SNS 이미지를 활용한 다중분류

통계학과 2020020344 정해성

1. 개요

최근 기업들은 고객이 올린 사진을 카테고리 별로 자동으로 분류하여 스토리를 만들어 제공하거나, 고객이 최근에 조회한 사진을 바탕으로 광고를 송출하고 있다. 이처럼 실제 사진을 카테고리 별로 분류할 수 있다면 이를 바탕으로 마케팅과 고객 서비스를 비롯한 많은 분야에 적용할 수 있다. 이러한 이유로 응용통계학특수연구1를 수강하며 배운 딥러닝 아키텍처를 현실 데이터에 적용해보고자 SNS 게시물 이미지를 활용한 다중분류를 주제로 선정하였다. 데이터를 수집한 플랫폼은 2019년 한국 미디어 패널조사 결과(정보통신정책연구원), SNS 종류별 이용률 변화를 참고하여 3위를 기록한 인스타그램으로 선택하였으며, 수집한 카테고리는 운동(#벌크업, #운스타그램), 책(#북스타그램, #독서스타그램), 반려동물(#펫스타그램, #반려동물), 맛집(#맛스타그램, #음식스타그램), 그리고 육아(#육아, #아기스타그램)로 총 5개다. 이와 같은 카테고리를 선정한 이유는 사진의 대상이 크게 차이가 나기 때문에 다중분류를 시도할 시 모형 학습이 잘 되리라 생각했기 때문이다. 또한, 각 카테고리 당 2개의 해시태그를 선정하였는데 선정한 해시태그마다 약 500개가량의 훈련이미지를 크롤링하여 최종적으로 수집한 총 훈련 이미지는 4993개다. 그리고 모형의 성능을 평가하기 위하여 각 카테고리마다 약 200개가량의 테스트 이미지를 수집하여 총 995장의 테스트 데이터도 별도로 준비하였다. 이처럼 수집한 훈련용 컬러 이미지 4993장을 훈련 데이터와 검증 데이터로 나누어 MovileNetV2를 활용한 전이학습 아키텍처로 다항분류 모형을 만들었다. 그 후 995장의 테스트 데이터로 test score를 산출하여 모형을 평가하였다.

2. 자료 수집

이미지의 대표성을 확보하기 위해 인스타그램 검색창에서 각 카테고리에 해당하는 해시태그를 검색한 후 '좋아요' 개수가 30개 이상인 게시물들의 사진을 크롤링하였다. 이 과정에서 본 분석에서는 사용하지 않았지만, 링크(post_link), 좋아요 수(Likes), 해시태그(hashtag), 그리고 이미지이름(image_local_name)을 크롤링하였다. 크롤링의 코드는 AlecMorgan(https://github.com/AlecMorgan)의 코드를 참조하였다. (자세한 크롤링 코드와 전체 코드는 메일에 첨부하였습니다.)

<수집 데이터>

범주	검색한 해시태그	수집한 이미지 개수(train, test)	
육아	#육아, #아기스타그램	1006장	200장
운동	#벌크업, #운스타그램	1000장	200장
반려동물	#반려동물, #펫스타그램	987장	195장
책	#독서스타그램, #북스타그램	1000장	200장
음식	#음식스타그램, #맛스타그램	1000장	200장

3. 전처리

데이터의 개수가 적어 MobileNetV2를 이용한 이전학습을 진행할 예정이므로 데이터의 형식이 이에 맞도록 전처리하였다. 우선 아래와 같이 전체 데이터를 검색한 해시태그(search_hashtag)와 각 폴더 내의 이미지 파일의 이름(img_name)을 반복문을 통하여 불러왔다.

In [2]:
```
# 검색한 해시태그(폴더) 불러오기
import os
search_hashtag = os.listdir('/content/drive/My Drive/응특연 기말레포트/data/')
search_hashtag
```

Out[2]:
```
['육아',
 '벌크업',
 '반려동물',
 '독서스타그램',
 '펫스타그램',
 '음식스타그램',
 '아기스타그램',
 '북스타그램',
 '운스타그램',
 '맛스타그램']
```

그 후 검색한 해시태그와 각 해시태그 폴더 내 이미지 파일의 이름(img_name)을 합쳐 df1 - df5라는 데이터 프레임을 생성하였다.

In [3]:
```
# 각 카테고리별 해시태그 묶기
hash1 = ['육아','아기스타그램']          # 육아
hash2 = ['벌크업','운스타그램']          # 운동
hash3 = ['반려동물','펫스타그램']        # 반려동물
hash4 = ['독서스타그램','북스타그램']    # 책
hash5 = ['음식스타그램','맛스타그램']    # 음식
```

In [4]:
```
# 카테고리별로 해시태그(search)와 이미지 이름(img_name)을 갖는 데이터 프레임(df1 ~ df5) 생성
import pandas as pd

# 육아(#육아, # 아거스타그램)
img_name=[]
search = []
for i,hashtag in enumerate(hash1):
    img_name.append(os.listdir(f'/content/drive/My Drive/응특연 기말레포트/data/{hashtag}'))
    for j in range(len(img_name[i])):
        search.append(hashtag)

img_name = [y for x in img_name for y in x]
df1 = pd.DataFrame()
df1['search'] = search
df1['img_name'] = img_name

# 운동(#벌크업, #운스타그램)
img_name=[]
search = []
for i,hashtag in enumerate(hash2):
    img_name.append(os.listdir(f'/content/drive/My Drive/응특연 기말레포트/data/{hashtag}'))
    for j in range(len(img_name[i])):
        search.append(hashtag)

img_name = [y for x in img_name for y in x]
df2 = pd.DataFrame()
df2['search'] = search
df2['img_name'] = img_name

# 반려동물(#반려동물, #펫스타그램)
img_name=[]
search = []
for i,hashtag in enumerate(hash3):
    img_name.append(os.listdir(f'/content/drive/My Drive/응특연 기말레포트/data/{hashtag}'))
    for j in range(len(img_name[i])):
        search.append(hashtag)

img_name = [y for x in img_name for y in x]
df3 = pd.DataFrame()
df3['search'] = search
df3['img_name'] = img_name

# 책(#독서스타그램, #북스타그램)
img_name=[]
search = []
for i,hashtag in enumerate(hash4):
    img_name.append(os.listdir(f'/content/drive/My Drive/응특연 기말레포트/data/{hashtag}'))
    for j in range(len(img_name[i])):
        search.append(hashtag)

img_name = [y for x in img_name for y in x]
df4 = pd.DataFrame()
df4['search'] = search
df4['img_name'] = img_name

# 음식(#음식스타그램, #맛스타그램)
img_name=[]
search = []
for i,hashtag in enumerate(hash5):
    img_name.append(os.listdir(f'/content/drive/My Drive/응특연 기말레포트/data/{hashtag}'))
    for j in range(len(img_name[i])):
        search.append(hashtag)

img_name = [y for x in img_name for y in x]
df5 = pd.DataFrame()
df5['search'] = search
df5['img_name'] = img_name
```

(1) x data

이렇게 생성한 데이터 프레임의 해시태그와 이미지 이름을 바탕으로 cv2 라이브러리를 활용하여 이미지를 불러온 후, (160,160,3)의 크기를 가지도록 각 이미지의 사이즈를 조정하였다. 불러온 이미지 데이터는 x라고 칭하였으며 총 4993개의 이미지를 불러왔다. 이 x 데이터를 모형에 사용하기 위해 넘파이 배열로 변환 후 각 픽셀이 0에서 1의 값을 갖도록 표준화하였다.

In [5]:
```
# 이미지 불러오기
import cv2
x1=[]
```

```
# 육아
count=0
for i in range(len(df1)):
  name = df1['img_name'][i]
  search = df1['search'][i]
  path = f'/content/drive/My Drive/응특연 기말레포트/data/{search}/{name}'
  img = cv2.imread(path)
  img = cv2.cvtColor(img,cv2.COLOR_BGR2RGB)
  img = cv2.resize(img,(160,160))    # 픽셀을 160x160로 사이즈 조정
  x1.append(img)
  if (i%100==0):
    print(i)

# 운동
count=0
for i in range(len(df2)):
  name = df2['img_name'][i]
  search = df2['search'][i]
  path = f'/content/drive/My Drive/응특연 기말레포트/data/{search}/{name}'
  img = cv2.imread(path)
  img = cv2.cvtColor(img,cv2.COLOR_BGR2RGB)
  img = cv2.resize(img,(160,160))    # 픽셀을 160x160로 사이즈 조정
  x1.append(img)
  if (i%100==0):
    print(i)

# 반려동물
count=0
for i in range(len(df3)):
  name = df3['img_name'][i]
  search = df3['search'][i]
  path = f'/content/drive/My Drive/응특연 기말레포트/data/{search}/{name}'
  img = cv2.imread(path)
  img = cv2.cvtColor(img,cv2.COLOR_BGR2RGB)
  img = cv2.resize(img,(160,160))    # 픽셀을 160x160로 사이즈 조정
  x1.append(img)
  if (i%100==0):
    print(i)
```

```
# 책
count=0
for i in range(len(df4)):
  name = df4['img_name'][i]
  search = df4['search'][i]
  path = f'/content/drive/My Drive/응특연 기말레포트/data/{search}/{name}'
  img = cv2.imread(path)
  img = cv2.cvtColor(img,cv2.COLOR_BGR2RGB)
  img = cv2.resize(img,(160,160))    # 픽셀을 160x160로 사이즈 조정
  x1.append(img)
  if (i%100==0):
    print(i)

# 음식
count=0
for i in range(len(df5)):
  name = df5['img_name'][i]
  search = df5['search'][i]
  path = f'/content/drive/My Drive/응특연 기말레포트/data/{search}/{name}'
  img = cv2.imread(path)
  img = cv2.cvtColor(img,cv2.COLOR_BGR2RGB)
  img = cv2.resize(img,(160,160))    # 픽셀을 160x160로 사이즈 조정
  x1.append(img)
  if (i%100==0):
    print(i)
```

In [6]:
```
print(len(x1))
```

```
4993
```

In [7]:
```
# x데이터를 넘파이 배열로 변환
import numpy as np
x = np.array(x1)

# 0에서 1의 값을 갖도록 x데이터 표준화
x = x.astype('float32')/255.
```

(2) y data

각 소분류에 맞는 대분류(맛집, 육아, 반려동물, 책, 운동)를 범주로 정리한 df의 행의 개수만큼 적용하여 이번 모형에서 목적변수가 될 y변수를 생성하였다. 그 후 이들에게 0~4까지의 라벨을 부여하여 최종적으로 분석에 사용될 y데이터를 만들었으며 후에 예측 결과에 사용하기 위해 y_original 변수를 생성했다.

```
In [8]:
# 각 이미지에 맞는 범주, target 생성
y = []
for i in range(df1.shape[0]):
  y.append('육아')
for i in range(df2.shape[0]):
  y.append('운동')
for i in range(df3.shape[0]):
  y.append('반려동물')
for i in range(df4.shape[0]):
  y.append('책')
for i in range(df5.shape[0]):
  y.append('음식')
```

```
In [9]:
# 최종 데이터 프레임
df = pd.concat([df1,df2,df3,df4,df5]).reset_index(drop=True)
df['category'] = y
df
```

Out [9]:

	search	img_name	category
0	육아	00191554-eb3a-49d4-af8d-2d7328769148.jpg	육아
1	육아	017ef00c-858d-4133-800e-8016d9a7587a.jpg	육아
2	육아	018ee93a-bc69-4d29-a512-9bf477ada336.jpg	육아
3	육아	03c7cb1d-6f04-461b-812d-c5bc02bdf797.jpg	육아
4	육아	02fbb2b4-61bd-4f62-811b-dbc2e228b329.jpg	육아
...	...	...	...
4988	맛스타그램	fddd8b5f-6fa4-4d58-8161-6c77b1c77d19.jpg	음식
4989	맛스타그램	ffeb0b7d-9c25-46f6-b7b8-09a3969a2a11.jpg	음식
4990	맛스타그램	ff39277f-4b2e-4d9c-99d5-ddc41c39e956.jpg	음식
4991	맛스타그램	ff1ab413-6cbc-47b1-9253-0e73ec2f06fd.jpg	음식
4992	맛스타그램	ff4013d6-2262-4d34-915c-e8bf781151ac.jpg	음식

4993 rows × 3 columns

```
In [10]:
# 범주형에 라벨 부여
df['y'] = df['category'].replace(['육아','운동','반려동물','책','음식'],[0,1,2,3,4])
y_original = ['육아','운동','반려동물','책','음식']
df.head()
```

```
In [12]:
# target y
y = df['y']
```

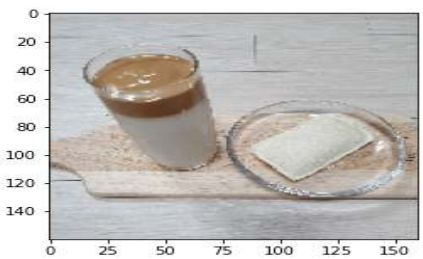
(3) 최종 데이터

최종적으로 완성된 data x는 (4993,160,160,3)인 4D 텐서로, 총 4993개의 160x160의 픽셀을 갖는 컬러 이미지이며, target y는 0,1,2,3,4(각각 육아, 운동, 반려동물, 책, 음식)로 총 5개의 범주를 갖는 (4993,)인 1D 텐서이다. 앞으로 분석할 x 이미지와 타겟 y가 잘 처리되었는지 확인하기 위해 그래프를 그려보았다.

타겟 y가 5개의 범주를 가지므로 다항분류의 문제이며 이를 위해 모델링 과정에서 최종 출력층의 활성함수는 Softmax, 그리고 손실함수는 Categorical Cross Entropy를 사용하기로 하였다.

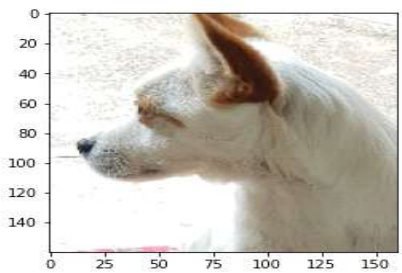
```
In [32]:
import matplotlib.pyplot as plt
plt.imshow(x[3997])
print(y[3997])
print(y_original[y[3997]])
```

```
4
음식
```




```
In [64]:
plt.imshow(x[2500])
print(y[2500])
print(y_original[y[2500]])
```

```
2
반려동물
```



4. 모델링

(1) 훈련 데이터와 테스트 데이터 분리 및 원-핫 벡터처리

모형 적합에 앞서 불러온 데이터를 80%의 train data와 20%의 test data로 분리시켜주었다(test data라고 칭하였지만 모형적합의 과정에서 검증데이터로 사용하였다). 또한 모형의 재현성을 부여하기 위해 random_state=15를 주어 다시 실행하더라도 동일한 훈련 데이터와 시험 데이터가 생성되도록 설정하였다. x_train과 y_train은 각각 (3994,160,160,3), (3994,)의 크기를 가지며, x_test와 y_test는 (999.160,160,3), (999,)의 크기를 가진다. 각 카테고리 별로 약 1000개가량의 데이터가 균일하게 존재하므로 카테고리마다 데이터 개수의 비율을 유지하고 모형을 훈련시키는 것이 중요하다고 판단하였다. 그래서 데이터를 분리할 때 각 카테고리 별 비율이 유지될 수 있도록 stratify=y 옵션을 넣어주었다.

In [33]:
```
from sklearn.model_selection import train_test_split
x_train, x_test, y_train, y_test=train_test_split(x, y, test_size=0.2,stratify=y, random_state=15)
print(x_train.shape)
print(x_test.shape)
print(y_train.shape)
print(y_test.shape)
```

```
(3994, 160, 160, 3)
(999, 160, 160, 3)
(3994,)
(999,)
```

여기서, 다항분류 문제를 다루고 있으므로 타겟 y를 원핫코딩을 해주어야할 필요가 있다. 그래서 y_train과 y_test를 tensorflow.utils의 to_categorical 함수를 이용해 각 카테고리 순서에 1을 부여하고 나머지에는 0을 부여하는 원-핫 벡터로 만들었다.

In [34]:
```
from tensorflow.keras.utils import to_categorical
y_train = to_categorical(y_train)
y_test = to_categorical(y_test)
print(y_train[:5])
print(y_test[:5])
```

```
[[0. 0. 0. 1. 0.]
 [0. 0. 1. 0. 0.]
 [0. 0. 1. 0. 0.]
 [0. 1. 0. 0. 0.]
 [0. 0. 0. 0. 1.]]
[[0. 0. 1. 0. 0.]
 [0. 0. 1. 0. 0.]
 [0. 0. 0. 1. 0.]
 [1. 0. 0. 0. 0.]
 [0. 0. 1. 0. 0.]]
```

(2) 모형 아키텍처

이전학습을 위하여 사전 학습된 모형인 MobileNetV2를 불러와 base_model에 객체화시켰다. 특별히 MobileNetV2를 사용한 이유는, VGG16, InceptionV3 역시 모형 훈련에 사용해보았지만 MobileNetV2가 가장 성능이 좋았기 때문이다. MobileNetV2 모형 뒤에 은닉층을 더 추가할 것이므로 include_top=False를 주었으며, weights='imagenet'을 사용했다.

In [35]:
```
from keras.applications import MobileNetV2, VGG16, ResNet50
from keras.layers import Input, Conv2D,GlobalAveragePooling2D, Dropout, Dense
from keras import Model

base_model=MobileNetV2(include_top=False, weights='imagenet',input_shape=(160,160,3))
```

또한 분석하고자 하는 x와 y 데이터에 맞도록 모수를 추가하고 훈련시키기 위하여 MobileNetV2의 뒤에 6개의 은닉층을 추가하였다. 먼저 GlobalAveragePooling2D 층을 추가하여 사전학습 모형을 통과한 후 크기가 (5,5,1280)인 데이터의 채널별로 픽셀의 평균을 내어 (None,1280)의 크기로 만들어주었다, 그 후 통계학적 관점에서 바로 출력층에 연결한다면 1280개의 공변량이 있는 것이므로 너무 많은 변수가 있다고 생각하였다. 이러한 이유로 변수선택의 효과가 나타나길 기대하며 출력층 직전 층에 32개의 노드를 갖는 Dense층을 추가하였다. 뿐만 아니라 데이터가 적기 때문에 과대적합을 방지하기 위하여 2개의 Dropout(0.5)층을 추가하였다. 마지막으로 최종 분류할 카테고리가 5개이므로 노드 수가 5개이며

softmax를 활성함수로 사용하는 Dense층을 추가하였다.

```
In [36]: m=base_model.output
         m=GlobalAveragePooling2D()(m)
         m=Dense(128,activation='relu')(m) # dense layer 1
         # m=Dense(64,activation='relu')(m) # dense layer 2
         m=Dropout(0.5)(m)
         m=Dense(32,activation='relu')(m) # dense layer 3
         m=Dropout(0.5)(m)
         preds=Dense(5,activation='softmax')(m) #final layer with softmax
         model=Model(inputs=base_model.input,outputs=preds)
```

그리고 다음과 같이 추가한 6개의 은닉층을 제외한 MobileNetV2 은닉층들의 모수는 학습되지 않아야하므로 layer.trainable=False를 주었다. (model.summary()의 output은 너무 길어 html 파일로 첨부하였습니다.)

```
In [37]: for layer in model.layers[:-6]:
             layer.trainable = False

         model.summary()
```

모형의 최적화 알고리즘은 adam을 사용하였고, 손실함수는 다중분류 문제이므로 categorical_crossentropy를 사용했다. 또한 accuracy를 이용해 모형의 성능을 추적하였다. 모형을 여러 번 돌려보며 정확도와 손실을 기준으로 대략적인 흐름을 파악한 결과, 배치 사이즈를 크게 하면 정확도와 손실이 훈련 데이터와 검증 데이터 모두 머무는 경향이 보여 local minimum에 빠지는 것을 방지하고자 배치 사이즈를 32로 정하였다. 첫 번째 모형으로는 총 50번의 에폭을 돌려보며 훈련데이터와 검증데이터의 accuracy와 loss의 변화를 살펴보았으며, 두 번째로는 keras의 callback 라이브러리를 사용하여 최적화를 점검하였다. 그 결과 9번째 에폭에서 훈련이 중단되었다. 그래프 상으로는 8 - 10번째 에폭 사이에서 train loss와 validation loss가 교차하며 accuracy 역시 마찬가지이다.

구체적으로 accuracy를 살펴보면, 에폭이 증가함에 따라 train accuracy는 당연히 계속 증가하며, validation accuracy는 0.7과 0.75 사이에서 머무르는 것을 볼 수 있다. loss에 대한 그래프를 살펴보면 오히려 에폭이 증가할수록 (즉, 에폭이 12정도 이상으로 늘어나면) train loss는 줄어들지만 validation loss는 증가하여 과대적합이 일어나는 현상을 보이고 있다.

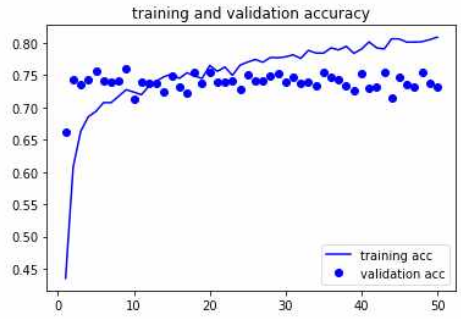
<1. 첫 번째 모형> - 케라스의 최적화 모형을 사용하지 않고 훈련 흐름을 파악(15에폭까지만 표시)

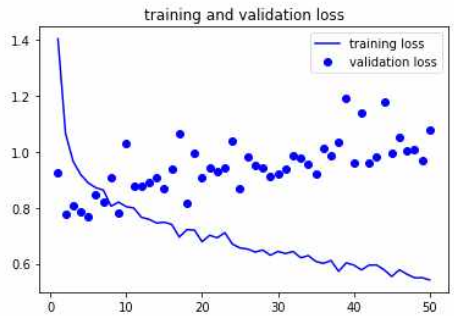
```
In [38]: model.compile(optimizer='adam',
                       loss='categorical_crossentropy', metrics = ['accuracy'])
```

```
In [39]: history = model.fit(x_train,y_train,epochs=50,batch_size=16,validation_data=(x_test,y_test))
```

```
Train on 3994 samples, validate on 999 samples
Epoch 1/50
3994/3994 [==============================] - 18s 5ms/step - loss: 1.4033 - accuracy: 0.4352 - val_loss: 0.9235 - val_accuracy: 0.6627
Epoch 2/50
3994/3994 [==============================] - 16s 4ms/step - loss: 1.0650 - accuracy: 0.6079 - val_loss: 0.7765 - val_accuracy: 0.7437
Epoch 3/50
3994/3994 [==============================] - 15s 4ms/step - loss: 0.9656 - accuracy: 0.6632 - val_loss: 0.8065 - val_accuracy: 0.7357
Epoch 4/50
3994/3994 [==============================] - 16s 4ms/step - loss: 0.9185 - accuracy: 0.6858 - val_loss: 0.7838 - val_accuracy: 0.7437
Epoch 5/50
3994/3994 [==============================] - 16s 4ms/step - loss: 0.8883 - accuracy: 0.6940 - val_loss: 0.7689 - val_accuracy: 0.7568
Epoch 6/50
3994/3994 [==============================] - 16s 4ms/step - loss: 0.8714 - accuracy: 0.7076 - val_loss: 0.8461 - val_accuracy: 0.7417
Epoch 7/50
3994/3994 [==============================] - 16s 4ms/step - loss: 0.8626 - accuracy: 0.7076 - val_loss: 0.8197 - val_accuracy: 0.7387
Epoch 8/50
3994/3994 [==============================] - 16s 4ms/step - loss: 0.8056 - accuracy: 0.7176 - val_loss: 0.9089 - val_accuracy: 0.7407
Epoch 9/50
3994/3994 [==============================] - 16s 4ms/step - loss: 0.8202 - accuracy: 0.7276 - val_loss: 0.7802 - val_accuracy: 0.7598
Epoch 10/50
3994/3994 [==============================] - 16s 4ms/step - loss: 0.8038 - accuracy: 0.7238 - val_loss: 1.0281 - val_accuracy: 0.7127
Epoch 11/50
3994/3994 [==============================] - 15s 4ms/step - loss: 0.7990 - accuracy: 0.7198 - val_loss: 0.8787 - val_accuracy: 0.7397
Epoch 12/50
3994/3994 [==============================] - 16s 4ms/step - loss: 0.7661 - accuracy: 0.7334 - val_loss: 0.8767 - val_accuracy: 0.7367
Epoch 13/50
3994/3994 [==============================] - 16s 4ms/step - loss: 0.7587 - accuracy: 0.7411 - val_loss: 0.8901 - val_accuracy: 0.7367
Epoch 14/50
3994/3994 [==============================] - 16s 4ms/step - loss: 0.7454 - accuracy: 0.7479 - val_loss: 0.9086 - val_accuracy: 0.7247
Epoch 15/50
3994/3994 [==============================] - 16s 4ms/step - loss: 0.7482 - accuracy: 0.7519 - val_loss: 0.8664 - val_accuracy: 0.7497
```

In [40]:
```
import matplotlib.pyplot as plt
acc=history.history['accuracy']
val_acc=history.history['val_accuracy']
loss=history.history['loss']
val_loss=history.history['val_loss']
epochs=range(1,len(acc)+1)
plt.plot(epochs,acc,'b',label='training acc')
plt.plot(epochs,val_acc,'bo',label='validation acc')
plt.title('training and validation accuracy')
plt.legend()
plt.figure()
plt.plot(epochs,loss,'b',label='training loss')
plt.plot(epochs,val_loss,'bo',label='validation loss')
plt.title('training and validation loss')
plt.legend()
plt.show()
```





<2. 두 번째 모형> - 케라스의 최적화 알고리즘 사용

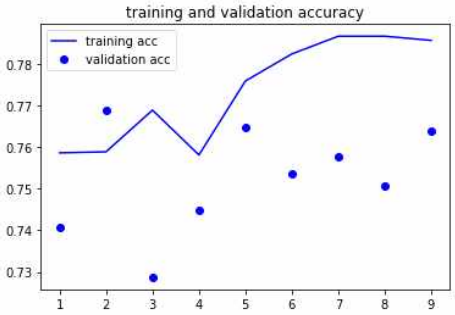
In [22]:
```
m=base_model.output
m=GlobalAveragePooling2D()(m)
m=Dense(128,activation='relu')(m) #dense layer 1
# m=Dense(64,activation='relu')(m) #dense layer 1
m=Dropout(0.5)(m)
m=Dense(32,activation='relu')(m) #dense layer 1
m=Dropout(0.5)(m)
preds=Dense(5,activation='softmax')(m) #final layer with softmax
model2=Model(inputs=base_model.input,outputs=preds)
```

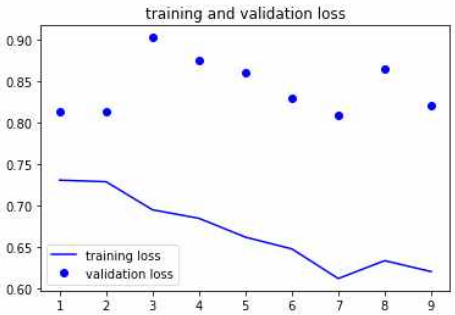
In [42]:
```
from keras.callbacks import EarlyStopping, ModelCheckpoint, ReduceLROnPlateau
callback_list = [EarlyStopping(monitor='acc', patience=2),
ModelCheckpoint(filepath='/content/drive/My Drive/응특연 기말레포트/model_check.h5', monitor='val_loss',save_best_only=True)]

model2.compile(optimizer='adam',
                  loss='categorical_crossentropy', metrics = ['acc'])
```

In [43]:
```
history = model2.fit(x_train,y_train,epochs=50,batch_size=32,callbacks=callback_list,validation_data=(x_test,y_test))
```

```
Train on 3994 samples, validate on 999 samples
Epoch 1/50
3994/3994 [==============================] - 16s 4ms/step - loss: 0.7305 - acc: 0.7586 - val_loss: 0.8131 - val_acc: 0.7407
Epoch 2/50
3994/3994 [==============================] - 13s 3ms/step - loss: 0.7286 - acc: 0.7589 - val_loss: 0.8134 - val_acc: 0.7688
Epoch 3/50
3994/3994 [==============================] - 13s 3ms/step - loss: 0.6947 - acc: 0.7689 - val_loss: 0.9028 - val_acc: 0.7287
Epoch 4/50
3994/3994 [==============================] - 13s 3ms/step - loss: 0.6844 - acc: 0.7581 - val_loss: 0.8755 - val_acc: 0.7447
Epoch 5/50
3994/3994 [==============================] - 13s 3ms/step - loss: 0.6617 - acc: 0.7759 - val_loss: 0.8602 - val_acc: 0.7648
Epoch 6/50
3994/3994 [==============================] - 13s 3ms/step - loss: 0.6475 - acc: 0.7824 - val_loss: 0.8302 - val_acc: 0.7538
Epoch 7/50
3994/3994 [==============================] - 13s 3ms/step - loss: 0.6117 - acc: 0.7867 - val_loss: 0.8082 - val_acc: 0.7578
Epoch 8/50
3994/3994 [==============================] - 13s 3ms/step - loss: 0.6333 - acc: 0.7867 - val_loss: 0.8645 - val_acc: 0.7508
Epoch 9/50
3994/3994 [==============================] - 13s 3ms/step - loss: 0.6200 - acc: 0.7857 - val_loss: 0.8207 - val_acc: 0.7638
```

In [44]:
```
import matplotlib.pyplot as plt
acc=history.history['acc']
val_acc=history.history['val_acc']
loss=history.history['loss']
val_loss=history.history['val_loss']
epochs=range(1,len(acc)+1)
plt.plot(epochs,acc,'b',label='training acc')
plt.plot(epochs,val_acc,'bo',label='validation acc')
plt.title('training and validation accuracy')
plt.legend()
plt.figure()
plt.plot(epochs,loss,'b',label='training loss')
plt.plot(epochs,val_loss,'bo',label='validation loss')
plt.title('training and validation loss')
plt.legend()
plt.show()
```





(3) 최종 모형

위의 두 모형의 결과를 토대로 최종모형을 전체 데이터 x와 y에 대하여 훈련시켰다. 에폭은 9로 하였으며 생성한 최종 모형을 model_final.h5로 저장하였다.

<3. 최종모형>

In [2]:
```
import numpy as np
x = np.load('/content/drive/My Drive/응특연 기말레포트/x.npy')
y = np.load('/content/drive/My Drive/응특연 기말레포트/y.npy')
x = x/255.
from tensorflow.keras.utils import to_categorical
y = to_categorical(y)
```

In [4]:
```
from keras.applications import MobileNetV2
from keras.layers import Input,GlobalAveragePooling2D, Dropout, Dense
from keras import Model

base_model=MobileNetV2(include_top=False, weights='imagenet',input_shape=(160,160,3))

m=base_model.output
m=GlobalAveragePooling2D()(m)
m=Dense(128,activation='relu')(m) # dense layer 1
# m=Dense(64,activation='relu')(m) # dense layer 2
m=Dropout(0.5)(m)
m=Dense(32,activation='relu')(m) # dense layer 3
m=Dropout(0.5)(m)
preds=Dense(5,activation='softmax')(m) #final layer with softmax
model_final = Model(inputs=base_model.input,outputs=preds)

for layer in model_final.layers[:-6]:
    layer.trainable = False

from keras.callbacks import EarlyStopping, ModelCheckpoint, ReduceLROnPlateau
callback_list = [EarlyStopping(monitor='acc', patience=2)]

model_final.compile(optimizer='adam',
                    loss='categorical_crossentropy', metrics = ['acc'])

history = model_final.fit(x,y,epochs=9,batch_size=32,callbacks=callback_list)
```

```
Epoch 1/9
4993/4993 [==============================] - 21s 4ms/step - loss: 1.3544 - acc: 0.4516
Epoch 2/9
4993/4993 [==============================] - 13s 3ms/step - loss: 1.0460 - acc: 0.6153
Epoch 3/9
4993/4993 [==============================] - 13s 3ms/step - loss: 0.9342 - acc: 0.6767
Epoch 4/9
4993/4993 [==============================] - 13s 3ms/step - loss: 0.8878 - acc: 0.6904
Epoch 5/9
4993/4993 [==============================] - 13s 3ms/step - loss: 0.8297 - acc: 0.7114
Epoch 6/9
4993/4993 [==============================] - 13s 3ms/step - loss: 0.8043 - acc: 0.7282
Epoch 7/9
4993/4993 [==============================] - 13s 3ms/step - loss: 0.7724 - acc: 0.7470
Epoch 8/9
4993/4993 [==============================] - 13s 3ms/step - loss: 0.7668 - acc: 0.7442
Epoch 9/9
4993/4993 [==============================] - 13s 3ms/step - loss: 0.7161 - acc: 0.7486
```

```
In [5]: model_final.save('/content/drive/My Drive/응특연 기말레포트/최종/model_final.h5')
```

5. 테스트 데이터로 성능 확인

앞서 훈련 데이터를 전처리한 것과 마찬가지로 테스트 데이터 이미지 995장을 대상으로 동일한 전처리 과정을 거친 후, model_final.h5로 저장한 모형을 불러와 evaluate() 함수를 사용하여 일반화 정도를 확인해보았다. 결과는 정확도 약 0.71로 훈련 데이터에 대한 accuracy 0.7486과 큰 차이가 없어 과대적합도 어느정도 덜한 모형이란 것을 확인할 수 있다.

```
In [13]: model_final.evaluate(test_x, test_y)
         995/995 [==============================] - 4s 4ms/step
Out[13]: [1.0207770022914637, 0.7105527520179749]
```

6. 직접 촬영한 사진 분류

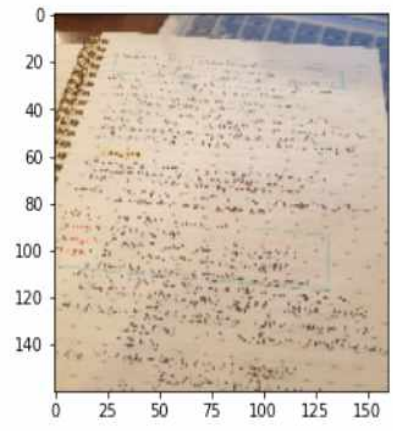
또한 직접 촬영한 사진을 바탕으로 예측을 한 결과는 다음과 같으며 사진이 속한 카테고리를 올바르게 분류하고 있는 것을 확인할 수 있다.

```
plt.imshow(test)
print('predict category:', y_original[y_class[0]])
```

```
predict category: 음식
```


```
plt.imshow(test)
print('predict category:', y_original[y_class[0]])
```

```
predict category: 책
```



6. 한계점 및 느낀점

우선 데이터를 직접 구하다보니 크롤링 과정에서 오류가 잦았으며 게시물 중 해시태그와 직접적인 관련이 없는 사진들이나, 광고성 게시글이 꽤 존재해서 모형의 성능이 일정 수준 이상 올라가지 않았다고 생각한다. 초반에는 과대적합이 굉장히 심하고 검증 데이터의 정확도는 0.35를 넘지 않았는데 원인을 찾아보니 실수로 x 데이터 이미지들을 255로 나눠 표준화하지 않았었다. 여기서 이미지 데이터를 처리할 때 표준화

의 중요성을 체감했다. 또한 모형을 적합할 시에는 하이퍼 파라미터의 작은 변화에도 모델의 성능이 크게 좌우되었기 때문에 세심한 튜닝이 필요했다. 뿐만 아니라, 임의로 결정한 5개의 카테고리이기 때문에 이를 보다 많은 범주의 수로 늘려볼 필요가 있으며 더 많은 훈련 데이터를 확보하여 모형을 확장시켜야할 필요가 있다고 생각한다.