками (docstrings) та type hints.
3. **Прокоментувати логіку** у вигляді коментарів або markdown-блоків.
4. **Покрити код юніт-тестами** (`pytest`), перевіривши:
    - форму та тип вихідних даних (матриці, моделі, метрики);
    - коректність результатів (наприклад, `accuracy ≥ 0.5`);
    - поведінку на граничних випадках (`with pytest.raises(...)`).

3. Варіанти коду

Варіант 1. CountVectorizer + Naive Bayes

```
from sklearn.datasets import fetch_20newsgroups
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.naive_bayes import MultinomialNB
train = fetch_20newsgroups(subset="train")
vec = CountVectorizer(stop_words="english")
X = vec.fit_transform(train.data)
clf = MultinomialNB()
clf.fit(X, train.target)
```

```
print("Train acc:", clf.score(X, train.target))
```

Варіант 2. TF-IDF + LogisticRegression (без нормалізації)

```
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.linear_model import LogisticRegression
tfidf = TfidfVectorizer(max_features=8000)
X = tfidf.fit_transform(train.data)
lr = LogisticRegression(max_iter=1000)
lr.fit(X, train.target)
```

Варіант 3. Препроцесінг зі стемінгом

```
import re, nltk, pandas as pd
stemmer = nltk.PorterStemmer()
def clean(s):
    s = re.sub(r"\W+", " ", s.lower())
    return " ".join(stemmer.stem(w) for w in s.split())
df = pd.DataFrame({"txt": train.data})
df["clean"] = df["txt"].apply(clean)
```

Варіант 4. Random Forest Classifier на розрідженій матриці

```
from sklearn.ensemble import RandomForestClassifier
rf = RandomForestClassifier(n_estimators=50, n_jobs=-1)
rf.fit(X.toarray(), train.target)
```

Варіант 5. Незбалансований train/test split (без stratify)

```
from sklearn.model_selection import train_test_split
Xtr, Xte, ytr, yte = train_test_split(X, train.target,
                                      test_size=0.3,
random_state=42)
```

Варіант 6. Pipeline + GridSearchCV (порожній param_grid)

```
from sklearn.pipeline import make_pipeline
from sklearn.model_selection import GridSearchCV
pipe = make_pipeline(TfidfVectorizer(),
LogisticRegression(max_iter=2000))
grid = GridSearchCV(pipe, param_grid={}, cv=3)
grid.fit(train.data, train.target)
```

Варіант 7. Помилкова оцінка ROC для багатокласу

```
from sklearn.metrics import roc_curve, auc
proba = clf.predict_proba(X)
fpr, tpr, _ = roc_curve(train.target, proba[:,1])   # багатоклас?
print(auc(fpr, tpr))
```

Варіант 8. HashingVectorizer (sign collision)

```
from sklearn.feature_extraction.text import HashingVectorizer
hv = HashingVectorizer(n_features=2**18)   # alternate_sign за
замовчуванням True
Xh = hv.transform(train.data)
```

Варіант 9. Custom tokenizer + LogReg, відсутні числа

```
token = lambda s: re.findall(r"[A-Za-z]+", s.lower())
cv = CountVectorizer(tokenizer=token)
Xc = cv.fit_transform(train.data)
lr = LogisticRegression(max_iter=1500).fit(Xc, train.target)
```

Варіант 10. Візуальний аналіз важливих слів (coef_)

```
import numpy as np
top = np.argsort(lr.coef_[0])[-10:]
print([tfidf.get_feature_names_out()[i] for i in top])
```

4. Хід роботи

1. **Ознайомлення з теорією:**
   Повторіть поняття TF-IDF, CountVectorizer, токенізацію, стемінг, лематизацію, базові моделі класифікації тексту.

2. **Аналіз вихідного коду:**
   Виберіть один із варіантів (1–10) і визначте:

   - яку задачу він виконує;
   - які в ньому помилки або слабкі місця;
   - як можна підвищити стабільність і якість коду.

3. **Рефакторинг:**
   Перепишіть код, структуровано розділивши функції:

   - `load_data()` — завантаження датасету;
   - `preprocess(texts)` — очищення тексту (регулярки, стемінг);
   - `vectorize(train, test, method="tfidf")` — побудова векторного подання;

- `train_model(X, y, model_name)` — навчання класифікатора;
- `evaluate(y_true, y_pred, y_proba=None)` — обчислення метрик.

4. **Написання тестів:**
   Переконайтеся, що кожна функція має тести:
   - `assert X.shape[0] == len(y)`
   - `assert 0.0 <= acc <= 1.0`
   - `with pytest.raises(ValueError): vectorize([""], method="bogus")`
5. **Оформлення результатів:**
   Підготуйте звіт із поясненням логіки коду, скріншотами результатів і висновками.

---

5. Чекліст для перевірки виконання завдання

Критерій	Виконано
Є функціональна структура коду (load_data, preprocess, vectorize, train_model, evaluate)	
Додано док-рядки та type hints	
Написано щонайменше 3 юніт-тести (pytest)	
Виправлено логічні або синтаксичні помилки у фрагменті	
Модель успішно навчається та оцінюється (accuracy ≥ 0.5)	
Продемонстровано реакцію на edge cases	
Код відповідає стилю PEP8 (`black`, `flake8`)	
Є короткі пояснення або markdown-коментарі	

---

6. Короткі теоретичні відомості

6.1. Основи класифікації тексту

Класифікація тексту — це задача автоматичного віднесення документа до однієї або кількох категорій (наприклад, "спорт", "політика", "наука").
Вона є підзадачею **обробки природної мови (NLP)** і широко застосовується у спам-фільтрах, аналізі настроїв, категоризації новин, рекомендаційних системах тощо.

Основні етапи:

1. **Збір та очищення даних.**
2. **Попередня обробка тексту (preprocessing).**
3. **Векторизація (перетворення тексту у числа).**
4. **Навчання класифікатора.**
5. **Оцінка якості.**

6.2. Попередня обробка тексту (Preprocessing)

Основна мета — підготувати тексти до векторизації. Типові кроки:

- **Нижній регістр** (`lower()`): уніфікація написання.
- **Видалення небажаних символів:** пунктуації, HTML-тегів, цифр (`re.sub`).
- **Токенізація:** розбиття тексту на слова (`nltk.word_tokenize`).
- **Видалення стоп-слів:** наприклад, “the”, “and”, “of” (`stopwords.words('english')`).
- **Стемінг / Лематизація:** зведення слів до базової форми (`PorterStemmer`, `WordNetLemmatizer`).

Приклад:

```
def preprocess(text: str) -> str:
    text = re.sub(r'\W+', ' ', text.lower())
    stemmer = nltk.PorterStemmer()
    return ' '.join(stemmer.stem(w) for w in text.split())
```

6.3. Методи векторизації

1. **CountVectorizer**
   Перетворює тексти у частотну матрицю: кожен елемент = кількість входжень слова.
   Формула:

   $$X_{ij} = \text{count}(w_j, d_i)$$

   Переваги: проста реалізація, швидкість.
   Недоліки: не враховує важливість слів.

2. **TF-IDF (Term Frequency–Inverse Document Frequency)**
   Враховує як частоту слова в документі, так і його рідкість у корпусі:

$$\text{tfidf}(t,d) = tf(t,d) \cdot \log \frac{N}{df(t)}$$

   Підкреслює унікальні терміни, знижує вагу поширених слів.

3. **HashingVectorizer**
   Використовує хешування для перетворення токенів у індекси фіксованої довжини. Швидкий і пам'яттєво-ефективний, але не має "зворотного словника".

4. **Word Embeddings (розширено, для довідки)**
   Методи на кшталт `Word2Vec`, `GloVe`, `FastText` створюють щільні вектори, які відображають семантику слів.

6.4. Класифікатори

Модель	Принцип роботи	Переваги	Недоліки
Multinomial Naive Bayes	Використовує теорему Байєса з припущенням незалежності ознак.	Простий, ефективний для текстів.	Ігнорує залежності між словами.
Logistic Regression	Лінійна модель, прогнозує імовірність належності до класу.	Добра інтерпретованість, стабільна.	Потребує нормалізації, чутлива до викидів.
Random Forest	Ансамбль дерев рішень, працює з нелінійними залежностями.	Стійкий, добре узагальнює.	Повільний на великих розріджених матрицях.

6.5. Оцінка якості моделей

Основні метрики:

- **Accuracy:** частка правильних передбачень.

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN}$$

- **Precision / Recall / F1-score:** баланс між помилками першого і другого роду.
- **ROC-AUC:** площа під ROC-кривою, корисна для бінарних моделей.

- **Cross-validation:** оцінка стабільності моделі на різних підмножинах даних (`GridSearchCV`).

6.6. Проблеми та типові помилки

1. **Незбалансований `train_test_split`** → використовуйте `stratify=y`.
2. **Багатокласова ROC-крива:** `roc_curve` працює лише для 2 класів.
3. **Використання `.toarray()` на великих розріджених матрицях:** призводить до переповнення пам'яті.
4. **Порожній `param_grid` у `GridSearchCV`:** модель не налаштовується.
5. **Хеш-колізії у `HashingVectorizer`:** перевіряйте параметр `alternate_sign=False`.

7. Корисні посилання

- 📘 Scikit-learn: https://scikit-learn.org/stable/modules/feature_extraction.html
- 📗 NLTK Documentation: https://www.nltk.org/
- 📙 Pytest: https://docs.pytest.org/
- 📒 Text classification tutorial (sklearn): https://scikit-learn.org/stable/tutorial/text_analytics/working_with_text_data.html
- 🧰 Code style tools: https://black.readthedocs.io/, https://flake8.pycqa.org/