Assignment 2

2013002545 정광은

1. 실험

형태소 분석기를 사용한다는 가정 아래에, 다음과 같이 성능을 높일 수 있을 것 같은
방안들을 생각해보았다. 최종적으로는 실제 18만개의 데이터로 측정하면 좋겠지만 실험은
시간과 자원이 부족하여 대략 3000개 정도의 데이터로 트레이닝한 후, 3000개의 모의 데이터에 대한
success rate을 측정해 보았다. [0]번은 Konlpy의 Kkm, 그 외에는 Konlpy의 Okt를 형태소 분석기로 사용하였고
쿼리중 학습해본적이 없는 단어의 경우 무시하기로 하였다.

다음과 같은 기준에 변화를 주어가며 실험을 해보기로 하였다.

[0] All Type vs Selective Type
- 품사구분 없이 전부 고려하기 VS 특정 품사만 고려하기

[1] Stemming VS No Stemming
- 단어를 원형으로 바꾸기 VS 나온 단어 그대로 사용하기

[2] Weighted Type vs No-Weighted Type
- 특정 품사에 Weight 주기 vs 모든품사는 동일하게 취급하기

[3] Verb Prefix Concatenation vs No-Verb Prefix Concatenation
- '안'하다, '못'하다를 '안하다' '못하다'로 묶기 vs 그냥 두기

[4] Smoothing 의 정도
- 한번도 세지지 않은 경우에도 계산상 확률을 0이라고 할 순 없는데,
이때 어느정도 작은 확률을 부여할 것인가

[0] All Type vs Selective Type
모든 품사 선택 VS 정보성이 높은 품사만 선택

기대 : 영어에서는 to 조차도 여러가지 의미를 갖지만 한국어에는 정말 의미가 없을 수도 있는
'은는이가을를'같은 단어들이 있기 때문에, 기본적으로 조사를 포함한 여타 품사를 제외하는 것이 무조건
나을 거라는 것이 초기 기대였다.

과정 : 검색해보니 kkma 라는 분석기가 가장 자세한 품사태깅을 제공하고 있었다.
사실 너무 다양한 품사가 있어서, 어떤 품사가 정보성이 높을지 알지 못했기 때문에
data를 형태소로 쪼개 본 예시를 보고 다음과 같은 품사를 골라내었다.

꼬꼬마		
태그	설명	
EC	연결 어미	
ECD	의존적 연결 어미	은데, ㄹ지, 라면, 니까, 길래
ECE	대등 연결 어미	좋'고', 었'ㄴ데', 보'면서',소통'ㄴ데',돌아가'고', 나오'고'
ECS	보조적 연결 어미	야, 어, 는지, 다
EF	종결 어미	
EFA	청유형 종결 어미	X
EFI	감탄형 종결 어미	구나, 구려, 는군, 군요
EFN	평서형 종결 어미	다, 으랴, 네, 네요, 습니다, ㅂ니다, 네, 어요, ㄴ다, 세요, 다고, 습니다
EFO	명령형 종결 어미	어라
EFQ	의문형 종결 어미	나요, 나, 냐, ㄴ가, 을까, 라니, 을까요, 자고
EFR	존칭형 종결 어미	X
ET	전성 어미	
ETD	관형형 전성 어미	ㄴ,ㄹ,는,은,을
ETN	명사형 전성 어미	기,ㅁ,음
EP	선어말 어미	있었'겠'지만, 빠지'었'어요, 해'었'던영화,
EPH	존칭 선어말 어미	하'시'ㄹ분, 싫'으시'면, 읽어주'시'고
EPP	공손 선어말 어미	X
EPT	시제 선어말 어미	나오'았'다, 모르'겠'다, 미치'었'다
IC	감탄사	젠장, 에라
JK	조사	
JC	접속 조사	유준상'이랑', 빨리지'나'가요, 수갑끊으려'하고', 용기'와'열정, 영화'랑'비슷
JKC	보격 조사	영화가/명작이/전설이/ 되는
JKG	관형격 조사	누구누구-의
JKI	호격 조사	청춘이'여'
JKM	부사격 조사	스토리'에', 채널'에서', 국민'으로써', 마음'에'
JKO	목적격 조사	을, 를
JKQ	인용격 조사	X
JKS	주격 조사	은는이가
JX	보조사	기계'도'없는데, dvd'까지'산, 재미'도'없고, 연기'는'좋다
MA	부사	
MAG	일반 부사	돈'많이', 영상'마저', '너무'좋다, '참'재미있다
MAC	접속 부사	하지만
MD	관형사	
MDN	수 관형사	'한'줄, '두'인종, '한'회, '두'번
MDT	일반 관형사	'그'당시, '이'영상
NN	명사	
NNG	보통명사	흥행, 영화, …
NNB	일반 의존 명사	한다는'것'자체가, 죽는'줄'알았다, 배우'중', 마무리짓는 '수'밖에
NNM	단위 의존 명사	10'점', 1'초'
NNP	고유명사	심형래, 양동 - 근데 부정확한 경우가 많았음
NP	대명사	개'뭐'같이, 그녀, '이것', '여기'서, '이것'을
NR	수사	8, 하나, '3'점, '다섯'개
OH	한자	
OL	외국어	dvd, cgv, sf, 'c'급
ON	숫자	X
SE	줄임표	…
SF	마침표, 물음표, 느낌표	!, ., ?
SO	붙임표(물결, 숨김, 빠짐)	~
SP	쉼표, 가운뎃점, 콜론, 빗금	,
SS	따옴표, 괄호표, 줄표	
SW	기타기호 (논리/수학기호, 화폐기호)	.. , !!! , ??
VA	형용사	괜찮, 좋, 나쁘
VXA	보조 형용사	죽이고'싶'은, 교훈을주려'듯하'고, 꼽을'만한', 볼'만한'
VC	지정사	
VCN	부정지정사 형용사 '아니다'	~은'아니'ㄹ까, 뿐'아니'라, B급도'아니'
VCP	긍정지정사, 서술격조사 '이다'	이다
VV	동사	
VXV	보조 동사	때묻지'않'았다, 나쁘지는'않'았다, 보고'있'어요, 답답해어'지'다, 잊히어'지'지'않'는다
VX	보조 용언	하지'못하'다, 끌지'못하'다
XP	접두사	
XPN	체언 접두사	저'예산, '맨'날 - 근데 오류가 많음
XPV	용언 접두사	X
XSA	형용사 파생 접미사	불쌍'하'다, 훌륭'하'다
XSN	명사파생 접미사	군인'들'의, 배우'들'의
XSV	동사 파생 접미사	발전'되', 허무'하'게, 표현'한' 느낌
XR	어근	'담백'한, '기발'한
UN	명사추정범주	쓰렉, 쒸, 빡

정보성이 낮은 품사

정보성이 중간인 품사

정보성이 높은 품사

```
    refined = kkma.pos(targetSentence) # refined = 형태소로 쪼개진 하나의 문장
    for item in refined:
        if item[1] in ['ECD', 'EFI', 'EFO', 'EFQ', 'IC', 'JX', 'MAG', 'MAC',
'NNG', 'NNM', 'NR', 'OL', 'ON', 'SE', 'SF', 'SO', 'SW', 'VA', 'VXA', 'VCN', 'VV',
'VX', 'VX', 'XR', 'UN']:
                # wordpool 에 넣는다
```

결과 :
원대한 포부와 달리 모든 품사를 포함하는 것이 오히려 근소하나마 성능이 좋았다.
이는 미련을 버리지 못하고 이 다음 단계인 'Stemming'나 서 테스트를 해봤을 때나
'Weight'값에 변화를 주어 다시 계산해봤을때도 일관적이였다.
조사와 같은 품사는 문법적인 장치일 확률이 많고 긍정이든 부정이든 쓰는 조사를 쓰는 빈도는 비슷하기 때문에 생각보다 노이즈가 크진 않았던 것 같다.

- **All Type**, No Stemming, No Weighted Type, No Verb Prefix Concatenation : success rate = **0.75775**
- **Selective Type**, No Stemming, No Weighted Type, No Verb Prefix Concatenation : success rate = 0.74342

- **All Type**, No Stemming, Weighted Type, No Verb Prefix Concatenation : success rate = **0.74908**
- **Selected Type**, No Stemming, Weighted Type, No Verb Prefix Concatenation : success rate = 0.73409

[1] Stemming VS No Stemming

기대 : 실제로는 같은의미지만 다른 형태로 표현되어 영향력이 적던 단어가, 더 큰 영향을 미치게 되어 성능을 높일 것이다.

과정 : kkma에서는 stemming을 제공하지 않아 Okt라는 형태소 분석기를 사용하였다. 품사를 모두포함/선별포함하는 변화에도 stemming의 효과가 일관적으로 나타나는지도 확인해 보았다.

```
    refined = okt.pos(targetSentence, norm=True, stem=True)
```

결과 :
근소한 차이이나 stemming은 품사를 선택해 넣든 아니든 **일관적으로** 성능을 높이는데 일조함을 확인하였다.

- All Type, **Stemming,** No Weighted Type, No Verb Prefix Concatenation : success rate = **0.75808**
- All Type , **No stemming,** No Weighted Type, No Verb Prefix Concatenation : success rate = 0.74508

- Selective Type, **Stemming,** No Weighted Type, No Verb Prefix Concatenation : success rate = **0.75641**
- Selective Type, **No Stemming,** No Weighted Type, No Verb Prefix Concatenation : success rate = 0.73775

[3] Weighted Type VS No-Weight Type

기대 : 지금까지 해보았을 때 생각보다 품사가 차이를 많이 내지는 못하였지만,
여전히 실제 data를 보았을 때 가장 의견을 함축적으로 의미할 수 있는 것은 명사/동사/형용사/부사등 몇몇 품사일 것 같았다. 그래서 이러한 단어에 대해서는 X1.2배 정도로 weight를 줘보면 좀 더 성능이 높아지지 않을까 생각해 보았다.

과정 : 특정 품사인 경우, count X 1.2 를 하여 로그를 계산한다

```
# wordpool[idx] = ["단어", "품사" S|P, -S|P, S|N, -S|N, Bool]
if countPos == 0:
    wordpool[idx][2] = log(1/(4*numAll),2)
    wordpool[idx][3] = log(1-(1/(4*numAll)),2)
else:
    if wordpool[idx][1] in ['Adjective', 'Noun', 'Verb', 'Adverb']:
        wordpool[idx][2] = log(countPos*1.2/numPos, 2)
        wordpool[idx][3] = log(1-(countPos*1.2/numPos), 2)
    else:
        wordpool[idx][2] = log(countPos/numPos, 2)
        wordpool[idx][3] = log(1-(countPos/numPos), 2)
if countNeg == 0:
    wordpool[idx][4] = log(1/(4*numAll), 2)
    wordpool[idx][5] = log(1-(1/(4*numAll)), 2)
else:
    if wordpool[idx][1] in ['Adjective', 'Noun', 'Verb', 'Adverb']:
        wordpool[idx][4] = log(countNeg*1.2/numNeg, 2)
        wordpool[idx][5] = log(1-(countNeg*1.2/numNeg), 2)
    else:
        wordpool[idx][4] = log(countNeg/numNeg, 2)
        wordpool[idx][5] = log(1-(countNeg/numNeg), 2)
```

결과 :
이 정도면 차이가 없다고 봐야할 것 같긴 하지만 weight을 안주는게 오히려 나았다.

- All Type, Stemming, **Weighted Type**, No Verb Prefix Concatenation : success rate = 0.75708
- All Type, Stemming, **No-Weight Type,** No Verb Prefix Concatenation : success rate = **0.75808**

[4] Prefix, Verb Concatenation

기대 : 형태소를 조건없이 분석해놓은 데이터를 보고 있으니 Okt에서 'VerbPrefix'라는 품사가 거의 대부분 '안보다', '못보다', '못하다' 에서의 부정어 '안', '못'임을 발견하였다. 물론 떨어져 있어도, 어차피 각자 카운트가

되긴 하겠지만 실제 의미는 '안'과 '못'을 동사와 합해야 이루어질 것 같아 이러한 verb prefix를 그 다음에 오는 동사와 합해서 분석하는 방법을 생각해보게 되었다.

과정 : Prefix가 나올시 단어모음집에는 넣지 않고 따로 저장해 두었다가,
그 다음에 Verb가 나오면 합해서 저장한다.

```
# refined 는 형태소로 쪼개진 하나의 쿼리문장
# refined[idx] = [('재밌다','verb'),('영화','noun'),('재밌다', 'verb')]

# ① '못', '안' 등은 다른 단어처럼 넣지 않고, verbPrefix에 저장
if refined[idx][1] == 'VerbPrefix':
    if idx+1 <= len(refined)-1:

    # 혹시 그 다음 'Verb'가 안나오는 경우엔 아무것도 하지 않음
          if refined[idx+1][1] == 'Verb':
                verbPreFix = refined[idx][0]
    else:
          # ② 그 다음 동사가 나오면 저장해둔 verbPreFix를 붙인 형태를 저장
          if refined [idx][1] == 'Verb':
                if verbPreFix != None:
                    tempTuple = (verbPreFix + ' ' +
                                 refined[idx][0], refined[idx][1])
                    del refined[idx]
                    refined.insert(idx, tempTuple)
                    verbPreFix = None
```

결과 :
거의 차이가 없지만 concatenation을 하는게 조금 더 성능을 향상시키긴 하였다.

- All Type, Stemming, No-Weighted Type, **Verb Prefix Concatenation** : success rate = **0.75908**
- All Type, Stemming, No-Weighted Type, **No Verb Prefix Concatenation** : success rate = 0.75808

[5] Smoothing 정도

기대 : 한번도 나오지 않는 단어라도 0을 붙이면 계산이 안되기 때문에 0에 가까운 작은 확률을 부여해야 했는데, 그 확률을 작게 주는 것이 실제와 가까우니 낫겠다는 생각은 들었다. 그러나 차이를 어느정도 낼 지는 예상하기 힘들어서 실험을 해보기로 했다.

과정 : 지금까지는 어떤 word가 특정경우에 한번도 나오지 않았다면 $\log_2(1/(4*\text{전체개수}))$ 로 계산을 해왔었는데 $\log_2(1/(1*\text{전체개수})$, $\log_2(1/(2*\text{전체개수})$, $\log_2(1/(16*\text{전체개수})$, $\log_2(1/(32*\text{전체개수}))$ 로 계산해보았다.

```
# wordpool[idx] = ["단어", "품사" S|P, -S|P, S|N, -S|N, Bool]
if countPos == 0:
    wordpool[idx][2] = log(1/(1*numAll),2)
    wordpool[idx][3] = log(1-(1/(1*numAll)),2)
else:
    wordpool[idx][2] = log(countPos/numPos, 2)
    wordpool[idx][3] = log(1-(countPos/numPos), 2)
if countNeg == 0:
    wordpool[idx][4] = log(1/(1*numAll), 2)
    wordpool[idx][5] = log(1-(1/(1*numAll)), 2)
else:
    wordpool[idx][4] = log(countNeg/numNeg, 2)
    wordpool[idx][5] = log(1-(countNeg/numNeg), 2)
```

결과 :

정말 뜻밖에 그냥 0을 1로 세는, 즉 실제보다 더 큰 확률을 쳐주었을 때 오히려 가장 success rate이 높았다. 이는 기대한 바와 부합도 아니고 반대되는 내용이였다.

- All Type, Stemming, No-Weighted Type, No Verb Prefix Concatenation, $\log_2$(1/(**1***전체개수)) : success rate = **0.76841**
- All Type, Stemming, No-Weighted Type, No Verb Prefix Concatenation, $\log_2$(1/(**4***전체개수)) : success rate = 0.75808
- All Type, Stemming, No-Weighted Type, No Verb Prefix Concatenation, $\log_2$(1/(**16***전체개수)) : success rate = 0.74475
- All Type, Stemming, No-Weighted Type, No Verb Prefix Concatenation, $\log_2$(1/(**32***전체개수)) : success rate = 0.74042

결론 :

여러가지 시도에도 3000개 training data로 학습된 이 분석기의 성능은 80%을 넘지를 못하였고, 변동사항을 적용해도 나오는 차이는 1% 내외였다. 그래도 실제로 18만개의 데이터로 트레이닝후에는 ratings_valid.txt 에 대하여 **83.72% 정도의 success rate**을 얻을 수 있었다.

All Type, Stemming, No-Weighted Type, Verb Prefix Concatenation, $\log_2$(1/(전체개수))

→ **success rate = 0.83712**

2. 선택된 조합의 코드 설명

```
# -*- coding: utf-8 -*-

from konlpy.tag import Kkma
from konlpy.tag import Twitter
from konlpy.tag import Okt
from math import log
from random import randint
import json

okt = Okt()
kkma = Kkma()

# 에러를 대비해서 단계마다 json 파일을 만듦
def makeJson(filename, data):
    with open(filename, 'w', encoding="utf-8") as make_file:
        json.dump(data, make_file, ensure_ascii=False, indent="\t")

def loadJson(filename):
    with open(filename) as f:
        data = json.load(f)
    return data

# 형태소 분석 + 사용단어 추출
def tokenize(filename, order):
    print(f"{order} : tokenize 중")
    data = []
    wordpool = []
    countPos = 0
    countNeg = 0
    count = 0
    verbPreFix = None

    with open(filename, 'r', encoding='utf8') as oldF:
        for line in oldF:
            if count == 0:
                count += 1
                continue
            else:
                # ① id를 제외하고 문장만추출
                temp = line.split()
                del temp[0]
```

```
if order == 'train':
    label = temp[-1]
    if label == '1':
        countPos += 1
    if label == '0':
        countNeg += 1
    del temp[-1]
targetSentence = ' '.join(temp)

# ② 형태소분석기로 쪼갬

refined = okt.pos(targetSentence, norm=True, stem=True)
buffer = []



# ③ 사용된 단어는 중복되지 않도록 wordpool이라는 자료구조에 넣음

for idx in range(len(refined)):

    # refined[idx] = [('재밌다','verb'),('영화','noun'),('재밌다', 'verb')]

    if refined[idx][0] != ".":
        if refined[idx][1] == 'VerbPrefix':
            if idx+1 <= len(refined)-1:
                if refined[idx+1][1] == 'Verb':
                 verbPreFix = refined[idx][0]
        else:
            if refined [idx][1] == 'Verb':
                if verbPreFix != None:
                    tempTuple = (verbPreFix + ' '
                                    + refined[idx][0], refined[idx][1])
                    del refined[idx]
                    refined.insert(idx, tempTuple)
                    verbPreFix = None
            if order == 'train':
                if [refined[idx][0],refined[idx][1],0,0,0,0,0]
                                                                not in wordpool:
                    wordpool.append([ refined[idx][0] ,
                              refined[idx][1] ,0,0,0,0, 0])
    buffer.append( [ refined[idx][0],refined[idx][1] ] )

# ④ 쪼개진 형태소는 다시 모아서 data라는 자료구조에 넣어줌

if order == 'train':
    buffer.append(label)
data.append(buffer)
count += 1
```

```
    if order == 'train':
        data.append([countPos, countNeg])

    print(f"{order} : tokenize완료")
    dict = {'refinedData' : data, 'wordpool' : wordpool}
    return dict

# 어떤 data set에서 특정 (word,type)이 긍부정에서 각각 몇번 쓰였는지를 리턴함
def countPosNeg(word, type, data):
    countPos = 0
    countNeg = 0

    # sentence : [[영화,명사],[별로다,형용사],'0']
    for sentence in data[:len(data)-1]: #마지막엔 전체 pos,neg의 개수가 들어있음
        for idx in range(len(sentence)-1):
            if sentence[idx][0] == word and sentence[idx][1] == type:
                if sentence[-1] == '1':
                    countPos += 1
                    break               # 한번 나온게 확인 되었으면 그 문장에서 세는건 그만 둔다
                if sentence[-1] == '0':
                    countNeg += 1
                    break
    return {'countPos' : countPos, 'countNeg' : countNeg}

# training set에서 사용된 word들의 조건부확률을 계산
def labelWordPool(data, wordpool):
    print("wordpool 통계분석 시작")

    # training data에서 긍정/부정 셋의 개수
    numPos = data[-1][0]
    numNeg = data[-1][1]
    numAll = numPos+numNeg

    for idx in range(0, len(wordpool)):

        countDict = countPosNeg(wordpool[idx][0], wordpool[idx][1], data)
        countPos = countDict['countPos']
        countNeg = countDict['countNeg']

        # wordpool[idx] = ["단어", "품사" log(S|P), log(-S|P), log(S|N), log(-S|N), Bool]
        # 마지막 bool은 나중에 Test Data Set에서, 해당 단어가 사용되었는지를 표시할 때 쓰려고 놔둠
        if countPos == 0:
```

```
            wordpool[idx][2] = log(1/(1*numAll),2)
            wordpool[idx][3] = log(1-(1/(1*numAll)),2)
        else:
            wordpool[idx][2] = log(countPos/numPos, 2)
            wordpool[idx][3] = log(1-(countPos/numPos), 2)
        if countNeg == 0:
            wordpool[idx][4] = log(1/(1*numAll), 2)
            wordpool[idx][5] = log(1-(1/(1*numAll)), 2)
        else:
            wordpool[idx][4] = log(countNeg/numNeg, 2)
            wordpool[idx][5] = log(1-(countNeg/numNeg), 2)

    # 전체 데이터에서 pos와 neg의 확률도 넘겨줌
    pos = log(numPos / (numPos + numNeg), 2)
    neg = log(numNeg / (numPos + numNeg), 2)
    wordpool.append([pos,neg])
    return {'wordpool' : wordpool}

# wordpool 자료구조에서 sentence안에 사용된 단어를 사용되었다고 표시해줌
def markUsedWord(wordpool, sentence, exclude):
    for targetWord in sentence[:len(sentence)-exclude]:
        for idx in range(len(wordpool)-1): #마지막에는 logPos, logNeg정보가 들어있음
            if wordpool[idx][0] == targetWord[0] and wordpool[idx][1] == targetWord[1]:
                wordpool[idx][6] = 1
    return wordpool

# 나이브 베이지안 모델에 근거해 labeling
def naiveBayes(data, order):
    print(f"{order} : 나이브베이지안")

    # training data는 test data와 달리 끝에 붙어있는 라벨이 있으므로 계산시 제외해야 함
    if order == 'train':
        excludeLabel = 1
        excludeCount = 1
    if order == 'test':
        excludeLabel = 0
        excludeCount = 0

    for index in range(0,len(data)-excludeCount):
        pos = 0
        neg = 0
        wordpool = loadJson('wordpool.json') # 모든 쿼리는 처음에는 Bool 부분이 모두 0인 wordpool이 필요함
        logPos = wordpool[-1][0]
```

```
        logNeg = wordpool[-1][1]
        wordpool = markUsedWord(wordpool, data[index], excludeLabel) # 해당쿼리에서 사용된 단어가
                                                                     # wordpool의 Bool부분에 표시됨

        # item = ["단어", "품사" log(S|P), log(-S|P), log(S|N), log(-S|N), Bool]
        for item in wordpool[:len(wordpool)-1]: # 마지막에는 [logPos, logNeg] 정보가 들어있음
            if item[6]==1:
                pos += item[2]
                neg += item[4]
            if item[6]==0:
                pos += item[3]
                neg += item[5]

        # 실제 그 라벨일 확률도 더해줌
        pos += logPos
        neg += logNeg

        if pos < neg:
            data[index].append('0')
        elif pos > neg:
            data[index].append('1')
        else:
            data[index].append(f"{randint(0,1)}")
    return data

# ratings_valid 의 성공률을 반납
def checkSuccess(data):
    count = 0
    correct = 0
    for item in data:
        count += 1
        if item[-1] == item[-2]:
            correct += 1
    return round(correct/count, 5)

# ratings_result_txt에 쓰는 부분
def writeResult(filepath, data):
    with open(filepath) as fp:
        lines = fp.read().splitlines()
    with open('ratings_result.txt', "w") as fp:
        count = 0
        for line in lines: # 첫번째줄 제외
            if count == 0:
```

```
                count += 1
                continue
            else:
                if type(data[count-1][-1])==list:
                    print(f"data[count-1][-1] = {data[count-1][-1]}")
                print(line + data[count-1][-1], file=fp)
                count += 1

def main():

    # Training / json 만들기
    dict = tokenize('ratings_train.txt', 'train')
    wordpoolDict = labelWordPool(dict['refinedData'],dict['wordpool'])
    wordpool = wordpoolDict['wordpool']
    makeJson('ratings_train.json', dict['refinedData'])
    makeJson('wordpool.json', wordpool)

    # Training 데이터를 json에서 불러옴
    refinedData = loadJson('ratings_train.json')
    wordpool = loadJson('wordpool.json')
    logPos = wordpool[-1][0]
    logNeg = wordpool[-1][1]

    # valid 데이터 분석 / json 만들기
    validDict = tokenize('ratings_valid.txt', 'train')
    makeJson('ratings_valid.json', validDict['refinedData'])
    validData = loadJson('ratings_valid.json')
    labeledValidData = naiveBayes(validData, 'train')
    makeJson('naive.json', labeledValidData)
    labeledValidData = loadJson('naive.json')
    print(f"success rate = {checkSuccess(labeledValidData)}")

    # test 데이터에 라벨링하기
    dict = tokenize('ratings_test.txt', 'test')
    makeJson('ratings_test.json', dict['refinedData'])
    testData = loadJson('ratings_test.json')
    labeledTestData = naiveBayes(testData, 'test')
    makeJson('result.json', labeledTestData)
    writeResult('ratings_test.txt',labeledTestData)

main()
```