HW #2 Problem Set

2071008 SeoyeonKim

1. Spam Classification via Naïve Bayes

더욱 자세한 구현 및 설명은 https://flowersayo.tistory.com/135 참고

a)

nb_train

주어진 **훈련 데이터를 기반으로 나이브 베이즈 분류기에 필요한 확률값을 계산**한다. 이 때, 라플라스 스무딩도 적용한다.

우리가 빠른 예측을 위하여 미리 계산해두어야 하는 값들은 다음과 같다.

- P(Spam), P(Not Spam)
- P(xi | Spam) i = 1 ~ 1448
- P(xi | Not Spam) i = 1 ~ 1448

```python
def nb_train(matrix, category):
    state = {}
    N = matrix.shape[1]  # 전체 특성(토큰)의 개수


    # 각 클래스에 속하는 문서 수
    spam_count = np.sum(category)  # 스팸 문서 수 (레이블이 1인 경우)
    not_spam_count = len(category) - spam_count  # 스팸이 아닌 문서 수 (레이블이 0인 경우)

    # 사전 확률 P(spam) 및 P(not_spam)
    P_spam = spam_count / len(category)
    P_not_spam = not_spam_count / len(category)

    # 스팸과 비스팸 문서에서 토큰이 등장한 횟수를 구함
    token_counts_spam = np.sum(matrix[category == 1], axis=0)   # 스팸 문서에서 각 토큰이 등장한 횟수
    token_counts_not_spam = np.sum(matrix[category == 0], axis=0)  # 비스팸 문서에서 각 토큰이 등장한 횟수

    # 라플라스 스무딩을 적용하여 조건부 확률 P(token|spam) 및 P(token|not_spam) 구하기
    P_token_given_spam = ( token_counts_spam + 1 ) / ( spam_count + N )
    P_token_given_not_spam = ( token_counts_not_spam + 1 ) / ( not_spam_count + N )

    # 학습된 상태를 저장
    state['P_spam'] = P_spam
    state['P_not_spam'] = P_not_spam
    state['P_token_given_spam'] = P_token_given_spam
    state['P_token_given_not_spam'] = P_token_given_not_spam

    return state
```

matrix[category == 1] 를 통해 본 matrix 에서 **스팸으로 라벨링 된 메일만 필터링**한다. axis=0, 즉 열 방향에 대하여 누적합을 계산함으로서 **전체 스팸 메일 중에서 i번째 토큰이 나타나는 총 빈도**를 계산해낼 수 있다. 스팸이 아닌 메일에 대해서도 동일하게 처리한다.

결과 (예시)

token_counts_spam : xi ∩ Spam , 스팸 메일 중에서 i번째 토큰이 등장한 총 횟수

[0]	[1]	[2]	[3]	..
0	5	6	7	..

token_counts_not_spam : xi ∩ Not Spam , 스팸이 아닌 메일 중에서 i번째 토큰이 등장한 총 횟수

[0]	[1]	[2]	[3]	..
0	13	14	5	..

P_token_given_spam : P(xi | Spam)

[0]	[1]	[2]	[3]	...
0	0.33	0.55	0.33	

P_token_given_spam : P(xi | Not Spam)

[0]	[1]	[2]	[3]	..
0	0.13	0.23	0.05	..

nb_test

훈련된 모델의 확률을 사용하여 각 문서가 스팸인지 아닌지를 예측한다.

```
def nb_test(matrix, state):
    output = np.zeros(matrix.shape[0])  # 예측 결과 저장

    # 사전 확률과 조건부 확률 불러오기
    P_spam = state['P_spam']
    P_not_spam = state['P_not_spam']
    P_token_given_spam = state['P_token_given_spam']
    P_token_given_not_spam = state['P_token_given_not_spam']

    # 각 문서에 대해 스팸 여부를 예측
    for i in range(matrix.shape[0]):
        # 각 문서의 토큰 빈도
        doc = matrix[i]

        # 로그 확률을 계산하여 언더플로우 방지
        log_prob_spam = np.log(P_spam) + np.sum(np.log(P_token_given_spam) * doc)
        log_prob_not_spam = np.log(P_not_spam) + np.sum(np.log(P_token_given_not_spam) *
doc)

        # 더 높은 로그 확률을 가진 클래스로 예측
        if log_prob_spam > log_prob_not_spam:
            output[i] = 1  # 스팸으로 분류
        else:
            output[i] = 0  # 스팸이 아닌 것으로 분류

    return output
```

여기서 확률을 구할 때 조금 혼란스러울 수 있는 점은 기존 예제와는 조금 다르게 각 문서의 특성에 해당하는 값들이 단순히 그 특성을 가지고 있다, 가지고 있지 않다로 판별되는 것이 아니라 특성이 나타나는 횟수를 기록하고 있다는 점이다.

이러한 상황에서는 각 토큰의 조건부 확률 P(token_i | spam)을 토큰 등장 빈도수만큼 거듭 제곱해줄 것이다. 예를 들어 어떤 문서에서 단어 출현 빈도가 a=3, b=4 일 때의 스팸 확률을 구한다면

P(Spam) * P(a | Spam) ^ 3 * P(b | Spam)^4 를 구해주는 셈이다.

따라서 우리는 P_token_given_spam, doc 를 가지고 다음과 같이 계산을 수행할 수 있다.

P_token_given_spam : P(xi | Spam)

[0]	[1]	[2]	[3]	...
0	0.33	0.55	0.33	

doc: Matrix[i], i번째 샘플의 단어 행렬

[0]	[1]	[2]	[3]	..
0	1	3		..

결과 : P_token_given_spam ^ doc

[0]	[1]	[2]	[3]	...
0	0.33^1	0.55^3	0.03^3	

즉, 최종적으로는 이 행렬의 값들을 모두 곱해서 **P(token_i | spam)** 를 구해내게 된다.

$$0.33^1 \times 0.55^3 \times 0.03^3 ...$$

언더플로우를 고려하여 로그를 적용하면 로그의 성질에 의하여 다음과 같이 단어 빈도 행렬의 값인 doc 이 상수로 앞에 빠져나오게 되고

$$1 \times log0.33 + 3 \times log0.55 + 3 \times log0.03 ...$$

로그를 취해 확률을 일반화하면 다음과 같은 공식이 나온다.

$$\log(P(spam)) + \sum(\log(P(token_i|spam)) \times \text{doc}[i])$$

이를 실제 코드로 구현한 결과는 다음과 같다.

```
log_prob_spam = np.log(P_spam) + np.sum(np.log(P_token_given_spam) * doc)
```

결과

```
kimseoyeon@gimseoyeons-MacBook-Air 38428-02-HW2 % python3 hw2_nb.py
Error: 0.0600
```

Error: 0.0600

b)

스팸과 비스팸을 구별하는 데 가장 중요한 토큰 5개를 찾을 것이다. 따라서 각 토큰이 스팸 클래스에서 얼마나 중요한지(특징이 되는지)를 나타내는 값을 계산하고, 가장 큰 값을 가진 5개의 토큰을 선택할 것이다.

이를 위하여 주어진 수식을 기반으로 로그 우도비(Log-likelihood ratio)를 계산한다.

$$\log\left(\frac{P(\text{token i}|\text{SPAM})}{P(\text{token i}|\text{NOTSPAM})}\right)$$

위의 수식을 코드에 반영하여, 토큰의 확률을 스팸과 비스팸에서 비교하는 로그 확률을 구한 다음, 그 값을 정렬하여 스팸을 변별하는 상위 5개의 토큰을 찾을 수 있다.

```python
# Spam 과 Not Spam 을 구분하는 상위 n개의 토큰 찾기
def findTopToken(top_n,state,tokens):

    P_token_given_spam = state['P_token_given_spam']
    P_token_given_not_spam = state['P_token_given_not_spam']

    # 각 토큰에 대한 로그 우도비(log-likelihood ratio)를 계산
    log_likelihood_ratio = np.log(P_token_given_spam) - np.log(P_token_given_not_spam)

    # 로그 우도비가 가장 큰 상위 top_n개의 토큰을 찾음
    top_token_indices = np.argsort(log_likelihood_ratio)[-top_n:][::-1]

     # 상위 top_n개의 토큰과 그 값을 출력
    print(f"\n== Top {top_n} tokens most indicative of SPAM ==\n")
    for idx in top_token_indices:
        print(f"Token: {tokens[idx]}, Log-Odds Ratio:
{log_likelihood_ratio[idx]:.4f}")==========\n")
```

결과

html, space, unsolicit, eat, vacat 가 스팸을 변별하는 Top 5 키워드로 선별되었다.

```
== Top 5 tokens most indicative of SPAM ==

Token: html, Log-Odds Ratio: 7.3856
Token: space, Log-Odds Ratio: 7.3307
Token: unsolicit, Log-Odds Ratio: 5.7823
Token: eat, Log-Odds Ratio: 5.5584
Token: vacat, Log-Odds Ratio: 5.5507

==========
```

c)

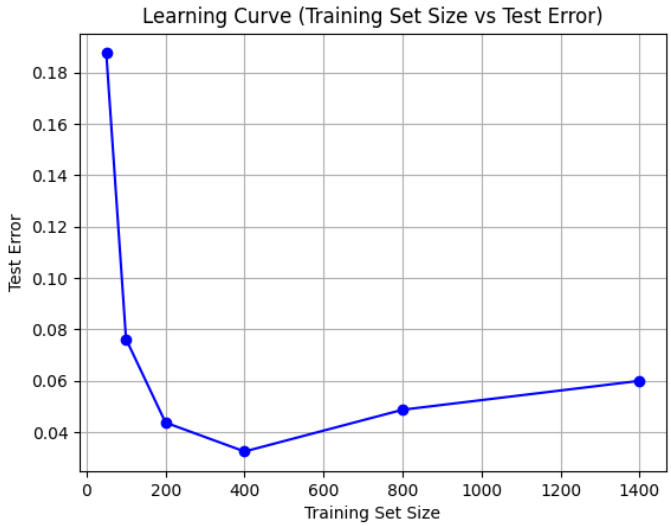
다양한 크기의 훈련 세트를 사용하여 모델을 학습하고, 그에 따라 테스트 세트에서의 오류율을 계산하여 학습 곡선(learning curve)을 그릴 것이다. 훈련 세트의 크기는 50, 100, 200, ... 최대 1400까지 주어지며, 각각의 훈련 세트에 대해 학습을 진행한 후 테스트 세트에서의 오류율을 계산한다.

```python
def plotLearningCurve(train_sizes, test_errors):

    plt.plot(train_sizes, test_errors, marker='o', color='b')
    plt.title('Learning Curve (Training Set Size vs Test
Error)')
    plt.xlabel('Training Set Size')
    plt.ylabel('Test Error')
    plt.grid(True)
    plt.show()
```

```python
    # 훈련 세트 크기별로 파일을 읽고 학습 후 오류율 계산
    training_set_size = [50, 100, 200, 400, 800, 1400]
    test_errors = []
    for size in training_set_size:
        trainMatrix, tokenlist, trainCategory = readMatrix(f'./data/hw2_MATRIX.TRAIN.
{size}')
        state = nb_train(trainMatrix, trainCategory)
        output = nb_test(testMatrix, state)
        error = evaluate(output, testCategory)
        test_errors.append(error)

    # 학습 곡선 그리기
    plotLearningCurve(training_set_size, test_errors)
```

결과

Training Set Size = 400 일 경우 가장 낮은 오차율을 보이는 것을 확인할 수 있다.



2. Spam Classification via SVM

더욱 자세한 구현 및 설명은 https://flowersayo.tistory.com/136 참고

a)

세 가지 SVM 분류기를 설정한다.

- **하드 마진 SVM (svm_clf_hard)**: 선형 커널을 사용하며, C 값이 ∞로 설정되어 있어 매우 엄격한 마진을 사용
- **소프트 마진 SVM (svm_clf_soft)**: C 값을 1로 설정하여 소프트 마진을 사용
- **가우시안 RBF 커널을 사용하는 SVM (svm_clf_rbf)**: 비선형 데이터를 처리할 수 있도록 RBF 커널을 사용하며, gamma와 C 값을 조정하여 모델을 설정

```
def main():
    # Please set a training file that you want to use for this run below
    trainMatrix, tokenlist, trainCategory = svm_readMatrix('./data/hw2_MATRIX.TRAIN.400')
    testMatrix, tokenlist, testCategory = svm_readMatrix('./data/hw2_MATRIX.TEST')

    # SVM Classifier model

    # Hard margin SVM
    svm_clf_hard = SVC(kernel="linear", C=float("inf"), max_iter=10_000, random_state=42)


    # Soft margin SVM
    svm_clf_soft = SVC(kernel="linear",gamma=1, C=0.1, max_iter=10_000, random_state=42)
    grid_search(svm_clf_soft,trainMatrix, trainCategory)  # {'svc__C': 0.1, 'svc__gamma': 1}


    # Gaussian RBF SVM
    svm_clf_rbf = SVC(kernel="rbf", gamma=0.001, C=10, max_iter=10_000, random_state=42)
    grid_search(svm_clf_rbf,trainMatrix, trainCategory) # {'svc__C': 10, 'svc__gamma':
0.001}

    # Naive Bayes Classifier
    nb_clf = MultinomialNB()

    scaler = StandardScaler()

    # Scaled version for each SVM and we will use these
    scaled_svm_clf_hard = make_pipeline(scaler, svm_clf_hard)
    scaled_svm_clf_soft = make_pipeline(scaler, svm_clf_soft)
    scaled_svm_clf_rbf = make_pipeline(scaler, svm_clf_rbf)
```

이 때, RBF, Soft SVM 에서 최적의 파라미터를 찾기 위하여 그리드 서치를 활용한다.

그리드 서치(Grid Search)는 다양한 **하이퍼파라미터 값의 조합**을 시도하여 그 중에서 **최적의 값**을 찾는 방법이다. **각 하이퍼파라미터 값의 모든 조합을 시도**하면서, 각 조합에 대한 성능을 평가한 후, 최적의 성능을 내는 파라미터를 선택한다.

```python
# SVM 모델 최적의 파라미터 찾기 - 그리드 서치 (grid search)
def grid_search(svm_model,trainMatrix, trainCategory):
    param_grid = {
        'C': [0.1, 1, 10, 100],          # C의 값 후보들
        'gamma': [1, 0.1, 0.01, 0.001],  # gamma의 값 후보들
    }

    # 그리드 서치 설정 (교차 검증을 5번 수행)
    grid_search = GridSearchCV(svm_model, param_grid, cv=5,
scoring='accuracy')
    # 데이터 로드 (trainMatrix, trainCategory) - 이미 존재하는 데이터 사용

    # 그리드 서치 실행 (훈련 데이터에 대해 최적 파라미터를 찾음)
    grid_search.fit(trainMatrix, trainCategory)

    # 최적의 파라미터와 그에 따른 성능 확인
    print(f"Best parameters: {grid_search.best_params_}")
    print(f"Best cross-validation accuracy: {grid_search.best_score_:.4f}")

    return
```

그리드 서치 결과 콘솔

```
Best parameters: {'C': 0.1, 'gamma': 1}
Best cross-validation accuracy: 0.9900
Best parameters: {'C': 10, 'gamma': 0.001}
Best cross-validation accuracy: 0.9825
```

그리드 서치 결과에 따르면 Soft margin SVM에서는 C=0.1 gamma=1, Gaussian RBF SVM 에서는 C=10 gamma=0.001 이 최적의 파라미터로 검색되므로 이 하이퍼파라미터로 각 모델을 생성한다.

결과

```
== compare SVM implementations  ==

Hard margin SVM Error: 0.0350
Soft margin SVM Error: 0.0350
Gaussian RBF SVM Error: 0.2300

====================================
```

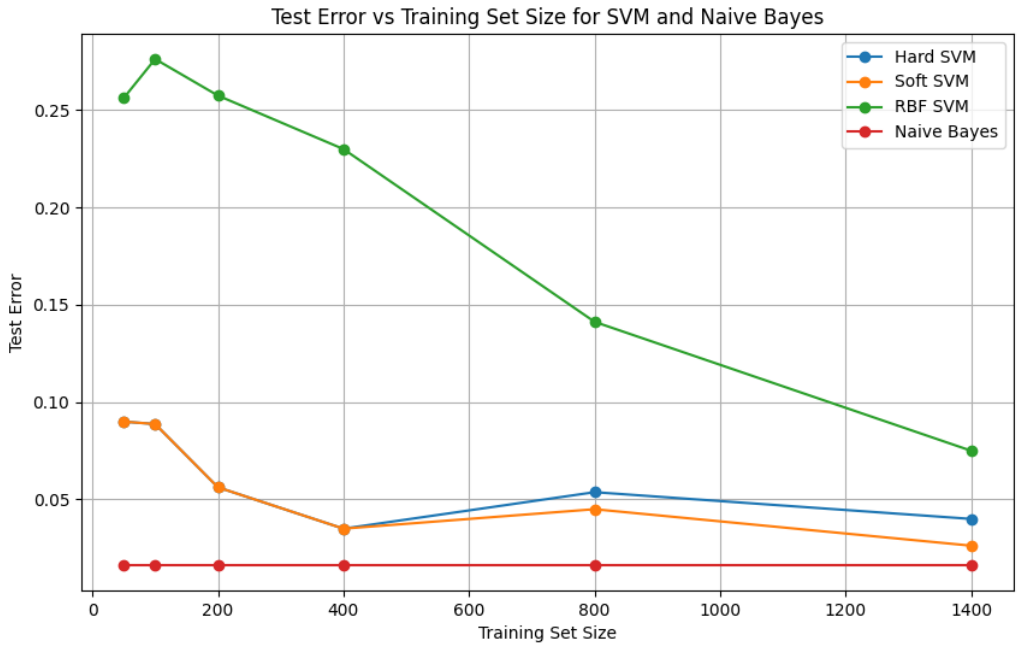
Hard margin SVM Error : 0.0350

Soft margin SVM Error : 0.0350

Gaussian RbF SVM Error : 0.23

b)

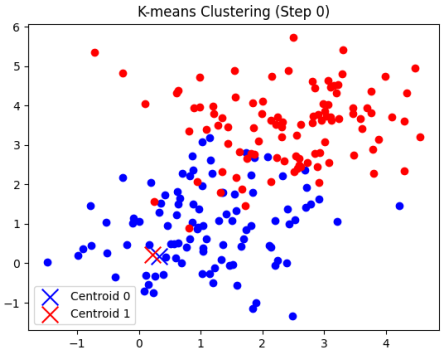
학습 데이터 수에 따른 4가지 모델(Hard, Soft, RBF, Naive Bayes)의 성능을 비교한다.

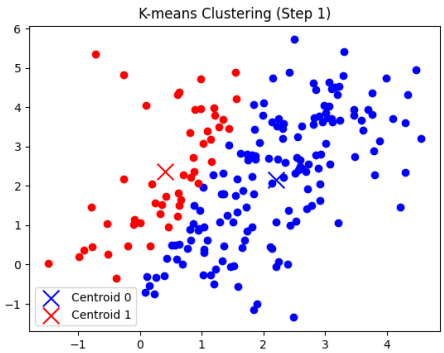


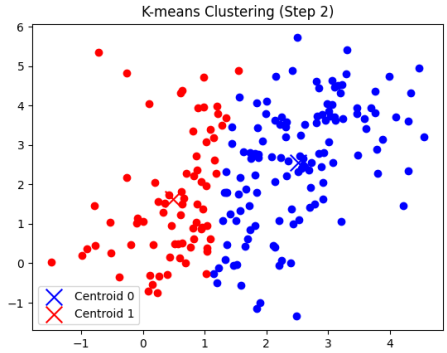
- **Naive Bayes**는 데이터 크기에 관계없이 매우 안정적인 성능을 보인다.
- **RBF SVM**은 많은 데이터가 필요하지만, 데이터가 충분히 커지면 성능이 크게 향상됩니다.
- **Soft SVM**은 적절한 데이터 크기에서 안정적인 성능을 보인다.
- **Hard SVM**은 작은 데이터셋에서는 성능이 좋지만, 데이터가 증가하면 성능이 저하될 수 있다.

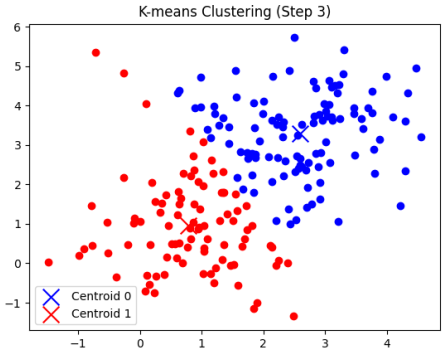
3. K-means Clustering

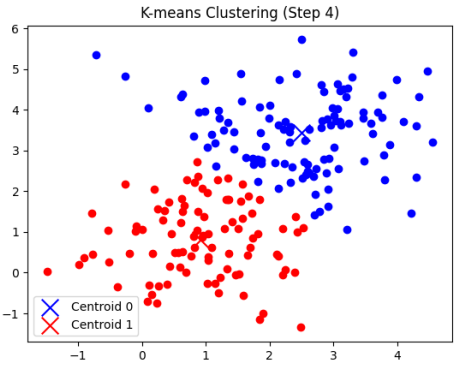
더욱 자세한 구현 및 설명은 https://flowersayo.tistory.com/137 참고

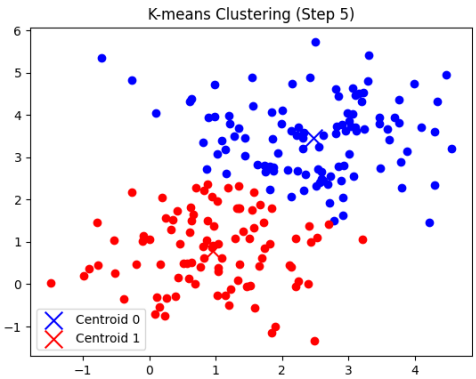


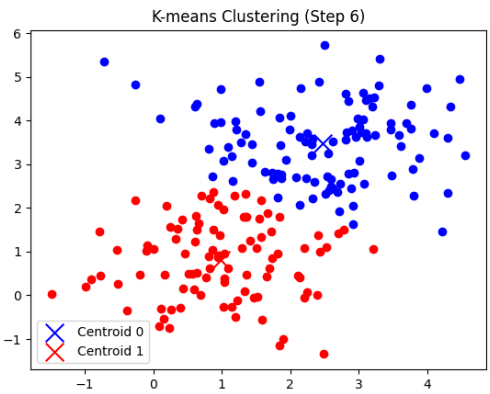


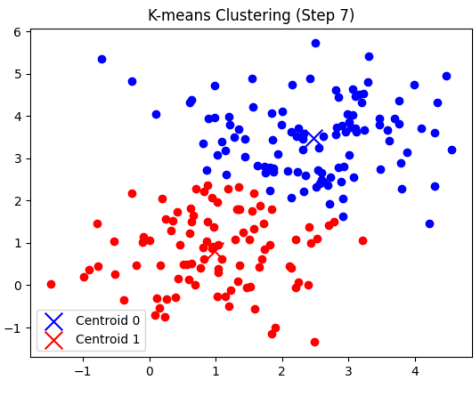












총 7번의 반복 후에 클러스터의 중심점 좌표가 tol(1e-5) 미만으로 움직이게 되므로 클러스터링이 완료되었다고 판단한다.