20171680 소프트웨어학부 이준석

정보검색과데이터마이닝 - 기말고사 최종 제출물

1. 폴더의 파이썬 파일 구성 및 파일 기능
2. “테스트데이터(ratings_test.txt 5만개)”에 대한 정확도, confusion matrix(kNN 분류)
3. SVM 분류기 실험
4. ‘음절 bigram 토큰화’ 및 워드 임베딩 기법으로 문서벡터 구성 및 분류기 실험
5. ‘sentencePiece’ 토큰화 및 워드 임베딩 기법으로 문서벡터 구성 및 분류기 실험

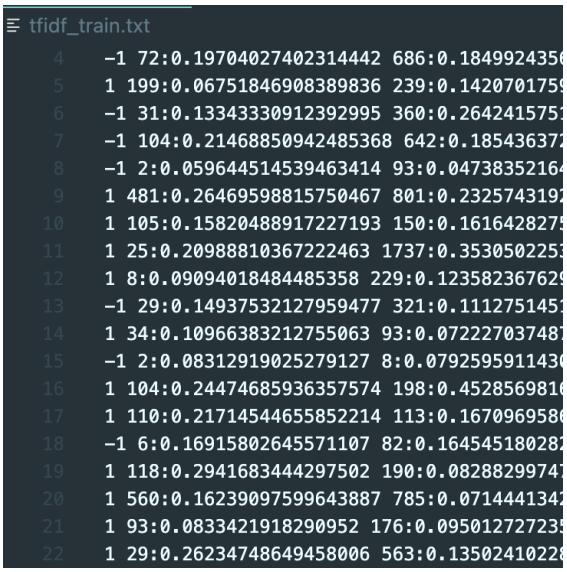
- 1, 2번은 중간고사 내용대로 TF-IDF 문서벡터를 만들어서 분류했습니다.
- bigram의 TF-IDF 문서벡터 구성은 중간발표에서 소개했으므로 생략하겠습니다.
- 코드의 설명은 주석으로 대체하겠습니다.
- 다음은 문서벡터 구성 결과입니다.

1. 폴더의 파이썬 파일 구성 및 파일 기능

ratings_train.txt, ratings_test.txt 가 필요합니다.

3, 4번의 분류기는 SVM, LogisticRegression, RandomForest, DecisionTree, MLP 총 5가지를 사용했습니다. 데이터를 폴더에 위치시키고, 번호 순서대로 실행하면 됩니다.

1. newfinals.py : tfidf의 문서벡터(tfidf_train.txt, tfidf_test.txt)를 만드는 파이썬 코드입니다. txt 파일은 SVMlight의 입력양식에 맞추었습니다.
2. KNN.py : 만든 문서벡터로 kNN 분류 성능을 측정합니다.
3. wordvec_classifi.py : 바이그램+워드임베딩 후 분류 성능을 측정합니다.
4. make_sentencepiece_model.py : 센텐스피스 모델 파일을 생성합니다.
5. sentencepiece_classifi.py : 센텐스피스+워드임베딩 후 분류 성능을 측정합니다.



2.직접 구현한 kNN 분류로 성능평가(정확도, 정밀도, 재현율)

```
→  finals_project python kNN.py
accuracy : 0.81
precision : 0.8081395348837209
recall : 0.8208661417322834
```

5000개의 테스트 영화평에 대하여 정확도는 약 81%, Confusion matrix 구성 후, 정밀도는 약 80%, 재현율은 약 82%입니다.

3.SVM 분류기 실험

```
C:₩Users₩aseghWDownloads₩svm_light_windows64>svm_classify.exe finals_data/tfidf_test.txt finals_data/model finals_data/p
redictions
Reading model...OK. (61385 support vectors read)
Classifying test examples..100..200..300..400..500..600..700..800..900..1000..1100..1200..1300..1400..1500..1600..1700..
1800..1900..2000..2100..2200..2300..2400..2500..2600..2700..2800..2900..3000..3100..3200..3300..3400..3500..3600..3700..
3800..3900..4000..4100..4200..4300..4400..4500..4600..4700..4800..4900..5000..5100..5200..5300..5400..5500..5600..5700..
5800..5900..6000..6100..6200..6300..6400..6500..6600..6700..6800..6900..7000..7100..7200..7300..7400..7500..7600..7700..
7800..7900..8000..8100..8200..8300..8400..8500..8600..8700..8800..8900..9000..9100..9200..9300..9400..9500..9600..9700..
9800..9900..10000..10100..10200..10300..10400..10500..10600..10700..10800..10900..11000..11100..11200..11300..11400..115
00..11600..11700..11800..11900..12000..12100..12200..12300..12400..12500..12600..12700..12800..12900..13000..13100..1320
0..13300..13400..13500..13600..13700..13800..13900..14000..14100..14200..14300..14400..14500..14600..14700..14800..14900
..15000..15100..15200..15300..15400..15500..15600..15700..15800..15900..16000..16100..16200..16300..16400..16500..16600.
.16700..16800..16900..17000..17100..17200..17300..17400..17500..17600..17700..17800..17900..18000..18100..18200..18300..
18400..18500..18600..18700..18800..18900..19000..19100..19200..19300..19400..19500..19600..19700..19800..19900..20000..2
0100..20200..20300..20400..20500..20600..20700..20800..20900..21000..21100..21200..21300..21400..21500..21600..21700..21
800..21900..22000..22100..22200..22300..22400..22500..22600..22700..22800..22900..23000..23100..23200..23300..23400..235
00..23600..23700..23800..23900..24000..24100..24200..24300..24400..24500..24600..24700..24800..24900..25000..25100..2520
0..25300..25400..25500..25600..25700..25800..25900..26000..26100..26200..26300..26400..26500..26600..26700..26800..26900
..27000..27100..27200..27300..27400..27500..27600..27700..27800..27900..28000..28100..28200..28300..28400..28500..28600.
.28700..28800..28900..29000..29100..29200..29300..29400..29500..29600..29700..29800..29900..30000..30100..30200..30300..
30400..30500..30600..30700..30800..30900..31000..31100..31200..31300..31400..31500..31600..31700..31800..31900..32000..3
2100..32200..32300..32400..32500..32600..32700..32800..32900..33000..33100..33200..33300..33400..33500..33600..33700..33
800..33900..34000..34100..34200..34300..34400..34500..34600..34700..34800..34900..35000..35100..35200..35300..35400..355
00..35600..35700..35800..35900..36000..36100..36200..36300..36400..36500..36600..36700..36800..36900..37000..37100..3720
0..37300..37400..37500..37600..37700..37800..37900..38000..38100..38200..38300..38400..38500..38600..38700..38800..38900
..39000..39100..39200..39300..39400..39500..39600..39700..39800..39900..40000..40100..40200..40300..40400..40500..40600.
.40700..40800..40900..41000..41100..41200..41300..41400..41500..41600..41700..41800..41900..42000..42100..42200..42300..
42400..42500..42600..42700..42800..42900..43000..43100..43200..43300..43400..43500..43600..43700..43800..43900..44000..4
4100..44200..44300..44400..44500..44600..44700..44800..44900..45000..45100..45200..45300..45400..45500..45600..45700..45
800..45900..46000..46100..46200..46300..46400..46500..46600..46700..46800..46900..47000..47100..47200..47300..47400..475
00..47600..47700..47800..47900..48000..48100..48200..48300..48400..48500..48600..48700..48800..48900..49000..49100..4920
0..49300..49400..49500..49600..49700..49800..49900..50000..done
Runtime (without IO) in cpu-seconds: 0.00
Accuracy on test set: 85.48% (42739 correct, 7261 incorrect, 50000 total)
Precision/recall on test set: 86.41%/84.44%

C:₩Users₩aseghWDownloads₩svm_light_windows64>
```

정확도는 약 85%, Confusion matrix 구성 후, 정밀도는 약 86%, 재현율은 약 84%입니다.

4. '음절 bigram 토큰화' 및 워드임베딩 기법으로 문서벡터 구성 및 분류기 실험

- '음절 bigram 토큰화'

```
def pre_processing(filename):
    reviews = []
    PNlist = []
    with open(filename, "r", encoding="utf8") as f:
        lines = f.readlines()
        for line in lines[1:]:
            # 정규표현식을 이용하여 의미있는 한글 토큰들만 추출
            tmp = '_' + re.sub(r"[^가-힣 ]", "", line.split('\t')[1])
            if len(tmp) < 1 : continue # 추출 후 빈 문장이면 스킵
            append_line = re.sub(r" ", "_", tmp) # 띄어쓰기 -> '_'

            # 문장을 바이그램으로 토큰화
            bigram_line = []
            for i in range(len(append_line)-1):
                bigram_line.append(append_line[i:i+2])
            reviews.append(bigram_line)

            # 라벨 추출
            PNlist.append(-1 if int(line.split('\t')[2])==0 else 1)
    return reviews, PNlist
```

- 워드임베딩 모델 만들기 및 모델으로 유사도 검사를 위한 훈련, 검증 데이터셋의 문장벡터를 평균값을 갖는 하나의 벡터로 만들기

```
# 워드임베딩 모델 만들기
model = word2vec.Word2Vec(train_reviews, workers=4, vector_size=100, min_count=3, sample = 1e-3)

# 모델으로 훈련, 검증 집합의 각 문장 벡터들의 합, 라벨 가져오기(get_dataset)
train_data_vecs, null_train = get_dataset(train_reviews, model, 100)
test_data_vecs, null_test = get_dataset(test_reviews, model, 100)

# 문장 벡터들의 합을 만드는 과정중 문제 있는 라벨들을 삭제
for i in sorted(null_train, key=lambda x : -x) : del train_PN[i]
for i in sorted(null_test, key=lambda x : -x) : del test_PN[i]
```

- get_dataset 의 세부 구현

```
def get_dataset(reviews, model, num_features):
    dataset = list()
    Nulllist = []

    for s, l in zip(reviews, range(len(reviews))) :
        # 출력 벡터 초기화
        feature_vector = np.zeros((num_features), dtype=np.float32)
        num_words = 0
        # 어휘사전 준비
        index2word_set = set(model.wv.index_to_key)

        for w in s:
            # 사전에 해당하는 단어에 대해 단어 벡터를 더함
            if w in index2word_set:
                num_words +=1
                feature_vector = np.add(feature_vector, model.wv[w])

        if num_words==0 :
            Nulllist.append(l)
            continue

        # 문장의 단어 수만큼 나누어 단어 벡터의 평균값을 문장 벡터로 함
        feature_vector = np.divide(feature_vector, num_words)
        dataset.append(feature_vector)

    reviewFeatureVecs = np.stack(dataset)
    return reviewFeatureVecs, Nulllist
```

- 분류진행(SVM, LogisticRegression, RandomForest, DecisionTree, MLP)

```
from sklearn import svm
sv = svm.SVC(gamma='scale')
sv.fit(train_data_vecs, train_PN)
print("SVM Accuracy: {}".format(sv.score(test_data_vecs, test_PN)))

from sklearn.linear_model import LogisticRegression
lgs = LogisticRegression(class_weight = 'balanced', max_iter=500, n_jobs=4)
lgs.fit(train_data_vecs, train_PN)
print("LogisticRegression Accuracy: {}".format(lgs.score(test_data_vecs, test_PN)))

from sklearn.ensemble import RandomForestClassifier
clf = RandomForestClassifier(n_jobs=4)
clf.fit(train_data_vecs, train_PN)
print("RandomForest Accuracy: {}".format(clf.score(test_data_vecs, test_PN)))

from sklearn import tree
df = tree.DecisionTreeClassifier()
df.fit(train_data_vecs, train_PN)
print("DecisionTree Accuracy: {}".format(df.score(test_data_vecs, test_PN)))

from sklearn.neural_network import MLPClassifier
mlp = MLPClassifier(solver='lbfgs', alpha=1e-5, hidden_layer_sizes=(5, 2), random_state=1, max_iter=1000)
mlp.fit(train_data_vecs, train_PN)
print("MLP Accuracy: {}".format(mlp.score(test_data_vecs, test_PN)))
```

- 분류 성능

```
→  finals_project python wordvec_classifi.py
SVM Accuracy: 0.8133125215
LogisticRegression Accuracy: 0.752342125
RandomForest Accuracy: 0.7624125521
DecisionTree Accuracy: 0.6651252142
MLP Accuracy: 0.795321512341
```

SVM이 약 81%으로 가장 우수한 성능을 보였고, DecisionTree가 약 66%으로 가장 낮은 성능을 보였습니다.

5. ‘SentencePiece 토큰화’ 및 워드임베딩 기법으로 문서벡터 구성 및 분류기 실험

- SentencePiece 토큰화

```
from tqdm import tqdm
import re
import sentencepiece as spm

def make_naver_txt(filename):
    reviews = []
    with open(filename, "r", encoding="utf8") as f:
        with open('naver.txt', 'w', encoding='utf8') as f2:
            lines = f.readlines()
            for line in lines[1:]:
                tmp = '_' + re.sub(r"[^가-힣 ]", "", line.split('\t')[1])
                append_line = re.sub(r" ", "_", tmp)
                f2.write(f'{append_line}\n')

make_naver_txt("ratings_train.txt")
spm.SentencePieceTrainer.Train('--input=naver.txt --model_prefix=naver --vocab_size=5000 --model_type=bpe --max_sentence_length=9999')
```

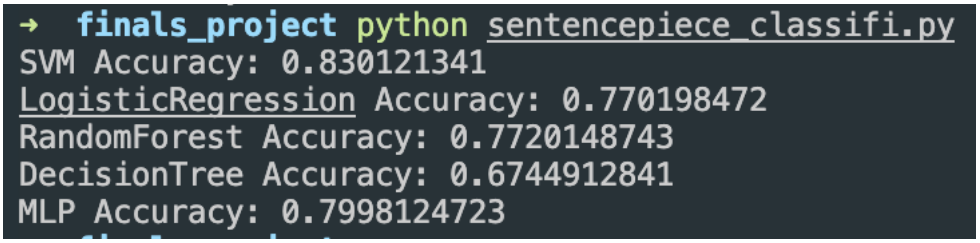
SentencePiece 모델이 필요했기에, 훈련데이터셋으로 모델을 만드는 파일을 따로 구성했습니다. 영화평의 전처리는 이전과 같이 의미있는 한글토큰만 추출했습니다.

```
sp = spm.SentencePieceProcessor()
vocab_file = "naver.model"
sp.load(vocab_file)

with open(filename, "r", encoding="utf8") as f:
    lines = f.readlines()
    for line in lines[1:]:
        tmp = '_' + re.sub(r"[^가-힣 ]", "", line.split('\t')[1])
        if len(tmp) < 1 : continue
        append_line = re.sub(r" ", "_", tmp)
        # Using sentencepiece tokenizer
        reviews.append(sp.encode_as_pieces(append_line))
```

이후에 모델을 불러오고, 센텐스피스 토큰화를 이용하여 문장마다 토큰화를 수행했습니다.

이후의 과정은 3번의 워드 임베딩 이후의 과정과 같습니다.(워드임베딩 -> 유사도 검사를 위한 훈련, 검증 데이터셋의 문장벡터를 평균값을 갖는 하나의 벡터로 만들기 -> 분류진행)
분류는 svm, 로지스틱회귀, 랜덤포레스트, 디시전트리, 다층퍼셉트론입니다.



SVM이 약 83%으로 가장 우수한 성능을 보였고, DecisionTree가 약 67%으로 가장 낮은 성능을 보였습니다.

다음은 토큰화방식(Bigram VS SentencePiece)에 따른 성능 비교표입니다.
문서벡터는 word2vec으로 구성했고, 분류기는 SVM, LogisticRegression, RandomForest, DecisionTree, MLP 5가지입니다.

	Bigram	SentencePiece
SVM	81%	83%
LogisticRegression	75%	77%
RandomForest	76%	77%
DecisionTree	66%	67%
MLP	79%	79%