**Artiﬁcail Intelligence assignment2 Report *2016025514* 서현아**

**"Naive Bayse Classiﬁcation을 이용한 영화 리뷰 긍정/부정 분류하기"**

# 문제 해결 방식

* 1. Training

주어진 Data Set은 각 리뷰에 대한 긍정/부정이 labeling 된 상태.

해당 Data Set 파일을 읽어 와서, 우선 긍정 리뷰와 부정 리뷰로 분리 후 리뷰에 등장하는 단어들이 각각 긍정 리뷰와 부정 리뷰에서 어떤 비율로 등장하는 지를 기록하는 possibility table을 제작.

* 1. Classify

위의 방식으로 진행된 Training 결과를 바탕으로 새로운 review가 들어 왔을 때, 해당 리뷰가 긍정인지 부정인 지를 파악.

P(긍정 | new review) 는 P(word1 | 긍정) x P(word2 | 긍정) x ... x P(word n | 긍정) P(부정 | new review) 는 P(word1 | 부정) x P(word2 | 부정) x ... x P(word n | 부정)

과 같은 방식으로 새로운 리뷰가 긍정일 확률과 부정일 확률을 비교하고, 확률이 더 높은 쪽을 결과로 여김.

**이 때, 새로운 리뷰를 구성하는 단어들이 training set에 있지 않을 경우에 대한 처리가 필요하다.**

* 1. Rating

최적화 방식이나 위에서 언급한 것과 같이 training set에 존재하지 않는 단어에 대한 처리 방식에 따라서, classiﬁcation의 정확도가 달라짐.

따라서 정확도를 판단하는 과정이 추가적으로 필요.

# Source Code (추가적인 설명은 함수 설명 아래의 코드 주석 참고)

* 1. getReviews(파일명) : 인자로 받은 파일 - Data Set 을 열어서 리뷰 list를 반환하는 함수

1

2

3

4

5

6

7

8

9

10

11

#리뷰 파일을 열어서 list에 리뷰를 담는 함수

def getReviews(filename): reviews = [] #encoding issue 처리

tmp = open(filename, 'r', encoding="UTF-8")

for line in tmp:

#첫 줄 제외하고 reviews 리스트에 리뷰 담기

if not line.startswith('id'): reviews.append(line)

return reviews

* 1. sortReviews(reviews) : getReviews를 통해 얻은 리뷰 list를 인자로 받고, 긍정 리뷰 list와 부정 리뷰 list로 분리하여 반환하는 함수

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

#리뷰 list를 분류기에 넣기 전에 긍정/부정 sorting 작업을 위한 함수

def sortReviews(reviews): #리뷰를 한 줄씩 저장해둘 리스트

tempgoodReviews = []

tempbadReviews = []

#단어로 잘라서 넣을 리스트

goodReviews = [] badReviews = []

#긍정 리뷰와 부정 리뷰로 분할하는 for loop for review in reviews:

line = review.split()

if line[-1] == '1': tempgoodReviews.append(line[1: len(line)-1])

elif line[-1] == '0': tempbadReviews.append(line)

#긍정 리뷰들의 단어 모음을 만드는 for loop for line in tempgoodReviews:

for word in line: goodReviews.append(word)

#부정 리뷰들의 단어 모음을 만드는 for loop for line in tempbadReviews:

for word in line: badReviews.append(word)

return goodReviews, badReviews

* 1. makeCount(reviews) : sortReviews의 결과로 나온 goodReviews, badReviews 리스트에 대해 각각 한 번씩 해당 함수를 적용. 긍정/부정 각 단어 리스트에서 각각의 단어가 몇 번씩 등장하는지 개수를 구하는 함수로 리스트 보다는 딕셔너리 자료 구조를 사용하는 것이 효율적이라 판단함. 따라서 key : 단어, value : 단어의 등장 횟수로 구성된 딕셔너리가 긍정/부정에 대해 각각 한 차례씩 반환.

1

2

3

4

5

6

7

#긍정/부정 각각 어떤 단어들이 몇 번 등장하는지 개수 체크하는 함수 >> 딕셔너리 자료구조 사

용

def makeCount(reviews):

#딕셔너리 counts 초기화

counts = defaultdict(lambda: [0, 0])

#긍정/부정 리뷰를 돌면서 각 단어가 몇 차례 등장하는지 파악

8

9

10

11

12

13

14

15

16

for i in range(len(reviews)):

#딕셔너리에 이미 존재하는 단어 등장 시, value 1 증가

if reviews[i] in counts: counts[reviews[i]] += 1

#딕셔너리에 존재하지 않는 단어 등장 시, 딕셔너리에 추가, value는 1로 초기화

else:

counts[reviews[i]] = 1

return counts

* 1. makePossibility(good\_count, bad\_count, goodReviews, badReviews) : makeCount의 결과인 긍 정/부정 단어 개수 딕셔너리와 긍정/부정 리뷰 단어 리스트를 인자로 받는 함수.

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

#리뷰의 단어 개수 체크한 결과를 확률로 바꿔주는 함수

def makePossibility(good\_count, bad\_count, goodReviews, badReviews): #긍정 단어 수

num\_good\_words = len(goodReviews) #부정 단어 수

num\_bad\_words = len(badReviews)

#전체 단어

total\_word = list(set(goodReviews + badReviews)) #전체 단어 수

num\_total\_words = len(total\_word)

#P(긍정)

total\_goodP = num\_good\_words/num\_total\_words #P(부정)

total\_badP = num\_bad\_words/num\_total\_words

#딕셔너리 possibility 초기화

possibility = defaultdict(lambda: [0, 0])

#전체 단어 for loop으로 돌면서 긍정/부정 리뷰들에 각각 몇 번씩 등장하는지 카운트

for word in total\_word:

#단어가 긍정 단어에 있다면

if word in good\_count: #good\_count는 딕셔너리 타입

#따라서 key : word를 이용하여 value를 찾음. num\_good = good\_count[word]

#단어가 긍정 단어에 없다면

elif word not in good\_count: num\_good = 0

#단어가 부정 단어에 있다면

if word in bad\_count: #bad\_count는 딕셔너리 타입

#따라서 key : word를 이용하여 value를 찾음.

37

38

39

40

41

42

43

44

45

46

47

48

49

50

51

52

53

54

55

56

리 타입

57

58

59

60

num\_bad = bad\_count[word] #단어가 부정 단어에 없다면

elif word not in bad\_count: num\_bad = 0

#Laplacing Smoothing

#긍정 리뷰나 부정 리뷰 한 쪽에서 한 번도 등장하지 않은 경우 확률이 0이 되는데 #추후 classification 과정에서 0인 확률이 곱해지는 상황을 피하기 위해서 #(등장횟수 + 1) / (긍정 혹은 부정 총 단어 수 + 전체 단어 수)로 확률을 계산함.

#Laplacing Smoothing을 하는 방식이 여러가지인 것으로 알고 있음. #해당 방식에 따라서 정확도에 차이가 발생할 것으로 생각.

#P(word|긍정)에 laplacing smoothing 처리

goodP = (num\_good + 1) / (num\_good\_words + num\_total\_words) #P(word|부정)에 laplacing smoothing 처리

badP = (num\_bad + 1) / (num\_bad\_words + num\_total\_words)

#Possibility table은

#key : word, value : [긍정리뷰 등장 확률, 부정리뷰 등장 확률] 형태의 딕셔너

#기존에는 possibility를 list 타입으로 두고 구현했었는데

#실행 시간이 1시간이 넘어서 무척 비효율적이었음

possibility[word] = [goodP, badP]

61 return possibility, total\_goodP, total\_badP

* 1. training() : 원래는 출력문을 통해 한 차례 training data set의 이름을 받고, training 함수의 인자로 넣으려 고 했으나, 과제물의 DataSet 자체가 ratings\_train.txt 파일 하나여서 그냥 코드 안에 파일명을 삽입하는 방 식으로 처리함. 해당 함수는 위에서 구현한 함수들을 갖고 training을 거쳐 최종적으로 possibility table을 구 하고, classiﬁcation 과정에서 사용될 P(긍정)과 P(부정)을 구하여 반환하는 함수. 해당 함수까지가 **'1. 문제 해 결 방식'에서 언급한 training 절차**.

1

2

3

4

5

6

7

8

9

10

11

12

13

#트레이닝 데이터 셋을 통해 학습 후 확률에 대한 표를 반환하는 함수 : possibility와 순서

가 동일한 단어모음 리스트 반환

def training():

#ratings\_train.txt 파일을 읽어서 받아온 리뷰들의 리스트 : origin\_reviews origin\_reviews = getReviews('ratings\_train.txt') #origin\_reviews를 긍정 리뷰와 부정 리뷰로 분할함

goodReviews, badReviews = sortReviews(origin\_reviews)

#긍정 리뷰에 대한 { 단어 : 단어 등장 횟수 } 스타일의 딕셔너리

good\_count = makeCount(goodReviews)

#부정 리뷰에 대한 { 단어 : 단어 등장 횟수 } 스타일의 딕셔너리

bad\_count = makeCount(badReviews)

#리뷰에 등장하는 전체 단어에 대한 { 단어 : [긍정 리뷰 등장 확률, 부정 리뷰 등장 확률]

} 스타일의 딕셔너리와 P(긍정), P(부정)

14 possibility, total\_goodP, total\_badP = makePossibility(good\_count, bad\_count, goodReviews, badReviews)

15

16 return possibility, total\_goodP, total\_badP

* 1. classify(possibility, total\_goodP, total\_badP, newReview) : training()의 리턴 값 세 가지와 새로운 리 뷰 문장을 인자로 받음. 위의 트레이닝 결과를 활용하여 새로운 리뷰가 긍정인지 부정인지에 따라서 1과 0을 리 턴하는 함수. 확률 값 자체가 매우 작으므로 **log를 취함으로써 컴퓨팅을 용이하게 함**. 원래는 확률의 곱이지만 log 변환 때문에 합으로 연산.
     1. #새로운 리뷰에 대해서 긍정인지 부정인지 판단해주는 함수
     2. def classify(possibility, total\_goodP, total\_badP, newReview): 3

1. #새로운 리뷰를 리스트로 변환
2. newWords = list(newReview.split()) 6
3. #각 단어의 긍정 확률, 부정 확률 변수를 0으로 초기화
4. log\_goodP = 0.0
5. log\_badP = 0.0 10
6. #새로운 리뷰 리스트 안에 있는 단어 각각에 대한 for loop
7. for word in newWords:
8. #단어가 possibility table에 존재할 경우
9. #즉, training set 에 존재할 경우
10. if word in possibility:
11. #딕셔너리 타입의 possibility에서 긍정 확률과 부정확률을
12. #key 값인 word를 통해 찾아냄
13. value = possibility[word] 19
14. #로그 연산이므로 확률의 곱을 로그의 합으로 표현함
15. log\_goodP += math.log(value[0])
16. log\_badP += math.log(value[1]) 23
17. #단어가 possibility table에 존재하지 않을 경우
18. elif word not in possibility:
19. #긍정 확률을 0.829로 처리하고
20. log\_goodP += math.log(0.829)
21. #부정 확률을 0.9로 처리하였더니 가장 정확도가 높게 나옴.
22. #이 부분은 valid text file에 한저된 것일 수 있으나
23. #프로젝트 진행과정에서 최선이라 생각함.
24. log\_badP += math.log(0.9) 32
25. goodP = log\_goodP + math.log(total\_goodP)
26. badP = log\_badP + math.log(total\_badP) 35
27. #단어 위치에 따른 오류 이슈 해결하는 부분
28. if goodP > badP:
29. if exceptions('안', '좋', newReview) == 0:
30. return 0

40

41

42

43

44

45

46

47

48

49

50

51

52

53

54

55

56

57

58

59

60

61

62

63

if exceptions('별로', '좋', newReview) == 0:

return 0

if exceptions('안', '재미', newReview) == 0: return 0

if exceptions('별로', '재미', newReview) == 0:

return 0

if exceptions('별로', '재밌', newReview) == 0: return 0

if exceptions('안', '재밌', newReview) == 0:

return 0

if exceptions('재미', '없', newReview) == 0: return 0

if exceptions('생각보다', '별로', newReview) == 0:

return 0

return 1

else:

if exceptions('생각보다', '괜찮', newReview) == 0: return 1

if exceptions('기대보다', '괜찮', newReview) == 0:

return 1

return 0

goodP > badP 일 경우 classiﬁcation의 결과, 새로운 리뷰는 긍정을 의미. 하지만 '좋', '재미', '재밌'과 같은 긍정을 표현하는 형태소 앞에 '안','별로'와 같은 부정적인 표현이 올 경우, 해당 리뷰는 부정임. 이에 대한 처리를 적절히 하지 못한 경우에 대비하여 exceptions()라는 함수를 만들어, 긍정 단어와 부정 단어의 위치 관계에 따 라 발생할 수 있는 분류의 오류를 처리하고자 노력함. 해당 방식이 100% 정확하게 오류를 거를 수 없으나, 해당 방식을 적용하기 전에 비해서 정확도가 5% 증가함.

* 1. exceptions(nope, fun, newReview) : 두 단어와 새로운 리뷰를 인자로 받음. 두 단어가 새로운 리뷰에 있을 경우, 각각의 위치에 대한 index 값을 비교하여 위의 classify() 함수에서 단어 위치에 따란 오류 문제 해결을 돕 는 함수

1

2

3

4

5

6

7

8

9

10

11

12

13

14

def exceptions(nope, fun, newReview):

flag = 2

#새로운 리뷰에 'fun' 이라는 단어가 있고

if fun in newReview:

#'fun'이라는 단어의 위치를 index\_fun이라고 함

index\_fun = newReview.find(fun)

#새로운 리뷰에 'nope' 이라는 단어가 있고

if nope in newReview:

#'nope'단어의 위치를 index\_nope이라고 함

index\_nope = newReview.find(nope)

#index\_fun의 위치가 index\_nope의 위치보다 더 뒤에 있다면

15

16

17

18

19

20

21

22

23

if index\_fun > index\_nope:

#0을 return flag = 0

#index\_fun의 위치가 index\_nope의 위치보다 앞에 있다면

else:

#변화 없음

flag =

return flag

* 1. applyNewReview(possibility, total\_goodP, total\_badP, newFileName) : training 결과인 possibility table과 P(긍정), P(부정), 새로운 리뷰들이 있는 텍스트 파일명을 인자로 받음. classiﬁcation 진 행 후 결과를 ratings\_result.txt로 만들어 주는 함수.

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

#새로운 리뷰 파일을 받아와서 classify 결과를 포함한 txt file을 반환하는 함수

def applyNewReview(possibility, total\_goodP, total\_badP, newFileName):

# 새로운 리뷰 파일의 분류 결과 파일 - 쓰기 모드로 오픈

resultFile = open('ratings\_result.txt', 'w', encoding="UTF-8") # 새로운 리뷰 파일 - 읽기 모드로 오픈

newReviewFile = open(newFileName, 'r', encoding="UTF-8")

# 새로운 리뷰를 한 줄 씩 읽음

for line in newReviewFile:

if line.startswith('id'): resultFile.write(line)

# 리뷰에 대한 classify 진행 후 결과를 결과 파일에 작성

else:

result = classify(possibility, total\_goodP, total\_badP,

line)

16

resultFile.write(line.rstrip('\n') + '

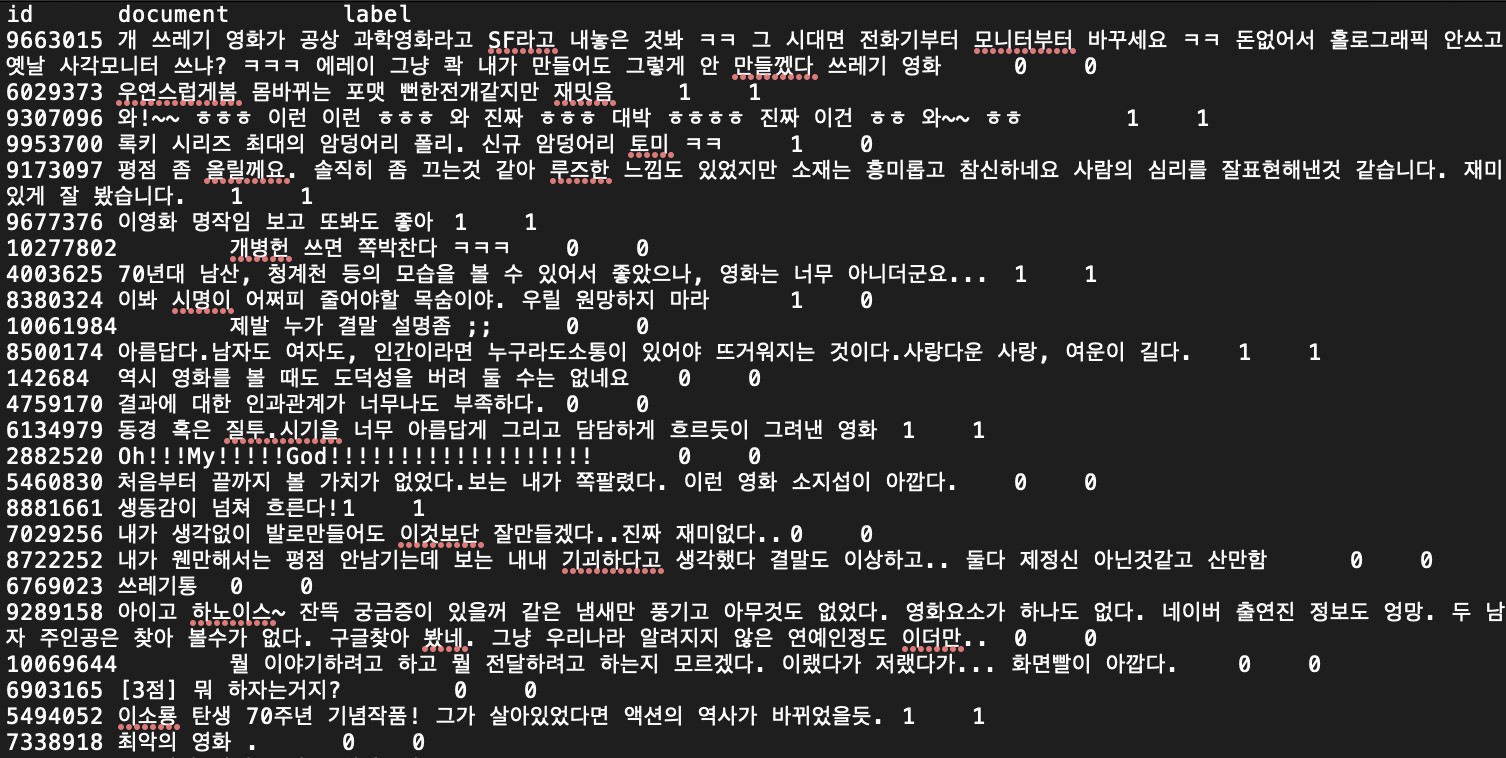
' + str(result)

+ '\n')

# 3. Result

1. ratings\_valid에 대한 트레이닝 결과 후 original labeling 과 비교한 정확도 : 85.99% (\* exceptions 함수 적용 전에는 80.19%가 최대였음)

2. ratings\_test에 대한 트레이닝 결과인 ratings\_result



# 4. 추가 Rating

ratings\_data 폴더에 제공된 ratings\_valid.txt 의 label값을 제거한 파일에 대해 ratings\_train.txt를 통한 학습을 바탕으로 classiﬁcation 진행. 결과 파일을 ratings\_result\_valid.txt라 하고 기존의 label 값과 비교하는 과정을 거 쳐 accuracy를 백분율로 표현.

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

valid\_traied = open('ratings\_result\_valid.txt', 'r', encoding="UTF-8")

valid\_origin = open('ratings\_valid.txt', 'r', encoding="UTF-8")

count = 0

length = 0 trained = [] origin = []

for line\_t in valid\_traied:

tmp = list(line\_t.split())[-1] trained.append(tmp)

length += 1

for line\_o in valid\_origin:

tmp = list(line\_o.split())[-1] origin.append(tmp)

for i in range(length):

if trained[i] == origin[i]: count += 1

print((count/length) \*100)

# 5. 어려웠던 점 & 한계점 & 문제 해결점 등

1. list type을 사용하여 count와 possibility를 제작하고 train을 돌렸더니 계속 해서 한시간이 넘게 걸렸다. 한 번의 결과를 보기 위해서도 너무 오랜 시간을 기다려야함이 힘들었다.

2. dictionary type을 사용했더니 너무 빨리 끝나서 맨처음에는 제대로 돌고 있는게 맞나 고민할 정도였다. 근데

list type을 사용했을 때와 결과가 같은 것을 보고 매우 감탄 했다!!!

3. 최적화 방식에 대해서 따로 수업시간에 배운 것이 없어서 적용하기가 좀 어려웠다.

4. 영화 리뷰의 특성상 명확히 긍정과 부정을 분류하기가 어려웠고 이로 인한 불가피한 부정확한 트레이닝이 발생 하는 거 같다