Assignment2

인공지능

2014005114

컴퓨터 전공

정근욱

1. 목표

   Naïve Bayes Classifier 를 이용해 영화평 분류하기

2. 용어

   - 문서: 트레이닝 데이터에 있는 데이터 하나하나
   - 단어: 문서를 토큰화 한 것

     ex) 문서: "apple is good" => 단어: {'apple', 'is, 'good'}
   - 딕셔너리: 트레이닝 데이터에 있는 모든 단어를 모은 것

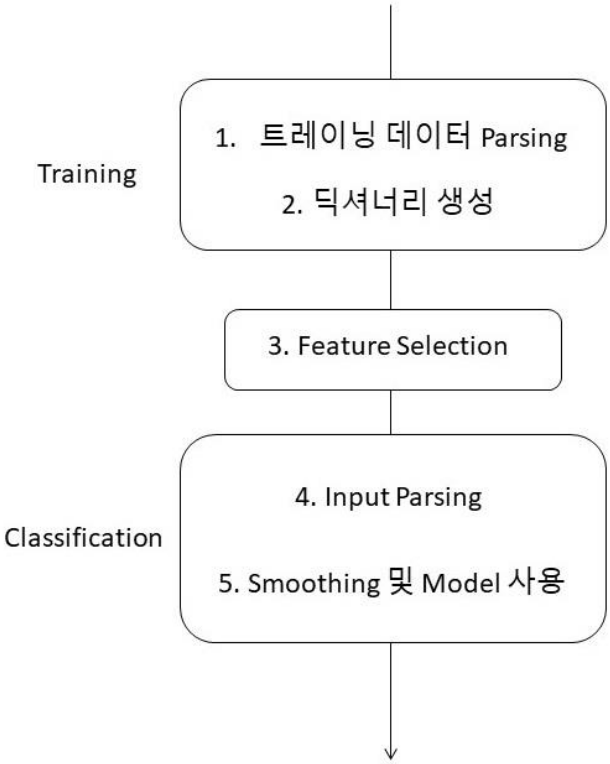
     ex) 문서 1: "apple is good", 문서 2: "banana is bad"

     =====> 딕셔너리: {'apple', 'is', 'good', 'banana', 'bad'}
   - 쿼리 문장: 라벨을 붙이기 위해 Naïve Bayes Classifier 에 입력되는 문장

3. 가정

   - 딕셔너리에 있는 모든 단어들은 라벨이 주어졌을 때 조건부 독립이다.
   - 쿼리 문장에 있는 단어들의 순서는 고려하지 않는다.

4. 기법



- Parsing
  - ✓ 훈련 데이터에 있는 문장 또는 쿼리 문장을 토큰으로 나누는 과정.
  - ✓ KoNLPy 라이브러리의 Mecab 클래스를 사용했다. 형태소로 나누는 morphs 메소드와 명사로 나누는 nouns 메소드를 모두 사용해 보았는데 전자의 성능이 더 좋아 형태소로 나누었다.
  - ✓ 중립적인 단어(ex '것', '이', '들')를 무시하기 위해 불용어(Stop Word)를 딕셔너리에서 제외했더니 성능이 소폭 하락하여서 사용하지 않았다.

- Feature Selection
  - ✓ 트레이닝 데이터에 있는 단어 중 classification 에 사용할 단어를 고르는 과정.
  - ✓ 트레이닝 데이터의 크기를 효과적으로 줄여주고 classification 에러를 증가시키는 noise feature 를 제거하는 역할을 한다.
  - ✓ 다음에 있는 Feature Selection 알고리즘을 이용해 utility 값을 구한 후 utility 값이 높은 단어들만 남기고 제거한다.

- Feature Selection Algorithm
  - ✓ Document Frequency: 단어를 포함하고 있는 문서 수 비교
  - ✓ Term Frequency: 전체 트레이닝 데이터에서 각 단어가 몇 번 나타나는지 비교
  - ✓ Mutual Information: 단어가 존재하는지, 존재하지 않는지가 라벨에 얼마나 영향을 미치는지 정보 이론 관점에서 측정한다. 단어가 속한 문서가 모두 같은 라벨을 가지고 있을 때 최대가 되고, 단어가 속한 문서의 라벨 분포가 전체 문서들의 라벨 분포와 같을 때 0 이 된다.

$$\mathrm{N}_{11} = \textit{단어가 존재 하고 라벨이 } \text{"1"인 문서 수}$$

$$\mathrm{N}_{10} = \textit{단어가 존재 하고 라벨이 } \text{"0"인 문서 수}$$

$$\mathrm{N}_{01} = \textit{단어가 존재 하지 않고 라벨이 } \text{"1"인 문서 수}$$

$$\mathrm{N}_{00} = \textit{단어가 존재 하지 않고 라벨이 } \text{"0"인 문서 수}$$

$$N = \mathrm{N}_{11} + N_{10} + N_{01} + N_{00}$$

$$I = \frac{\mathrm{N}_{11}}{N}\log_2\frac{NN_{11}}{(N_{10}+N_{11)(N_{01}+N_{11})}} + \frac{\mathrm{N}_{01}}{N}\log_2\frac{NN_{01}}{(N_{00}+N_{01)(N_{01}+N_{11})}}$$
$$+\frac{\mathrm{N}_{10}}{N}\log_2\frac{NN_{10}}{(N_{10}+N_{11)(N_{00}+N_{10})}} + \frac{\mathrm{N}_{00}}{N}\log_2\frac{NN_{00}}{(N_{00}+N_{01)(N_{00}+N_{10})}}$$

  - ✓ Kai Square: 단어와 라벨이 얼마나 독립적인지 측정한다. 값이 높을수록 비독립적, 즉 단어와 라벨이 많이 연관되어 있다는 뜻이므로 Feature 로 사용한다.

$$\chi^2 = \frac{(\mathrm{N}_{11}+N_{10}+N_{01}+N_{00}) \times (\mathrm{N}_{11}N_{00}-N_{10}N_{01})^2}{(\mathrm{N}_{11}+N_{01}) \times (\mathrm{N}_{11}+N_{10}) \times (N_{10}+N_{00}) \times (N_{01}N_{00})}$$

  - ✓ 이 4 가지 방식에 대한 비교는 뒤쪽에 제시

- Classifying

  ✓ Multinomial Model: 잘 알려진 대로 Naïve Bayes Classifier 는 다음식을 사용해 분류한다.

$$\text{Label} = \text{argmax}(\text{P}(\text{Label}) \prod P(term|Label))$$

하지만 작은 확률을 곱하다 보면 underflow 가 발생할 수 있다. 따라서 우변에 로그를 취한식을 이용한다.

$$\text{L} = \text{argmax}(\log \text{P}(\text{Label}) + \sum_{term\ in\ query} \log P(term|Label))$$

이 모델은 단어들을 셀 때 중복을 허용한다. 예를 들어 "영화는 영화다" 라는 문장이 있다면 '영화'라는 단어를 2 번 센다. 이때 조건부 확률은 다음과 같다.

$$\text{P}(term|Label = L) = \frac{\textit{라벨이 L 인 문서에 term 이 등장하는 횟수}}{\sum_{\textit{모든 단어}} \textit{라벨이 L 인 문서에 단어가 등장하는 횟수}}$$

  ✓ Bernoulli Model: 베르누이 모델은 앞의 모델과 몇 가지 차이가 있다. 먼저, 한 문장에서 같은 단어가 여러 번 단어가 등장할 때 그것을 다 계산한 Multinomial Model 과 달리 Bernoulli 모델은 단어가 여러 번 등장해도 한번만 계산한다. 즉, 중복을 생각하지 않고 단어가 문장에 등장하는지, 안하는지만 확인한다.

두번째로 베르누이 모델은 쿼리 문장에 나오지 않는 단어도 고려한다. 다음 식처럼 등장하지 않는 단어는 1 에서 확률을 뺀 값을 로그를 취해 더한다.

$$\text{L} = \text{argmax}(\log \text{P}(\text{Label}) + \sum_{term\ in\ query} \log P(term|Label) + \sum_{term\ not\ in\ query} \log(1 - P(term|Label))$$

$$\text{P}(term|Label = L) = \frac{\textit{라벨이 L 이고 term 이 등장하는 문서수}}{\textit{라벨이 L 인 문서수}}$$

✓ Laplace Smoothing: 위 두가지 모델을 이용해 확률을 계산하다 보면 두가지 문제가 생긴다. 첫번째는 쿼리 문장에 있는 단어가 딕셔너리에 없는 경우이다. 이 경우에는 그 단어를 무시하고 다음 단어를 계산한다. 딕셔너리에 없는 단어이기 때문에 무시해도 영향은 미미하다.

두번째 경우는 쿼리 문장에 있는 단어가 딕셔너리에는 있지만 조건부 확률이 0 인 경우다. 이 경우에는 Laplace Smoothing 을 사용했다. Laplace Smoothing 은 딕셔너리에 있는 단어들이 기본적으로 한번은 발생한다고 가정한다. 이를 사용해 Multinomial 모델과 Bernoulli 모델에서의 조건부 확률을 바꾸면 다음 두 식이 된다.

$$\text{Multinomial: } \mathrm{P}(term|Label = L) = \frac{\textit{라벨이 L인 문서에 term이 등장하는 횟수} + 1}{\sum_{\textit{모든 단어}}(\textit{라벨이 l인 문서에단어가등장하는 횟수}) + \textit{딕셔너리 단어수}}$$

$$\text{Bernoulli: } \mathrm{P}(term|Label = L) = \frac{\textit{라벨이 L이고 term 이 등장하는 문서수} + 1}{\textit{라벨이 L인 문서수} + \textit{라벨의 가짓수}}$$

5. 코드

- parser 함수

```
from konlpy.tag import Mecab

mecab = Mecab()

def parser(sentence):
    return mecab.morphs(sentence)
```

✓ konlpy 라이브러리의 Mecab 클래스를 사용해서 객체가 가지고 있는 morphs 메소드를 호출했다.

- train 함수

```
def train(file_name):
    print("-"*60)
    print("Start training...")

    start_time = timeit.default_timer()
    dic = dictionary.Dictionary()
    file_path = os.path.join("ratings_data", file_name)

    with open(file_path) as f:
        next(f)

        for i, line in enumerate(f):
            #if i%1000 == 0:
            #    print(i)

            id_, sentence, label = split_doc(line.strip())

            dic.increment_label(label)
            term_list = nlp.parser(sentence)

            already = set()
            for t in term_list:
                if t not in already:
                    dic.add_term_repeat(t, label)
                    dic.add_term_once(t, label)
                    already.add(t)
                else:
                    dic.add_term_repeat(t, label)

    end_time = timeit.default_timer()

    print("End training!")
    print("Training of '{}'".format(file_name))
    print("Training time: {:.3} sec".format(end_time-start_time))
    print("Total {} docs".format(dic.total_doc()))
    print("Good doc:", dic.doc_frequency("1"))
    print("Bad doc:", dic.doc_frequency("0"))
    print("Number of terms:", len(dic.once["term"]))
```

  - ✓ parser 함수를 이용해 트레이닝 데이터의 문자열을 토큰화 한 뒤 딕셔너리를 구현한 dic 객체에 추가했다.

- Feature Selection 관련 함수

```
def doc_frequency(dic, term):
    return (dic.once["term"][term]["0"] + dic.once["term"][term]["1"] )

def term_frequency(dic, term):
    return (dic.repeat["term"][term]["0"] + dic.repeat["term"][term]["1"] )

def mutual_information(dic, term):
    n = dic.cls["0"] + dic.cls["1"] + 4
    n11 = dic.once["term"][term]["0"] + 1
    n10 = dic.once["term"][term]["1"] + 1
    n01 = dic.once["doc"]["0"] - n11 + 1
    n00 = dic.once["doc"]["1"] - n10 + 1

    return (n11*math.log2(n*n11/((n11+n10)*(n11+n01)))
          + n01*math.log2(n*n01/((n00+n01)*(n01+n11)))
          + n10*math.log2(n*n10/((n10+n11)*(n00+n10)))
          + n00*math.log2(n*n00/((n00+n01)*(n00+n10))))/n

def kai_square(dic, term):
    n = dic.cls["0"] + dic.cls["1"]
    n11 = dic.once["term"][term]["0"]
    n10 = dic.once["term"][term]["1"]
    n01 = dic.once["doc"]["0"] - n11
    n00 = dic.once["doc"]["1"] - n10

    return (n*(n11*n00-n10*n01)**2)/((n11+n01)*(n11+n10)*(n10+n00)*(n01+n00))
```

  - ✓ 위에서 설명한 Feature Selection 알고리즘들을 구현했다.

- smoothing 과 model 함수들

```
def smoothing_multinomial(dic, term, label):
    return (dic.repeat["term"][term][label]+1)/(dic.repeat["doc"][label]+len(dic.repeat["term"]))

def smoothing_bernoulli(dic, term, label):
    return (dic.once["term"][term][label]+1)/(dic.cls[label]+len(dic.cls))

def multinomial(term_list, dic, label):
    result = math.log(dic.doc_frequency(label))

    for t in term_list:
        try:
            result += math.log(smoothing_multinomial(dic, t, label))
        except KeyError:
            continue

    return result

def bernoulli(term_list, dic, label):
    result = bernoulli.tmp_result[label]
    result += math.log(dic.doc_frequency(label))

    for t in term_list:
        try:
            result -= math.log(1-smoothing_bernoulli(dic, t, label))
            result += math.log(smoothing_bernoulli(dic, t, label))
        except KeyError:
            continue

    return result
```

  - ✓ 위에서 제시한 Naïve Bayes Classifier 의 두가지 모델을 구현한 multinomial 함수과 bernoulli 함수. 두가지 모두 쿼리 문장에 있는 단어들에 대해 smoothing 을 이용해 확률을 계산한다. KeyError 가 발생할 경우 continue 를 통해 무시하고 넘어가게 했는데, 이는 쿼리 문장에는 있지만 딕셔너리에는 없는 단어를 무시함을 의미한다. 둘 중에 어느 모델을 사용하는지에 따라 다른 smoothing 함수를 사용했다.

- classify 함수

```
def classify(query_term_list, dic, predict_fun):
    predict0 = predict_fun(query_term_list, dic, "0")
    predict1 = predict_fun(query_term_list, dic, "1")

    if predict0 > predict1:
        return "0"
    else:
        return "1"
```

  - ✓ predict_fun 에는 바로 위에 있는 multinomial 함수 또는 bernoulli 함수가 들어간다. 두 모델로 값을 측정한 뒤, 대소 비교 결과에 따라 라벨을 결정한다.

- nbc 함수

```
def nbc(query_file, result_file, dic, predict_fun):
    print("-"*60)
    print("Start Classification...")
    print("Classification using", predict_fun.__name__)
    if predict_fun == bernoulli:
        bernoulli.tmp_result = {"0":0, "1":0}
        for t in dic.once["term"]:
            tmp = bernoulli.tmp_result
            tmp["0"] += math.log(1-smoothing_bernoulli(dic, t, "0"))
            tmp["1"] += math.log(1-smoothing_bernoulli(dic, t, "1"))

    in_f =  open(query_file, "rt")
    out_f = open(result_file, "wt")

    out_f.write(in_f.readline())

    for i, line in enumerate(in_f):
        #if i%1000 == 0:
        #    print(i)

        id_, sentence, label = training.split_doc(line.strip())
        term_list = nlp.parser(sentence)
        new_label = classify(term_list, dic, predict_fun)

        if line.strip()[-1] in ("0", "1"):
            new_line = line.strip()[:-1] + new_label + "\n"
        else:
            new_line = line.strip() + "\t" + new_label + "\n"

        out_f.write(new_line)
    in_f.close()
    out_f.close()
    print("Finish Classification!")
```

  - ✓ 쿼리 파일을 읽은 뒤 classify 함수를 호출해서 라벨을 결정한다. 결정된 라벨을 기존 데이터에 뒤에 붙여서 결과 파일에 출력한다.

- main 함수

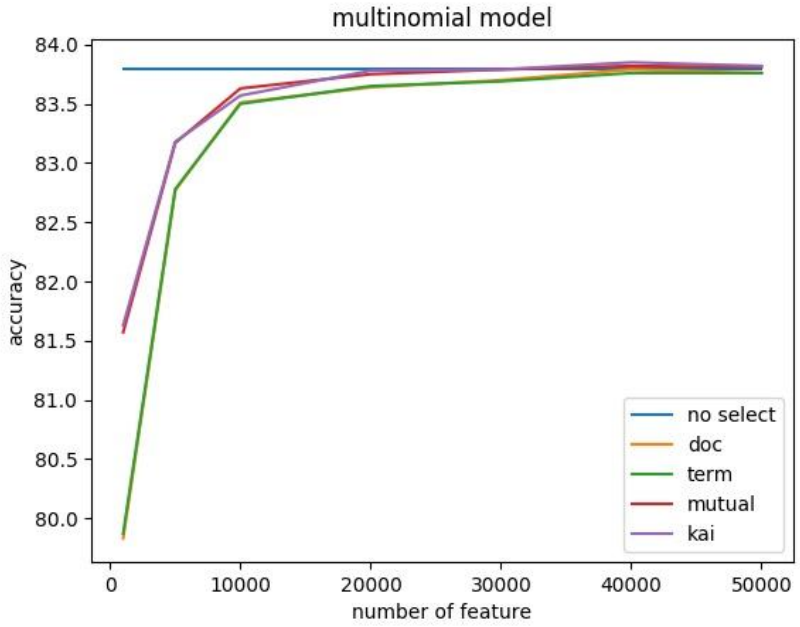
```
def main():
    training_file = "ratings_train.txt"
    query_file = "ratings_test.txt"
    result_file = "ratings_result.txt"

    start_time = timeit.default_timer()

    predict_fun = classifier.multinomial

    dic = training.train(training_file)
    classifier.nbc(query_file, result_file, dic, predict_fun)

    end_time = timeit.default_timer()
    print("Total time: {:.3} sec".format(end_time - start_time))
```

  - ✓ 최종적으로는 Feature Selection 기법을 사용하지 않았기 때문에(다음 결과 항목 참조) train 함수로 트레이닝 해 딕셔너리가 있는 dic 객체를 생성한 뒤 nbs 함수로 분류한다.

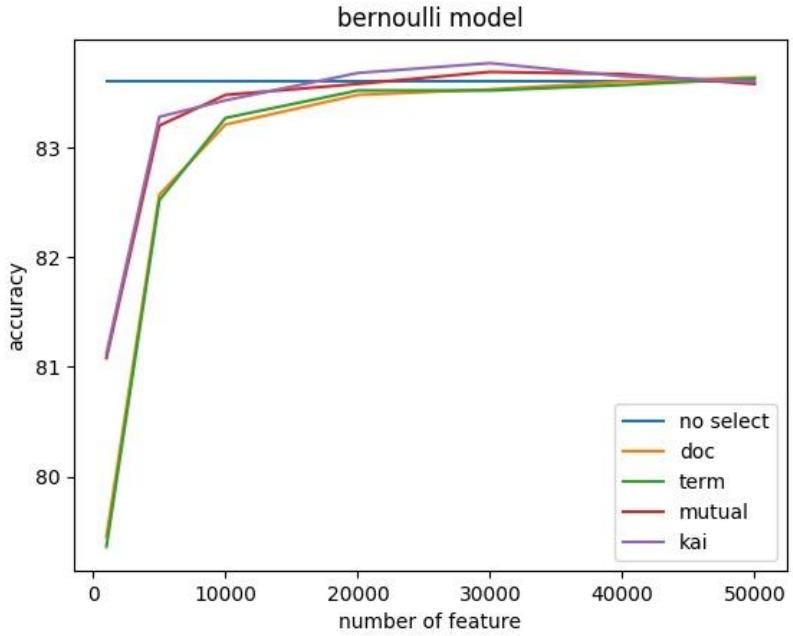
6. 결과

- 각 모델에 대해 Feature Selection 을 적용 한경우와 4 가지 Feature Selection 을 적용했을 때, 그리고 그때마다 Feature 의 수를 다르게 해서 실험했다.

- Multinomial Model



- Bernoulli Model



- 정확도는 'ratings_valid.txt'의 실제 라벨과 프로그램을 통해 분류한 라벨이 얼마나 일치하는지를 기준으로 측정하였다.

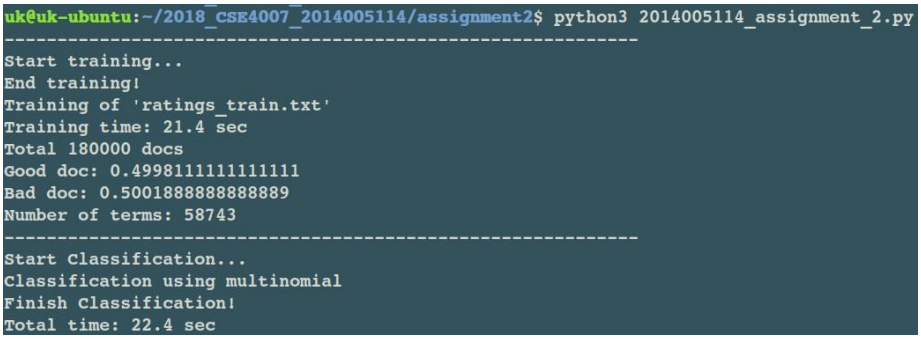
- 실험 결과 Multinomial 모델과 Bernoulli 모델이 비슷한 정확도를 보여주었다. Feature Selection 기법을 비교했을 때는 doc frequency 와 term frequency 방식이 mutual information, kai square 방식보다 성능이 떨어 짐을 보였다. 하지만 그보다 더 중요한 결과는 두가지 모델 모두 feature selection 의 효과가 미미했다는 것이다.

  Multinomial 모델은 kai square 방식으로 40000 개의 feature 를 추출했을 때 83.85%의 정확도를 보여주었다. Feature Selection 을 하지 않았을 때 83.8%의 정확도를 보여주었으므로 0.05% 증가했다. 전체 데이터가 10000 개 이므로 5 개를 더 맞춘 것이다.

  Bernoulli 모델은 feature selection 을 사용하지 않았을 대 83.61% 였는데 가장 좋은 건 kai square 방식으로 30000 개의 feature 를 추출했을 때 83.77% 가 되었다. Multinomial 보다는 Bernoulli 모델에서 feature selection 의 효과가 더 좋았는데, 이는 Bernoulli 모델이 noise feature 에 민감함을 보여준다.

  그럼에도 불구하고 전반적으로 feature selection 의 성능은 좋지 않다. 그 이유는 우리의 영화평 데이터가 대부분 문장 하나로 매우 짧기 때문으로 추측된다. 쿼리 문장이 짧기 때문에 우리가 비교할 수 있는 단어도 적다. 그런 상황에서 딕셔너리의 크기를 감소시키는 것은 비교할 때 사용할 수 있는 작은 확률들까지 제거해버려서 오히려 방해가 될 수 있다. 따라서 제출한 프로그램에는 feature Selection 을 사용하지 않은 Multinomial 모델을 사용했다.

7. 실행



- '2014005114_assignment_2.py' 파이썬 소스코드를 실행하면 'ratings_train.txt' 파일에 있는 데이터를 학습한 뒤 'ratings_test.txt' 파일을 분류해서 ratings_result.txt 파일에 결과를 저장한다. 실행시간 20 초 정도 소요된다.

- 문장을 파싱하기 위한 konlpy 라이브러리를 우분투에선 pip3 를 이용해 쉽게 설치할 수 있다(참조 4). konlpy 라이브러리의 mecab 클래스를 사용하기 위해서는 추가적인 설치가 필요한데 참조 4 번 링크의 3 번째 항목을 참조한다(윈도우 불가). 만약 설치하다가 'Python.h' 파일이 없다는 오류가 뜬다면 python-dev 과 python3-dev 을 설치한다(참조 5).

8. 개선

전반적으로 다양한 기법을 사용해보아도 정확도가 올라가지 않았다. 그 이유는 쿼리 문장이 짧기 때문이라고 생각된다. 참조 6 에 있는 논문을 보면 텍스트의 길이가 짧을 때 좋은 성능을 보여주는 다른 스무딩 방법들을 제시하고 있다. 이것들을 쓰면 성능이 좋아지겠지만, unigram 이 아니라 n-gram 을 사용해야 되서 구현이 어려울 것 같다.

9. 참조

- 1 - Christopher D. Manning, Prabhakar Raghavan and Hinrich Schütze, Introduction to Information Retrieval, Cambridge University Press. 2008.
- 2 - Machine Learning, Tom Mitchell, McGraw Hill, 1997.
- 3 - 불용어 리스트(https://bab2min.tistory.com/544)
- 4 - https://konlpy-ko.readthedocs.io/ko/latest/install/#ubuntu
- 5 - https://stackoverflow.com/questions/21530577/fatal-error-python-h-no-such-file-or-directory
- 6 - http://www.ntu.edu.sg/home/gaocong/papers/wpp095-yuan.pdf