# 전공종합설계1(가)
# 결과보고서

| 과목명 | 전공종합설계1(가) |
| --- | --- |
| 교수님 | 김석윤 교수님 |
| 학번 | 20152439 |
| 이름 | 최해민 |

# 목 차

0. 프로젝트(논문)명
머신러닝을 이용한 얼굴 특징점 분석을 통한 편측 저작 여부 추정

0. 팀명
Balanchew: Balanchew : Balance와 Chew를 결합하여 균형 있는 저작 운동을 목표로 한다는 뜻 (팀원 : 최해민, 개인으로 진행)

# 1. 서론

## 1-1. 기획 의도

음식을 섭취할 때, 습관적으로 한 쪽으로만 음식물을 씹는 사람이 많다. 하지만 지속적으로 한 쪽으로 음식물을 씹게 되면, 한쪽 치아와 안면 근육만을 사용하게 되어 안면 비대칭이 발생한다. 안면 비대칭 환자의 경우 치주 질환, 턱관절 장애, 두통 등 다양한 질환이 발생하기 때문에 이를 인지하고 예방해야 한다. 이에 본 연구에서는 신경망을 이용한 얼굴 특징점 분석을 통한 편측 저작(한쪽으로만 음식을 씹는 것)의 방향을 추정하였다.

## 1-2. 관련 연구

### 1-2-(1) Unliteral Chewing

저작이란 고형의 음식물 입자를 삼키기 쉽게 잘게 부수는 아래턱의 상하좌우 운동, 즉 음식물을 씹는 행위를 말한다. 음식을 섭취할 때 이상적인 저작 형태는 양쪽 치열을 모두 사용하는 양측 저작이다. 하지만, 여러 원인에 의해서 어느 한 쪽으로 저작하는 편측 저작을 하는 경우가 많은데, [1] 연구에서 치대 학생들의 저작 습관을 분석한 결과 77.8%가 편측 저작을 함을 보였다. 편측 저작 습관은 치아의 맹출로 인해 교합 간섭이나 교합면 마모 부족 등에 의해 발생할 수도 있고, 잇몸의 통증이나 치아의 상실, 습관, 교정 치료로 인한 치아의 이동 등이 원인이 되기도 한다.

지속적으로 한쪽으로 음식을 섭취할 경우 교합력, 골밀도, 측두 하악 장애 증상 등 신체에 여러 가지 문제가 발생한다. [3]연구에서는 편측 저작 습관이 있는 성인의 선호 저작측과 반대측의 치간 치조골 골밀도를 비교한 결과, 대부분의 부위에서 선호하는 저작 측면의 골밀도가 더 크게 나타났다는 결과를 보였다. [4] 연구는 편측 저작 습관자의 선호 저작 측면의 교합 접촉 면적과 교합력이 반대 측면에 비해서 크게 나타남을 보고했다. [2] 연구는 지속적인 편측 저작이 저작근의 과활성을 일으켜 측두 하악 장애의 원인이 될 것이라고 분석했고, [6] 연구에서도 측두 하악 장애 증에 따른 구강 행동 유형의 문항별 평균값을 비교한 결과, '한쪽으로 음식을 씹는 습관'의 문항에서 유의한 차이가 있는 것을 보고했다. 측두 하악 장애가 발생하면 악관절과 저작근의 통증, 하악 운동 범위의 제한, 턱관절 잡음, 두통, 안면 통증 등을 유발한다.[7] 또한, 한쪽으로만 씹게 되면, 저작 근육이 한쪽으로 더 발달하여 안면 비대칭을 유발할 수 있고, 자주 사용하지 않는 쪽의 치아에 음식물이 남아서 치주 질환 발생률이 높아진다.

### 1-2-(2) Chewing Detection

편측 저작 습관을 확인하고 예방하기 위해서는 저작 습관을 관측해야 한다. 현재 저작 습관을 분석하기 위한 다양한 방법들 중 하나는 전자식 통합 악기능검사 장비인 Biopak system을 턱관절, 안면 근육의 움직임, 하악의 운동 궤적을 측정하여 진단할 수 있다. 또한 저작 운동에 관한 다양한 연구가 진행되어 왔다. [8] 연구는 음식물에 따른 평균 측방거리가 4.9~6.7mm이고, 수직 개구량은 14.5~18.7mm로 [9]연구는 평균 측방거리가 4.2~5.3mm고, 수직 개구량이 13.5mm로 각각 분석하였다. [10] 연구에서는 우측 저작자가 좌측 저작자에 비해 저작 운동로가 정중선에서 더 편향되는 양상을 보였고, 우측 이환자의 경우 좌측 이환자에 비해 편향이 크게 나타났다고 밝혔다. 또한 [11] 연구에서 저작 운동을 할 때에 치아를 접촉하여 씹게 되는 쪽의 과두가 평균 2.5mm 아래로 이동하였고, 반대쪽 과두는 4mm 아래로 이동하였으며 7° 회전하였다고 보고하였다. 이처럼, 저작 운동 분석은 주로 저작 운동 방향과 속도, 크기를 관찰하는 방법을 많이 사용한다. 본 연구에서는 얼굴 인식 및 특징점 추출을 해주는 DeepFace를 활용하여 얼굴의 특징점의 움직임을 분석하여 저작 운동 방향을 예측하였다.

### 1-2-(3) Face Recognition을 활용한 연구

- Real-Time Driver-Drowsiness Detection System Using
얼굴 특징점 분석을 통해 눈 깜박임을 분석하여 운전자의 졸음 운전 여부 판단[18]
- Covfefe: A Computer Vision Approach For Estimating Force Exertion
얼굴 특징점의 움직임을 분석하여 힘을 주고 있는 정도 판별[19]

이처럼, 얼굴 특징점을 통해서 다양한 결과를 도출한 연구들이 많이 진행되어 왔다. 본 연구에서도 얼굴 특징점의 움직임을 SVM(Support Vector Machine)과 KNN(K-Nearest Neighbors)을 통해 저작 운동 방향을 분류하였다.

## 2. 본론1 - 데이터 수집 및 정제

### 2-1. Data Collection

| Female | Male | total |
|---|---|---|
| 10 | 1 | 11 |

표 1. 실험 참가자 수와 남녀 분포

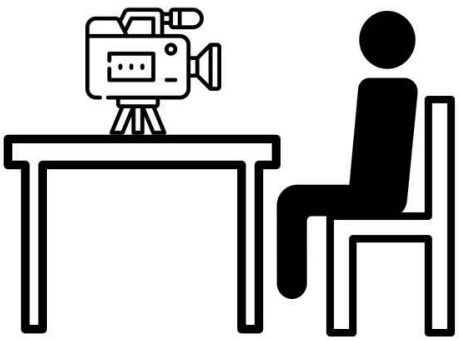


그림 1. 편측 저작 영상 촬영 방식

편측 저작 분류를 위한 데이터를 수집하기 위해 11명의 참여자를 대상으로 실험을 진행했다. 카메라를 바라보고 정면으로 앉은 상태에서 좌측과 우측으로 각각 10회씩 음식을 씹는 모습을 비디오로 촬영했다. 참여자의 나이 대는 20대 8명, 30대 1명, 50대 2명이고, 10명의 여성과 1명의 남성이 참여했다. 참여자 별 편측 속도 차이로 인해 영상의 길이에 약간의 차이가 있었고, 8±1.5초로 나타났다.

## 2-2. Video processing

촬영한 영상은 참여자 기준으로 좌, 우측으로 씹은 것이기 때문에 영상을 좌우 반전하는 과정을 먼저 진행하였다. 그리고 10회의 저작 운동을 회차별로 하나의 영상으로 트리밍하여 10개의 영상으로 만들고, 프레임 단위로 이미지를 추출하였다.

```python
import cv2

# Create VideoCapture object
vid_input = cv2.VideoCapture("../vid/CHM_right_Trim.mp4")
count = 0
print("start")

# check video file open successfully
while (vid_input.isOpened()):
    # Get frame from video, get success : ret = True / fail : ret= False
    ret, frame = vid_input.read()
    height, width, channel = frame.shape
    matrix = cv2.getRotationMatrix2D((width / 2, height / 2), 90, 1)
    frame = cv2.warpAffine(frame, matrix, (width, height))
    frame = cv2.flip(frame, 1)

    # fail to get frame
    if not ret:
        break
    # succeed
    else: # save the frame
        cv2.imwrite("right_%d.jpg" % count, frame)
    print('frame%d.jpg saved!' % count)
    count += 1

vid_input.release()
```

영상을 비디오 객체로 저장한 후에 vid_imput.read()를 통해 frame을 추출한다. 추출한 이미지를 flip(frame, 1)을 통해 영상을 좌우 반전하여 jpg 형식으로 저장했다. 프레임 별로 추출을 완료하면 비디오 객체 할당을 해제했다.

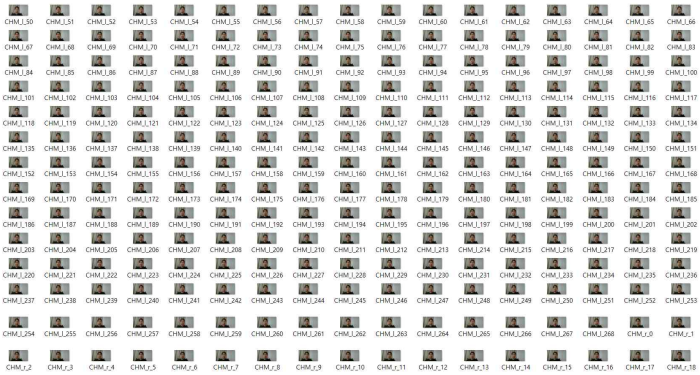


그림 2. 트리밍 후 프레임 단위로 추출한 영상들 중 일부

## 2-3. Face Detection & Landmark Prediction

| Research | Jaw | Mouth | Total |
|---|---|---|---|
| Dlib [11] | 17 | 20 | 68 |
| [12] | 1 | 4 | 20 |
| BoRMaN [13] | 0 | 2 | 5 |
| FACE++[14] | 19 | 18 | 106 |
| MTCNN[15] | 0 | 2 | 5 |

표 2. Face Landmark 검출 방법의 입, 턱, 전체 특징점 수


그림 3. (a) Dlib, (b) 연구 [12], (c) BorMan, (d) FACE++, (e) MTCNN 결과 예시

추출한 이미지에서 얼굴 특징점을 검출하기 위해서 Dlib을 활용했다. Dlib은 현재 가장 보편적으로 쓰이는 랜드마크 추출 방법이고, 본 연구에서 필요한 특징점인 입술과 턱에 대한 정보를 가장 잘 나타낸 기법이다. 얼굴 이미지에서 특징점을 검출하기 위하여 다양한 연구들이 진행되어 왔다.(표1) [12] 연구의 검출 방법, BoRMaN[13], MTCNN[15]은 입과 턱의 특징점 수가 적어서 저작 운동에 따른 변화를 잘 나타내지 못할 것이고, FACE++는 Dlib과 비슷한 특징점 수를 나타내지만 입술의 안쪽 특징점을 윗 입술과 아랫 입술로 구별하지 않아서 입을 벌리는 것을 판별하기 어려울 것이라고 판단했다.

Dlib 라이브러리를 활용하여 얼굴의 특징점을 찾기 위해서는, 이미지에서 얼굴을 찾아내는 Face Detection과 얼굴에서 landmark를 추출해내는 Face Shape Prediction 2단계를 진행한다. 먼저, Face Detector는 HOG(Histogram of Oriented Gradients)와 슬라이딩 윈도우를 이용한 SVM(Support Vector Machine) 선형 분류기를 통해 얼굴을 구별해낸다. 얼굴을 판별하고 나면, Face Shape Predictor를 통해 68개의 landmark의 위치를 추출한다.

Dlib은 CNN을 활용한 특징점 검출도 지원하고 있다. 본 연구에서 활용되는 이미지를 Dlib HOG detector과 Dlib CNN detector를 통해 landmark를 검출한 결과, CNN detector의 경우 그림 1-(b)와 같은 오류가 자주 발생하였다. 또한, Shape Prediction시에 194개의 landmark를 검출할 수 있는 데이터를 이용하여 특징점을 추출할 수 있는데, 그림 1-(c)와 같은 오류가 자주 발생하였다. 따라서, 본 연구에서는 HOG detector와 68개의 landmark 검출 데이터 셋을 활용하여 진행하였다.

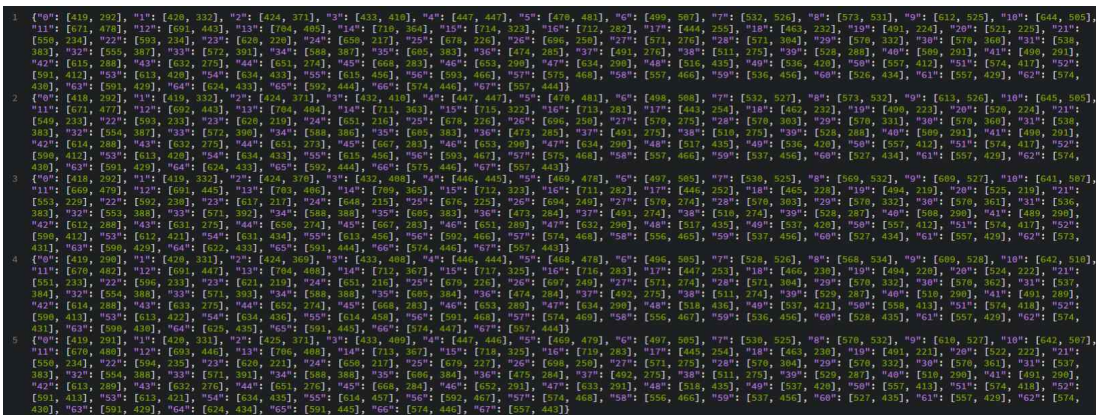
```python
import dlib
import cv2
import json

# create list for landmarks
ALL = list(range(0, 68))
RIGHT_EYEBROW = list(range(17, 22))
LEFT_EYEBROW = list(range(22, 27))
RIGHT_EYE = list(range(36, 42))
LEFT_EYE = list(range(42, 48))
NOSE = list(range(27, 36))
MOUTH_OUTLINE = list(range(48, 61))
MOUTH_INNER = list(range(61, 68))
JAWLINE = list(range(0, 17))

# create face detector, predictor
detector = dlib.get_frontal_face_detector()
predictor = dlib.shape_predictor('shape_predictor_68_face_landmarks.dat')

count = 0
```

ALL, RIGHT_EYEBROW, LEFT_EYEBROW, RIGHT_EYE, LEFT_EYE, NOSE, MOUTH_OUTLINE, MOUTH_INNER, JAWLINE 9개의 그룹으로 나누었고, 저장되는 landmark의 값은 각 리스트의 인덱스 값과 같다. dlib.get_frontal_face_detector()를 통해 detector를 생성하고, dlib.shape_predictor를 통해 predictor를 생성하였다. 랜드마크 검출에 이용되는 데이터 셋은 68개의 특징점을 찾을 수 있는 데이터 셋으로 입력하였다.

```python
count = 0
# capture the image in an infinite loop -> make it looks like a video
with open('./CHM_right' +'.json', "w") as json_file:
    while True:
        # Get frame from video
        image = cv2.imread('./video_frame/CHM_right_' + str(count) +'.jpg', cv2.IMREAD_COLOR)

       # resize the video
        image = cv2.resize(image, dsize=(640, 480), interpolation=cv2.INTER_AREA)
        img_gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

        # Get faces (up-sampling=1)
        face_detector = detector(img_gray, 1)
        # the number of face detected
        print("The number of faces detected : {}".format(len(face_detector)))

        # one loop belong to one face
        for face in face_detector:
            # face wrapped with rectangle
            cv2.rectangle(image, (face.left(), face.top()), (face.right(), face.bottom()), (0, 0, 255), 3)
            #create list to contain landmarks
            landmark_list = []
```

```python
            # detect 68 facial landmarks
            landmarks = predictor(image, face)

            # append (x, y) in landmark_list
            for p in landmarks.parts():
                landmark_list.append([p.x, p.y])
                cv2.circle(image, (p.x, p.y), 2, (0, 255, 0), -1)

            # transform landmark_list to dict and save in json
            key_val = [ALL, landmark_list]
            landmark_dict = dict(zip(*key_val))
            print(landmark_dict)
            json_file.write(json.dumps(landmark_dict))
            json_file.write('\n')

        cv2.imshow('result', image)
        cv2.imwrite('./test/CHM_r_' + str(count) +'.jpg', image)
        count += 1

        # wait for keyboard input
        key = cv2.waitKey(1)
        # if esc,
        if key == 27:
            break
# vid_in.release()
image.release()
```

프레임을 하나씩 불러와서 landmark를 추출한다. cv2.imread로 이미지를 불러와서 객체를 생성한 후, 필요한 경우 resize를 진행하고 cv2.cvtColor를 통해 회색조 이미지로 변환한다. face detector로 검출한 얼굴에서의 랜드마크를 predictor(image, face) 함수가 찾아서 landmarks에 저장한다. landmarks에 저장된 (x, y) 값을 {“0”:[x값, y값], “1”:[x값, y값]…} 형태로 딕셔너리에 저장한다. 열어놓은 json 파일에 그 딕셔너리를 추가하여 영상 전체에 대한 좌표값을 json으로 저장한다. 또한, cv2,circle, cv2,rectangle 함수를 이용하여 이미지 상에 특징점을 동그라미로 나타내고 얼굴을 감싸는 사각형도 함께 표시하여 저장하였다.




JSON 파일 예시

## 2-4. 데이터 전처리

### 2-4-(1) 참여자 별 얼굴 크기 및 위치 정규화

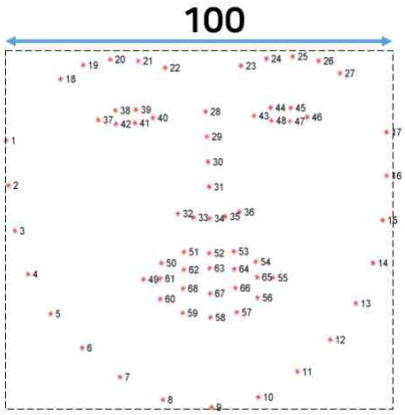



그림 5. 얼굴 정규화 (가로 100)

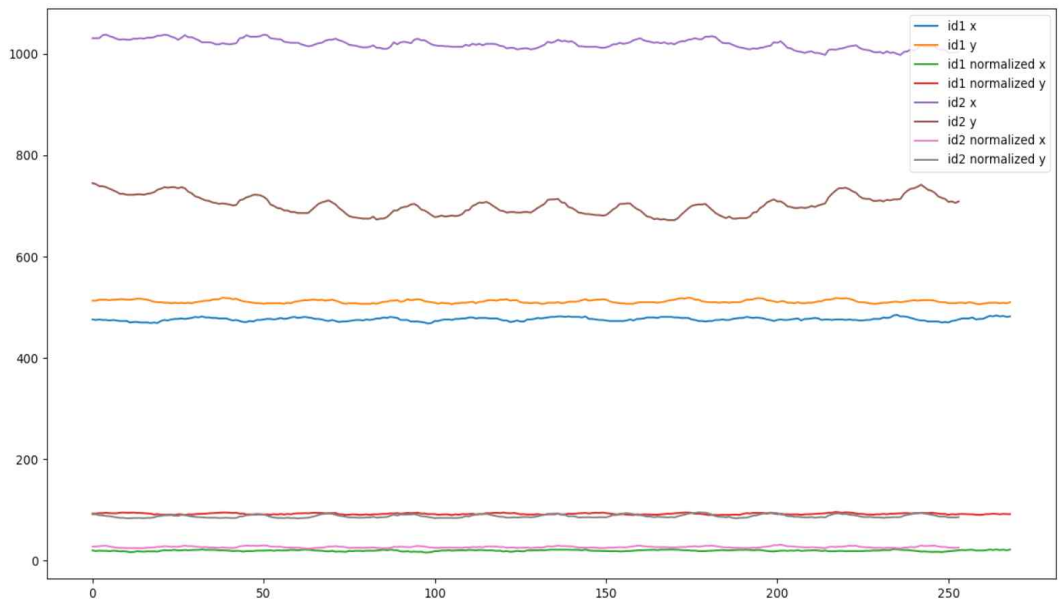
참여자들의 얼굴 크기에 차이가 있고, 카메라에 얼굴을 가까이 하거나 멀리함에 따라서, 그리고 영상에서의 얼굴의 위치에 따라서 좌표값이 다르게 나타난다. 그래서 영상에서 얼굴이 있는 부위만을 추출하고, 얼굴의 가로 길이를 100으로 변환하였다. 먼저 얼굴의 x좌표값과 y좌표값의 최대와 최소값을 구한 후에 이미지의 가로 길이는 저작 운동 시에 크게 변화하지 않았고, 세로 길이는 저작 운동 시에 턱을 움직이게 되면 상하로의 변화가 크기 때문에 고정하지 않았다.

```python
import json
import pandas as pd
import math

x = pd.DataFrame()
y = pd.DataFrame()
x1 = pd.DataFrame()
y1 = pd.DataFrame()

with open("./video_lm/id1_right.json", "r") as f:
    lines = f.readlines()
    count = 0
    for line in lines:
        count += 1
        # print(count)
        data = json.loads(line)
        df = pd.DataFrame.from_dict(data, orient='index').T
        # print(df)

        xlist = df.loc[0]
        xlist_df = pd.DataFrame(xlist).T
        ylist = df.loc[1]
        ylist_df = pd.DataFrame(ylist).T

        xdist = xlist[xlist.idxmax(axis=1)] - xlist[xlist.idxmin(axis=1)]
        ydist = ylist[ylist.idxmax(axis=1)] - ylist[ylist.idxmin(axis=1)]
```

```python
        x1 = pd.concat([x1, xlist_df])
        y1 = pd.concat([y1, ylist_df])

        normalized_x = (xlist - xlist[xlist.idxmin(axis=1)]) * (100 / xdist)
        normalized_x = pd.DataFrame(normalized_x).T
        x = pd.concat([x, normalized_x])
        # print(normalized_x)

        normalized_y = (ylist - ylist[ylist.idxmin(axis=1)]) * (100 / xdist)
        normalized_y = pd.DataFrame(normalized_y).T
        y = pd.concat([y, normalized_y])
        # print(normalized_y)

    x1.reset_index(drop=True)
    y1.reset_index(drop=True)

    x1.to_csv('./video_lm/id1_right_x_normalized.csv', sep = ',', na_rep = 'NaN', index=False)
    y1.to_csv('./video_lm/id1_right_y_normalized.csv', sep = ',', na_rep = 'NaN', index=False)
```

랜드마크 별 좌표값을 딕셔너리 형태로 저장한 json파일을 열어서 각 좌표의 x값과 y값을 나눠서 xlist와 ylist에 저장했다. xylis와 ylist의 최대값과 최소값을 찾아서 그 차를 구해 각각 xdist, ydist에 저장했다. xdist와 ydist는 현재 얼굴의 가로와 세로 길이를 나타낸다. 100에서 이를 나눈 값을 x 좌표값에 곱하면 정규화된 x 좌표값을 구할 수 있다. json파일에 있는 모든 좌표값을 정규화 시켜서 x값을 normalized_x에, y값을 normalized_y에 저장한 후 csv 파일로 변환하여 저장하였다.

```
plt.figure(figsize=(16, 9))
plt.plot(vis_df, label='id1 x')
plt.plot(vis_df2, label='id1 y')
plt.plot(vis_dfn, label='id1 normalized x')
plt.plot(vis_df2n, label='id1 normalized y')

plt.plot(vis2_df, label='id2 x')
plt.plot(vis2_df2, label='id2 y')
plt.plot(vis2_dfn, label='id2 normalized x')
plt.plot(vis2_df2n, label='id2 normalized y')

plt.legend(loc='upper right')
plt.show()
```



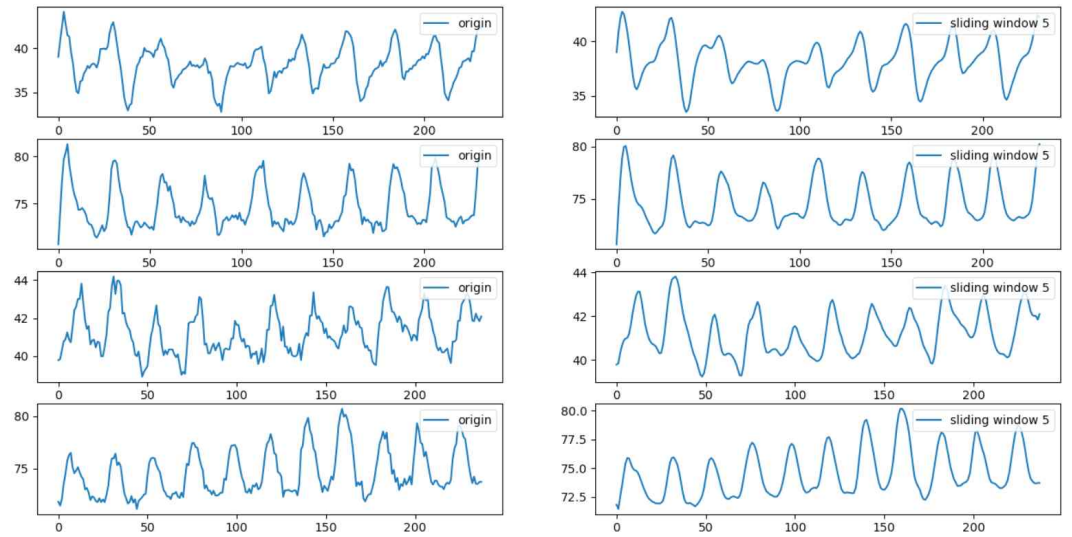

두 사람을 각각 id1, id2라고 했을 때, 정규화 전과 정규화 후의 6번 랜드마크의 좌표 데이터를 matplotlib.pyplot을 이용해서 나타냈다. vis_df와 vis_df2는 id1의 x 좌표와 y좌표가 입력된 데이터 프레임이고, 뒤에 n을 붙여 정규화된 데이터를 입력했다. id2에 대해서도 같은 과정을 진행한 후 이를 그래프로 나타내면 다음과 같다. id1의 x, y 좌표는 500내외에 위치하고, id2의 x는 약 700, y는 약 1000에 위치한다. 하지만 정규화 후의 그래프를 보게 되면 id1과 id2의 x좌표와 y좌표가 거의 동일한 범위 내에서 위치하는 것을 확인할 수 있다.

## (2) 데이터 필터링

```python
import pandas as pd
import matplotlib.pyplot as plt

lx11=pd.read_csv("../video_lm/id1_left_x.csv")
ly11=pd.read_csv("../video_lm/id1_left_y.csv")
rx11=pd.read_csv("../video_lm/id1_right_x.csv")
ry11=pd.read_csv("../video_lm/id1_right_y.csv")

lxn = pd.DataFrame()
for i in range(0,68):
    lxtmp = lx11[str(i)].T
    lxtmp2 = pd.DataFrame([lxtmp[0]])
    lxtmp2 = lxtmp2.append([lxtmp[1]])
    for j in range(2, len(lx11) - 2):
        lxtmp2 = lxtmp2.append([(lxtmp[j-2] + 2* lxtmp[j-1]
                                 + 3 * lxtmp[j] + 2 * lxtmp[j+1] + lxtmp[j+2]) / 9])
    lxtmp2 = lxtmp2.append([lxtmp[len(lx11) - 2]])
    lxtmp2 = lxtmp2.append([lxtmp[len(lx11)-1]])
    lxn = pd.concat([lxn,lxtmp2.T])
lxn = lxn.reset_index(drop=True).T.reset_index(drop=True)
```

```python
x1, y1, x2, y2 = lx11['59'], lxn[59], ly11['59'], lyn[59]
x3, y3, x4, y4 = rx11['59'], rxn[59], ry11['59'], ryn[59]

lxn.to_csv('../move_dir/filtered_data/Sis_left_x_5.csv', sep=',', na_rep='NaN', index=False)
lyn.to_csv('../move_dir/filtered_data/Sis_left_y_5.csv', sep=',', na_rep='NaN', index=False)
rxn.to_csv('../move_dir/filtered_data/Sis_right_x_5.csv', sep=',', na_rep='NaN', index=False)
ryn.to_csv('../move_dir/filtered_data/Sis_right_y_5.csv', sep=',', na_rep='NaN', index=False)

plt.figure(figsize=(12, 7))
plt.subplot(421)
plt.plot(x1, label="origin")
plt.legend(loc='upper right')

plt.subplot(422)
plt.plot(y1, label="sliding window 5")
plt.legend(loc='upper right')

plt.subplot(423)
plt.plot(x2, label="origin")
plt.legend(loc='upper right')

plt.subplot(424)
plt.plot(y2, label="sliding window 5")
plt.legend(loc='upper right')

plt.subplot(425)
plt.plot(x3, label="origin")
plt.legend(loc='upper right')

plt.subplot(426)
plt.plot(y3, label="sliding window 5")
plt.legend(loc='upper right')

plt.subplot(427)
plt.plot(x4, label="origin")
plt.legend(loc='upper right')

plt.subplot(428)
plt.plot(y4, label="sliding window 5")
plt.legend(loc='upper right')
plt.savefig("../move_dir/filtered_data/Sis5.png")
plt.show()
```

얼굴 특징점의 움직임을 그래프로 나타내보면, 얼굴 특징점 검출이 잘못 나타나는 부분도 발생하고 미세한 움직임이 모두 나타나기 때문에 데이터 필터링을 진행했다. 먼저, 데이터를 가시화하여 범위 내에서 크게 벗어나는 데이터를 제거한 후, 현재 점의 index를 i라고 했을 때, i-2, i-1, i, i+1, i+2를 각각 [1, 2, 3, 2, 1]의 가중치를 두어 필터링을 진행했다. 위는 좌측 저작 운동의 x좌표 움직임에 대해서만 나타낸 것이고, 좌우측 x, y 좌표의 움직임을 모두 변환하였다. 필터링 전, 후의 그래프는 다음과 같다.



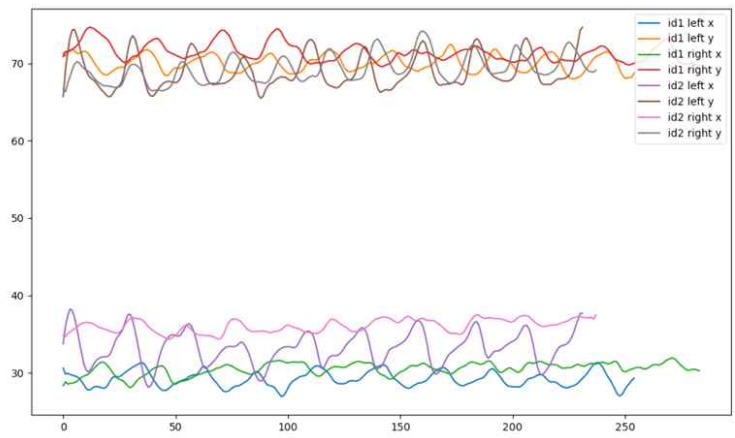

id2의 59번 랜드마크의 움직임을 나타낸 그래프이다. 그래프가 형태는 유지하면서도 노이즈 없이 부드럽게 나타나는 것을 볼 수 있다. 또한, 전체적인 흐름의 변화를 줄여 이후에 랜드마크의 움직임을 분석할 때에 오차가 적게 발생하도록 했다.

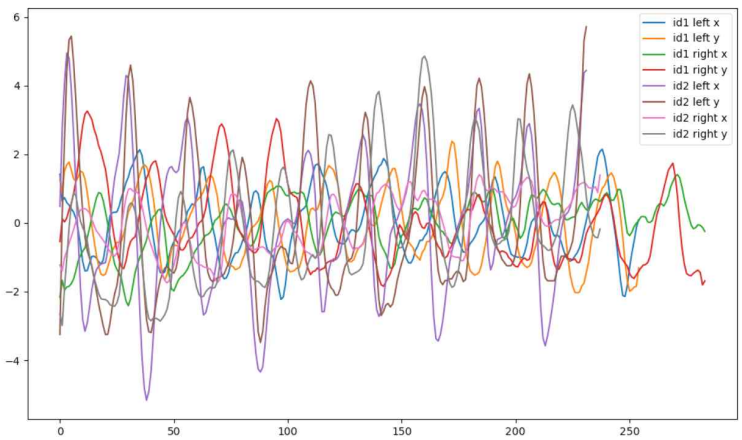
## (3) 좌표값 정규화

```python
import pandas as pd
import matplotlib.pyplot as plt


lx=pd.read_csv("../move_dir/filtered_data/CHM_left_x_5.csv")
ly=pd.read_csv("../move_dir/filtered_data/CHM_left_y_5.csv")
rx=pd.read_csv("../move_dir/filtered_data/CHM_right_x_5.csv")
ry=pd.read_csv("../move_dir/filtered_data/CHM_right_y_5.csv")

avg = [0 for _ in range(0,68)]
# calculate left x average
for i in range(0,68):
    sum = 0
    for j in range(0, len(lx)):
        sum += lx[str(i)][j]
    avg[i] = sum / len(lx)

# all data - average
for i in range(0,68):
    for j in range(0, len(lx)):
        lx[str(i)][j] -= avg[i]

lx.to_csv('../move_dir/filtered_data/Sis_left_x_5_nor.csv', sep=',', na_rep='NaN', index=False)
ly.to_csv('../move_dir/filtered_data/Sis_left_y_5_nor.csv', sep=',', na_rep='NaN', index=False)
rx.to_csv('../move_dir/filtered_data/Sis_right_x_5_nor.csv', sep=',', na_rep='NaN', index=False)
ry.to_csv('../move_dir/filtered_data/Sis_right_y_5_nor.csv', sep=',', na_rep='NaN', index=False)
```

```
x1, x2, x3, x4 = alllx[str('48')], ally[str('48')], allrx[str('48')], allry[str('48')]
y1, y2, y3, y4 = alllx2[str('48')], ally2[str('48')], allrx2[str('48')], allry2[str('48')]

plt.figure(figsize=(12, 7))
plt.plot(x1, label="id1 left x")
plt.plot(x2, label="id1 left y")
plt.plot(x3, label="id1 right x")
plt.plot(x4, label="id1 right y")
plt.plot(y1, label="id2 left x")
plt.plot(y2, label="id2 left y")
plt.plot(y3, label="id2 right x")
plt.plot(y4, label="id2 right y")
plt.legend(loc='upper right')
plt.show()
```

68개의 데이터를 저장할 avg 리스트를 만들고, 한 사람의 저작 운동을 하는 동안의 68개의 랜드마크 데이터의 좌표값의 합을 구한 후 총 데이터 개수로 나누어 평균값을 구한다. 이후에, 각 좌표값에서 평균값을 빼서 0을 기준으로 좌표가 움직이도록 수정하여 csv 파일로 저장했다.




id1과 id2의 48번 랜드마크의 좌측으로 씹을 때와 우측으로 씹을 때의 x, y 좌표를 나타낸 그래프이다. 정규화 전에는 x, y 좌표의 변화를 한 눈에 보기 어려웠지만, 정규화 이후에는 한눈에 볼 수 있다. 또한, 0을 기준으로 그래프가 분포하기 때문에 각 좌표의 움직임 크기도 쉽게 파악할 수 있다.

## 3. 본론 - 데이터 분석

### 3-1. 특징점 움직임 분석

### 3-1-(1) 특징점 좌표값 분석

```python
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns


xfile = pd.read_csv('./move_dir/filtered_data/Sis_left_x_5_nor.csv')
yfile = pd.read_csv('./move_dir/filtered_data/Sis_left_y_5_nor.csv')
xlist = pd.DataFrame()
ylist = pd.DataFrame()

for i in range(len(xfile)):
    df = pd.DataFrame(xfile.loc[i])
    df.rename(columns={i: 'value'}, inplace=True)
    xlist = pd.concat([xlist, df], axis=0)

for i in range(len(yfile)):
    df = pd.DataFrame(yfile.loc[i])
    df.rename(columns={i: 'value'}, inplace=True)
    ylist = pd.concat([ylist, df], axis=0)
```

```python
xlist["axis"] = 'x'
ylist["axis"] = 'y'
xlist = xlist.reset_index()
ylist = ylist.reset_index()
xlist['index'] = xlist['index'].astype(int)
ylist['index'] = ylist['index'].astype(int)

plt.figure(figsize=(8,6))
plt.title('Landmarks analysis')
plt.subplot(211)
sns.boxplot(x='index', y='value', data=xlist)
sns.swarmplot(x="index", y="value", data=xlist, size = 1)
plt.xlabel('face landmarks')
plt.ylabel('value')

plt.subplot(212)
sns.boxplot(x='index', y='value', data=ylist)
sns.swarmplot(x="index", y="value", data=ylist, size = 1)
plt.xlabel('face landmarks')
plt.ylabel('value')

plt.legend(loc='upper right')
plt.show()
```

정규화까지 완료한 데이터로 각 랜드마크 별 움직임을 boxplot을 통해 가시화시켜 보았다.

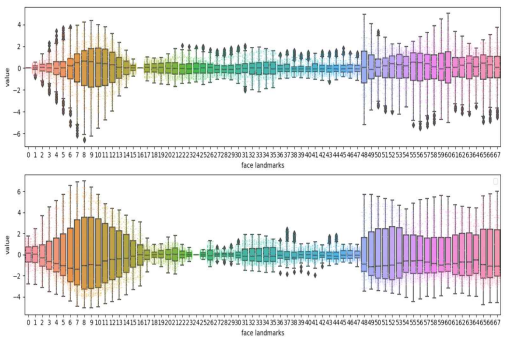


그림 5. id2의 좌편측 저작 운동시 x좌표값(위)과 y좌표값(아래) 변화 그래프

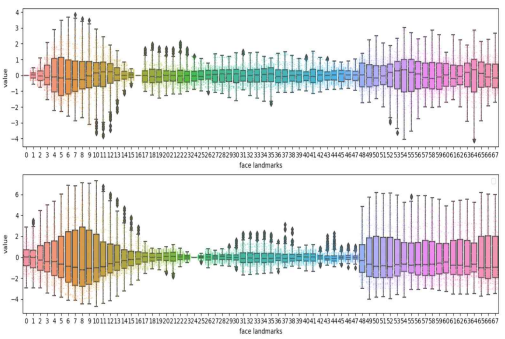


그림 6. id2의 우편측 저작 운동시 x좌표값(위)과 y좌표값(아래) 변화 그래프

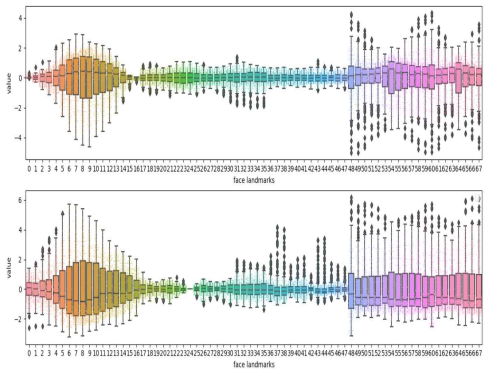


그림 7. id3의 좌편측 저작 운동시 x좌표값(위)과 y좌표값(아래) 변화 그래프

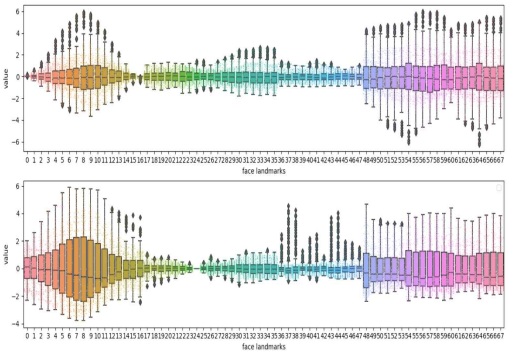


그림 8. id3의 우편측 저작 운동시 x좌표값(위)과 y좌표값(아래) 변화 그래프

먼저, 좌측과 우측으로 씹었을 때에 얼굴 특징점의 움직임 정도가 차이가 날 것이라고 판단을 하고 그래프를 가시화 시켜보았다. 좌측으로 씹을 때는 좌측의, 우측으로 씹을 때는 우측의 특징점들의 좌표값 변화가 클 것이라고 예상했지만, 그렇지 않았다. 위 그림은 id1과 id3의 좌측으로 음식을 씹었을 때와 우측으로 음식을 씹었을 때의 x, y값을 boxplot으로 나타낸 것이다. 턱 부분(0~16번)과 입 부분(48번~67번)의 움직임이 크게 나타나는 것은 확인할 수 있지만 좌측과 우측의 유의미한 차이를 찾기는 어려웠다.

또한, x좌표값을 기반으로 SVM(Support Vector Machine)과 KNN(K-Nearest Neighbors) 학습을 진행한 후 분류한 결과가 각각 54%, 36%로 낮은 결과가 나타났다.

## (2) 특징점 이동 거리 분석

```python
all_x = pd.read_csv('./move_dir/filtered_data/CHM_left_x_5_nor.csv')
all_y = pd.read_csv('./move_dir/filtered_data/CHM_left_y_5_nor.csv')
all_x2 = pd.read_csv('./move_dir/filtered_data/CHM_right_x_5_nor.csv')
all_y2 = pd.read_csv('./move_dir/filtered_data/CHM_right_y_5_nor.csv')

ptrn = [29, 58, 86, 112, 140, 166, 187, 211, 235, 253]
ptrn2 = [3, 33, 64, 87, 115, 151, 174, 203, 228, 251, 277]

dist_avg = pd.DataFrame()
tmp_sum = pd.DataFrame()

for j in range(len(ptrn) - 1):
    count = 0
    tmp_sum = [0 for _ in range(0, 68)]
    for i in range(ptrn[j], ptrn[j + 1]):
        count += 1
        dist = [0 for _ in range(0, 68)]
        for k in range(0, 68):
            dist[k] = math.sqrt(((all_x[str(k)][i] - all_x[str(k)][i + 1]) ** 2) + \
                                ((all_y[str(k)][i] - all_y[str(k)][i + 1]) ** 2))
            tmp_sum[k] += dist[k]
    for q in range(0, 68):
        tmp_sum[q] = tmp_sum[q] / count
    dist_avg = pd.concat([dist_avg, pd.DataFrame(tmp_sum).T])

dist_avg = dist_avg.reset_index(drop=True)
dist_avg_t = dist_avg.T
```

```python
plt.figure(figsize=(16, 9))
plt.subplot(211)
sns.boxplot(data=dist_avg)
# plt.title("average movement boxplot")
plt.xlabel("landmark num")
plt.ylabel("movement")

plt.subplot(212)
sns.boxplot(data=dist_avg2)
plt.title("average movement boxplot")
plt.xlabel("landmark num")
plt.ylabel("movement")
plt.show()
```

$$d = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

공식을 이용하여 한 점의 현재 프레임(i)의 x, y 좌표와 다음 프레임(i+1)의 x, y 좌표 간의 거리를 계산하였다. 1회 씹는 동안의 움직임의 합을 구하여 dist 리스트에 저장하였고, 이를 프레임 수로 나눠서 1회 씹는 동안의 움직임의 평균값을 구하였다. 이를 그래프로 가시화 시킨 것은 아래와 같다.

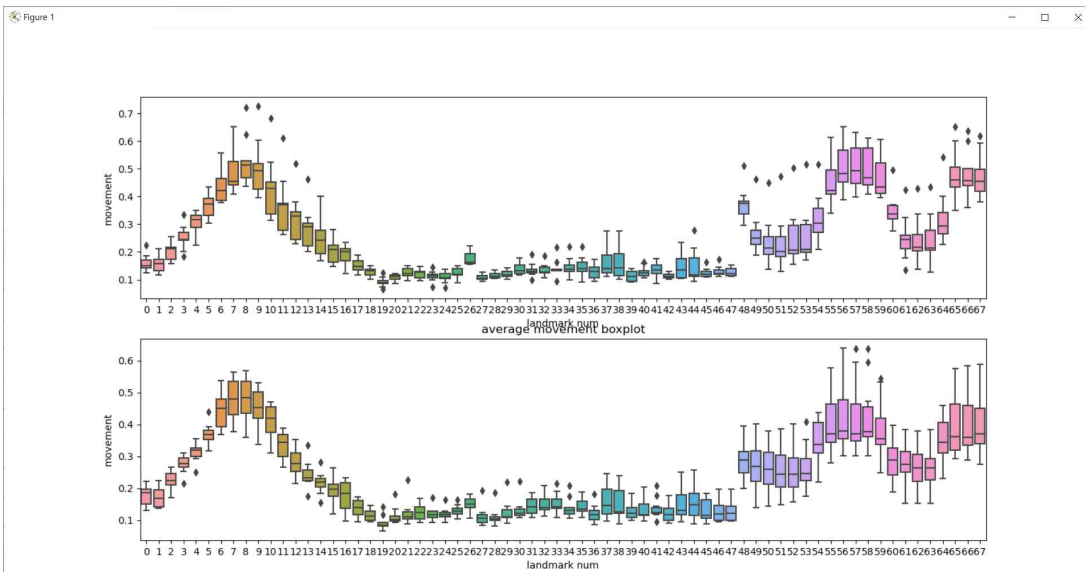



그림 9. id1의 좌측(위)과 우측(아래)로 씹을 때의 특징점의 움직인 거리 그래프

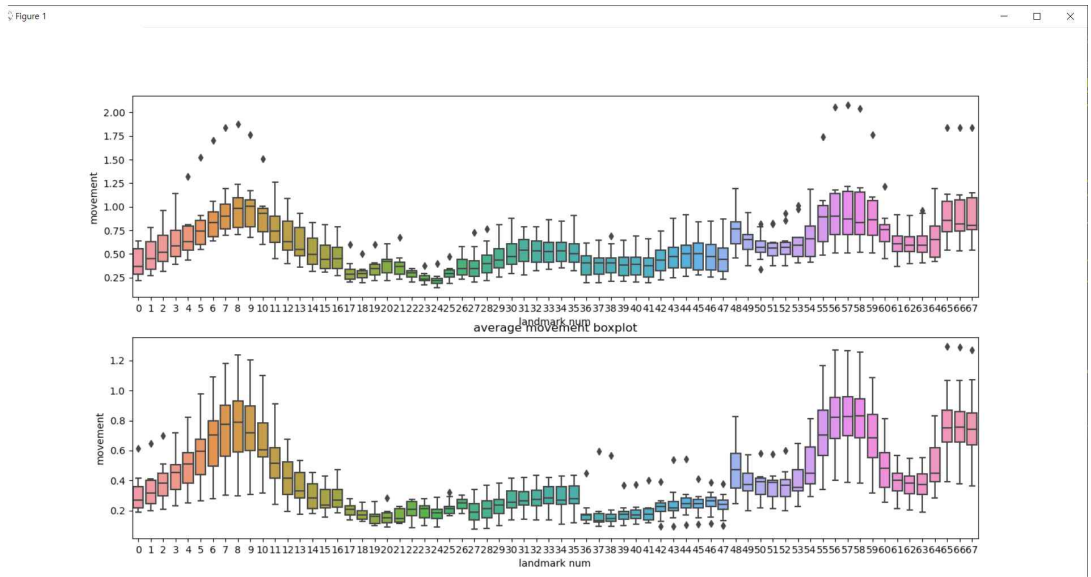



그림 10. id3의 좌측(위)과 우측(아래)로 씹을 때의 특징점의 움직인 거리 그래프

id1과 id3의 좌측과 우측으로 씹을 때의 특징점의 움직인 거리를 가시화 시킨 결과를 분석한 결과 유의미한 결론을 내릴 수 없었다. 또한, 특징점의 움직인 거리를 통해 SVM(Support Vector Machine)과 KNN(K-Nearest Neighbors) 학습을 시킨 후 분류한 결과 55.7%, 42.7%의 낮은 결과가 나타났다.

## (3) 특징점 벡터값 분석

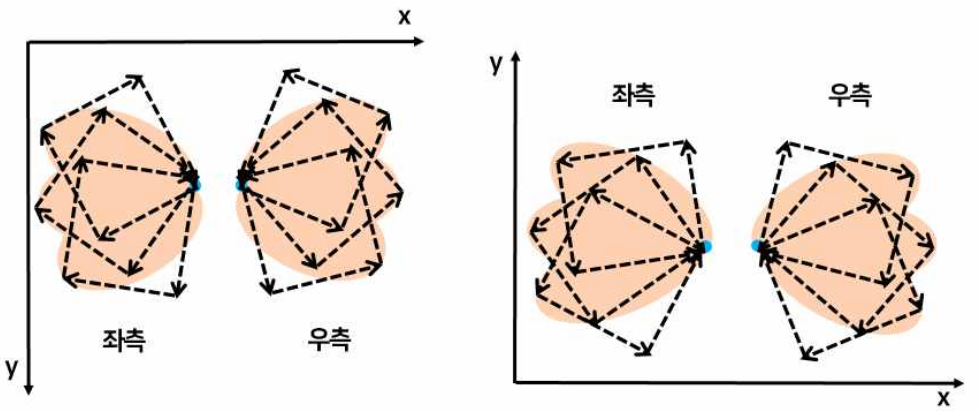
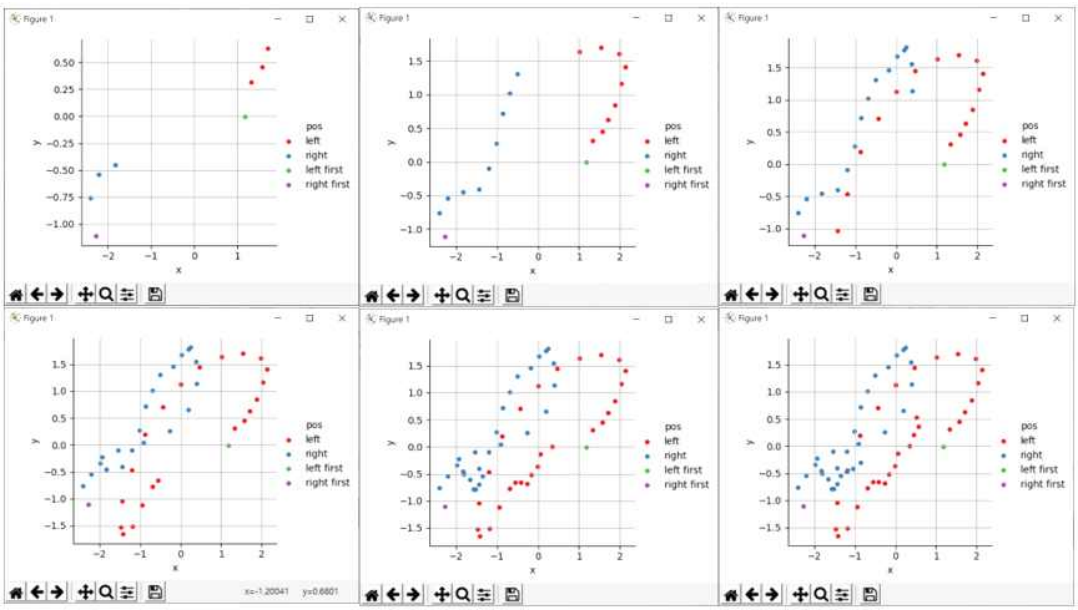



그림 11. 저작 운동 영상에서 좌측으로 씹을 때와 우측으로 씹을 때의 점들의 예상 운동 경로 (좌), 축의 방향을 바꾸기 위해 상하 반전을 한 예상 운동 경로 (우)

좌표값의 변화량과 움직인 거리로는 분류가 잘 되지 않아서 특징점의 벡터값에 대한 분석을 해보았다. 참여자의 저작 운동 영상을 확인해본 결과 좌측으로 씹을 때와 우측으로 씹을 때의 특징점이 움직이는 방향에서 위와 같은 차이가 보였다. 이를 확인해보기 위해 1회의 저작 운동 시에 점이 움직이는 경로를 가시화해보았다.

```python
ptrn = [29, 58, 86, 112, 140, 166, 187, 211, 235, 253]
ptrn2 = [33, 64, 87, 115, 151, 174, 203, 228, 251, 277]
rnge = [48, 54, 66]
for i in rnge:
    for j in range(len(ptrn)-1):
        cnt = 0
        while True:
            if cnt > ptrn[j+1] - ptrn[j]:
                break
            cnt += 3
            alll = pd.concat([pd.DataFrame(lx[str(i)].loc[ptrn[j]:ptrn[j] + cnt]).T,
                              pd.DataFrame(ly[str(i)].loc[ptrn[j]:ptrn[j] + cnt]).T]).T
            alll['pos'] = 'left'
            alll.columns = ['x', 'y', 'pos']
            allr = pd.concat([pd.DataFrame(rx[str(i)].loc[ptrn[j]:ptrn[j] + cnt]).T,
                              pd.DataFrame(ry[str(i)].loc[ptrn[j]:ptrn[j] + cnt]).T]).T
            allr['pos'] = 'right'
            allr.columns = ['x', 'y', 'pos']
            all = pd.concat([alll, allr])
            all.reset_index(drop=True)

            first = all.loc[ptrn[j]]
            first['pos'] = ['left first', 'right first']
            all = all.drop([ptrn[j]])
            all = all.append(first)

            sns.FacetGrid(all, hue="pos", palette='Set1', height=4).map(plt.scatter, "x", "y", s=15).add_legend()
            plt.grid()
            plt.show()
```

x, y 좌표값 데이터프레임을 합한 후 pos, 즉 'left'와 'right' 방향 별로 나눠서 scatterplot을 만들어보았다. 영상에서 1회 씹을 때마다의 특징점의 움직임을 알기 위해, 프레임 순서대로 점이 추가적으로 찍히도록 구현하였다. 그리고 씹기 시작할 때의 첫 번째 점을 'left first'와 'right first'로 지정하여 점이 움직이기 시작하는 점을 가시화하였다.




id2의 48번 점의 움직임을 나타내 본 결과가 다음과 같았다. 사람마다, 점마다 차이가 있었지만 대부분의 특징점들이 위에서 예상한 이동 경로와 비슷하게 나타난 것을 확인할 수 있었다.

## 3-2. 좌, 우측 저작 운동의 벡터값 비교

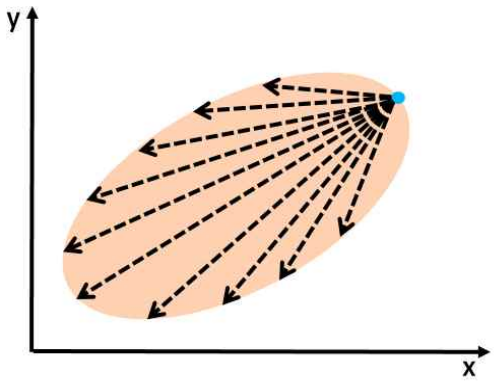



그림 12. 첫 번째 프레임과 이후의 프레임들 간의 좌표값의 벡터값

음식을 1회 씹을 때에 첫 번째 프레임에서 다른 프레임으로의 특징점 위치 변화를 벡터값으로 할 때, y 좌표의 벡터값은 일정하지 않았지만, x좌표의 벡터값은 좌측으로 씹었을 때는 대체로 음수(-)값이, 우측으로 씹었을 때는 대체로 양수(+)값을 보이는 것을 알 수 있었다. 정확하게 확인하기 위해서 이를 구현하였다.

```python
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

lx = pd.read_csv('./move_dir/filtered_data/id2_left_x_5_nor.csv')
ly = pd.read_csv('./move_dir/filtered_data/id2_left_y_5_nor.csv')
rx = pd.read_csv('./move_dir/filtered_data/id2_right_x_5_nor.csv')
ry = pd.read_csv('./move_dir/filtered_data/id2_right_y_5_nor.csv')

ptrn = [0, 23, 50, 74, 97, 132, 150, 180, 199, 225]
ptrn2 =[0, 25, 48, 71, 93, 114, 136, 152, 176, 199, 214]
rnge = [48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67]

vect_lx_cnt, vect_ly_cnt, vect_rx_cnt, vect_ry_cnt = [], [], [], []
vect_lx_all, vect_ly_all, vect_rx_all, vect_ry_all = [], [], [], []

for k in rnge:
    print('=========== landmark no.' + str(k) + '==============')
    print('---------------left---------------')
    for i in range(len(ptrn) - 1):
        lx1 = lx[str(k)][ptrn[i]]
        ly1 = ly[str(k)][ptrn[i]]
        for j in range(ptrn[i] + 1, ptrn[i + 1]):
            vect_lx = lx[str(k)][j] - lx1
            vect_ly = ly[str(k)][j] - ly1
            vect_lx_all.append(vect_lx)
            vect_ly_all.append(vect_ly)
```

```
vect_lx_df, vect_rx_df = pd.DataFrame(vect_lx_all), pd.DataFrame(vect_rx_all)
vect_ly_df, vect_ry_df = pd.DataFrame(vect_ly_all), pd.DataFrame(vect_ry_all)

alll, allr = pd.concat([vect_lx_df.T, vect_ly_df.T]).T, pd.concat([vect_rx_df.T, vect_ry_df.T]).T
alll['pos'], allr['pos'] = 'left', 'right'
all = pd.concat([alll, allr])
all.columns = ['x', 'y', 'pos']

cnt = 0
for i in vect_lx_all:
    if i < 0:
        cnt += 1
print('lx total: ' + str(len(vect_lx_all)) + ', lx < 0: ' + str(cnt))

cnt = 0
for i in vect_rx_all:
```

```
cnt = 0
for i in vect_ry_all:
    if i < 0:
        cnt += 1
print('ry total: ' + str(len(vect_ry_all)) + ', ry < 0: ' + str(cnt))

sns.FacetGrid(all, hue="pos", height=4).map(plt.scatter, "x", "y").add_legend()
plt.grid()
plt.show()
```

48(49번 랜드마크)부터 67(68번 랜드마크)까지 특징점들의 벡터값을 구한 것의 일부이다. 음식을 1회 씹는 동안의 첫 번째 프레임의 좌표값 x를 lx에 저장하고 y를 ly에 저장했다. 이후의 프레임들의 x, y값에서 lx, ly의 값을 뺀 것을 리스트로 저장하여 scatterplot으로 나타냈다. 또한, 프레임 좌표값의 수와 그 중 0보다 작은 것의 수를 출력하였다.

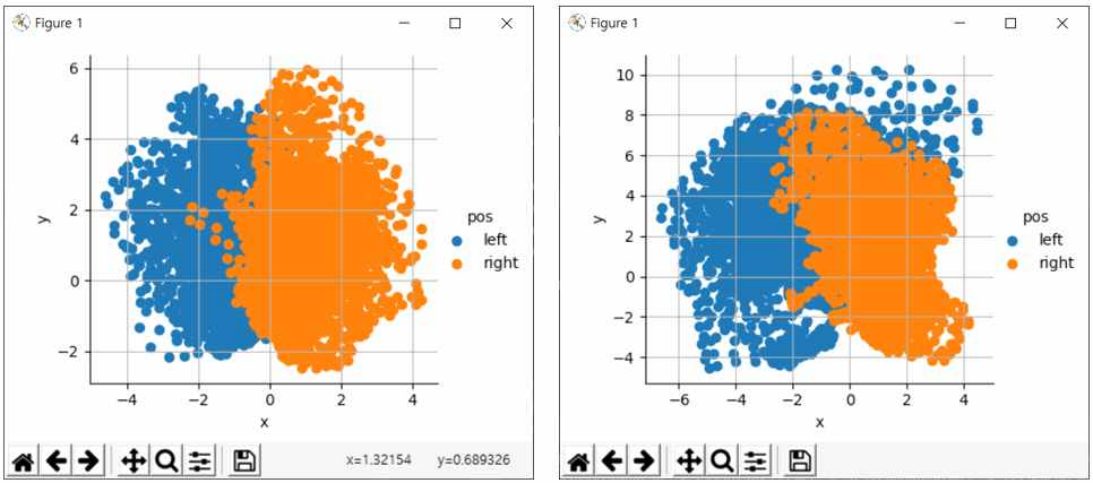



그림 13. id1(좌)과 id2(우)가 음식을 왼쪽으로 씹을 때와 오른쪽으로 씹을 때의 벡터값 그래프

```
----------------id1-------------------
lx total: 4300, lx < 0: 3536
rx total: 4780, rx < 0: 520
ly total: 4300, ly < 0: 1529
ry total: 4780, ry < 0: 1566
```

```
----------------id2-------------------
lx total: 4320, lx < 0: 3228
rx total: 4080, rx < 0: 841
ly total: 4320, ly < 0: 896
ry total: 4080, ry < 0: 1472
```

그림 14. id1(위)과 id2(아래)가 음식을 왼쪽으로 씹을 때의 벡터의 x(lx), y(ly)값 전체 원소 수와 0보다 작은 원소 수, 오른쪽으로 씹을 때의 벡터의 x(rx), y(ry)의 값 전체 원소 수와 0보다 작은 원소 수

음식을 1회 씹는 동안의 첫 번째 프레임의 특징점(49번~68번) 좌표값에서 나머지 프레임의 좌표값 간의 벡터값을 확인해보았다. 그래프와 출력 결과에서 모두 확인할 수 있듯이, 0보다 작은 lx(음식을 왼쪽으로 씹을 때의 x벡터 값) 수가 rx 수보다 많은 것을 확인할 수 있었다.

```
----------------id1-------------------
lx total: 7955, lx < 0: 6173
rx total: 8843, rx < 0: 1388
ly total: 7955, ly < 0: 2958
ry total: 8843, ry < 0: 2807
```

```
----------------id1-------------------
lx total: 14620, lx < 0: 10057
rx total: 16252, rx < 0: 5162
ly total: 14620, ly < 0: 6052
ry total: 16252, ry < 0: 5776
```

그림 15. 1~17번과 49~68번 랜드마크의 벡터값 출력 결과(위)와 1~68번 랜드마크의 벡터값 출력 결과

아래 결과는 눈썹, 눈, 코의 벡터값을 합한 결과이다. 눈썹과 눈과 코는 씹는 방향과 관계 없이 거의 동일하게 위치하므로 벡터값의 x 좌표가 0 내외에서 작은 차이로 움직일 것이다. 결과에서도 볼 수 있듯이, 위의 결과와 다르게 lx와 rx가 0보다 작은 점의 수가 크게 차이나지 않는 것을 알 수 있었다. lx < 0과 rx < 0의 비율을 나타내면 다음과 같다.

- 1~17, 49~68번 랜드마크 결과 : 0.22
- 1~68번 랜드마크 결과 : 0.51

## 3-3. 머신러닝을 통한 저작 운동 방향 분류

### 3-3-(1) PCA

먼저 전체 랜드마크가 유의미한 분류를 할지 알아보기 위해 PCA(Principal Component Analysis)를 통해 차원 축소를 하여 가시화해보았다. 현재 랜드마크는 68개이고, 이 중 턱과 입을 제외한 나머지는 37개이다. 이 중 움직임이 크지 않을 1, 2, 3, 15, 16, 17번 랜드마크를 제외하면 31개의 랜드마크가 된다. 이 31개의 랜드마크를 모두 feature로 사용할 경우 31차원이 되어 알아보기 힘들다. 따라서 분류를 잘 할 수 있는 방향으로 차원을 축소하였다.

```
pca = PCA(n_components=3)
X_pca_array = pca.fit_transform(X)
PCAdf = pd.DataFrame(X_pca_array, columns = ['Principal Component 1','Principal Component 2','Principal Component 3'])
finalDf = pd.concat([PCAdf, y], axis=1)
print(finalDf)
print('[PCA1, PCA2, PCA3] : ', end="")
print(pca.explained_variance_ratio_)
print('3개의 Principal Component으로 전체 분산의 ', end="")
print(sum(pca.explained_variance_ratio_), end="")
print('을 설명 가능하다.')
```

sklearn.decomposition 라이브러리의 PCA를 활용하여 차원 축소를 진행해보았다. PCA(n_components=3)을 통해 3개의 성분으로 전체 분산을 설명할 수 있는지 확인해보았다.

```
[PCA1, PCA2, PCA3] : [0.84546735 0.08097362 0.03312121]
3개의 Principal Component으로 전체 분산의 0.9595621870336601을 설명 가능하다.
```

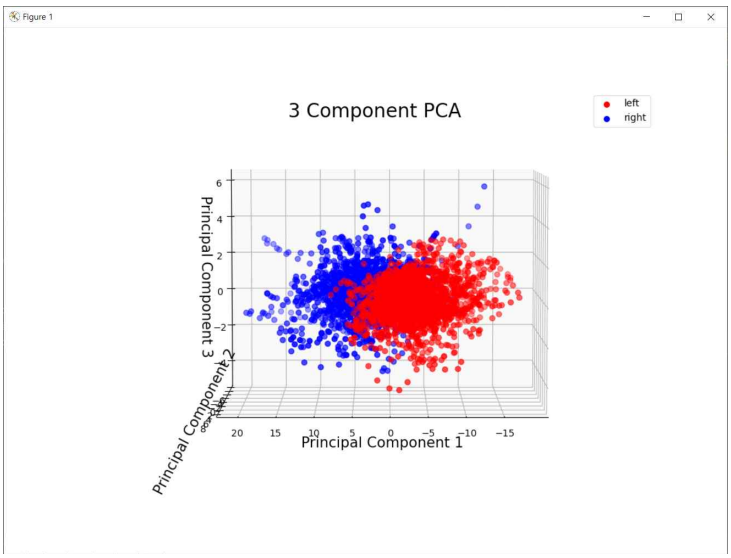



그림 16. 3개의 PCA로 차원을 축소한 결과 그래프

출력한 결과를 보면, PCA1 성분만으로 84.5%를 설명할 수 있고, PCA2와 PCA3는 각각 8%, 3.3%를 설명함을 알 수 있다. 3개의 성분으로 전체 분산의 약 95%를 설명할 수 있으므로 차원 축소가 잘 이루어진 것을 알 수 있다. 또한, 이를 그래프로 나타낸 것이 그림과 같고, 이는 left와 right의 분류를 잘 하고 있는 것을 확인할 수 있다.

## 3-3-(2) Feature Extraction

PCA 가시화를 통해서 랜드마크를 통해 유의미한 분류가 가능함을 보았다. 31개의 feature 중에는 상관관계가 큰 것이 있을 수 있고, 이는 학습에 좋지 않은 영향을 주기 때문에, 랜덤 포레스트와 피어슨 상관계수를 이용하여 특징 추출을 진행했다. 피어슨 상관계수 값이 0.7 이상 또는 -0.7 이하면 상관도가 크며, 0.9 이상 또는 -0.9 이하면 매우 크다고 할 수 있다.(Mukaka, 2012; O'Rourke and Hatcher, 2013).

```python
select = ['0', '1', '2', '3', '4', '5', '6', '7', '8', '9', '10', '11', '12', '13', '14',
          '15', '16', '48', '49', '50', '51', '52', '53', '54', '55', '56', '57', '58', '59',
          '60', '61', '62', '63', '64', '65', '66', '67']

data = mov.loc[:,select]
target = mov["pos"]
cor = data.corr()
cor = pd.DataFrame(cor)
cor.to_csv('correlation.csv')
print(cor[['58', '66', '9', '14', '2']])

# Correlation Map
plt.figure(figsize=(30,30))
sns.heatmap(data = data.corr(), annot=True,fmt = '.2f', linewidths=.5, cmap='Blues')
plt.show()

training_data, validation_data , training_labels, validation_labels = \
    train_test_split(data, target, stratify=target, random_state=0)

# RandomForest Feature Importance
n_feature = 37
index = np.arange(n_feature)
forest = RandomForestClassifier(n_estimators=100, n_jobs=-1)
forest.fit(training_data, training_labels)
plt.barh(index, forest.feature_importances_, align='center')
plt.yticks(index, select)
plt.ylim(-1, n_feature)
plt.xlabel('feature importance', size=15)
plt.ylabel('feature', size=15)
plt.show()
```

37개의 feature들 중 Feature Importance가 높은 것을 뽑아내기 위한 코드이다. 먼저 corr() 함수를 통해 feature들 간의 Pearson 상관계수를 구했고, 이를 heatmap으로 나타냈다. 그리고 sklearn.ensemble 라이브러리의 RandomForestClassifier()를 이용하여 feature importance를 나타냈다. n_estimators를 100으로 하여 100개의 트리를 만든다. 결과값인 37개의 feature에 대한 각 feature의 중요도를 그래프로 나타냈다.

| | 58 | 54 | 14 | 2 | 9 |
|---|---|---|---|---|---|
| 0 | -0.162771 | -0.308014 | -0.147675 | -0.212732 | -0.209249 |
| 1 | 0.557678 | 0.497653 | 0.316523 | 0.829031 | 0.629016 |
| 2 | 0.707409 | 0.568910 | 0.428836 | 1.000000 | 0.751839 |
| 3 | 0.787122 | 0.533189 | 0.505528 | 0.891645 | 0.749504 |
| 4 | 0.815598 | 0.501633 | 0.548280 | 0.777642 | 0.739738 |
| 5 | 0.845301 | 0.584669 | 0.611395 | 0.760255 | 0.798120 |
| 6 | 0.877397 | 0.698480 | 0.680170 | 0.772172 | 0.872803 |
| 7 | 0.904765 | 0.762902 | 0.722007 | 0.775080 | 0.930157 |

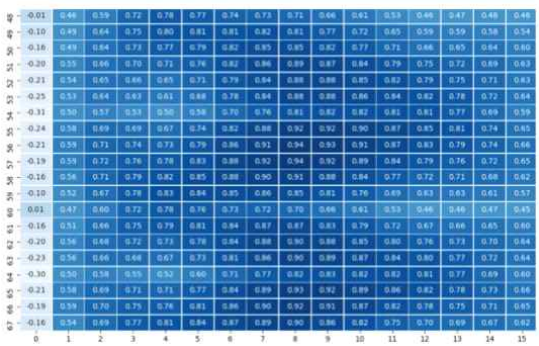


그림 17. 각 feature의 상관계수를 출력한 것의 일부(좌)와 이를 heatmap으로 가시화한 그래프의 일부(우)

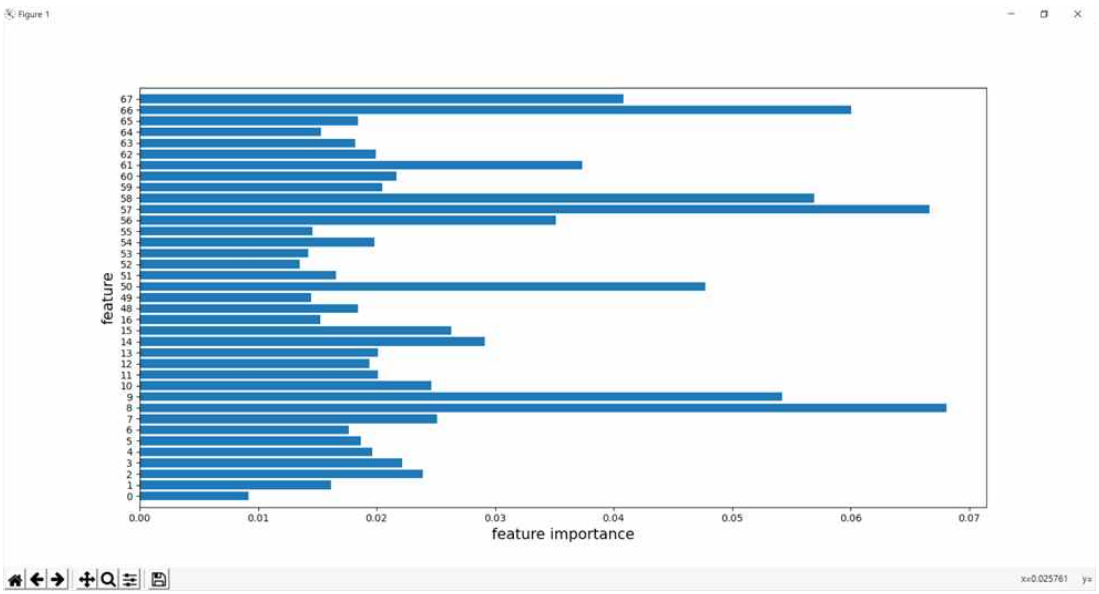
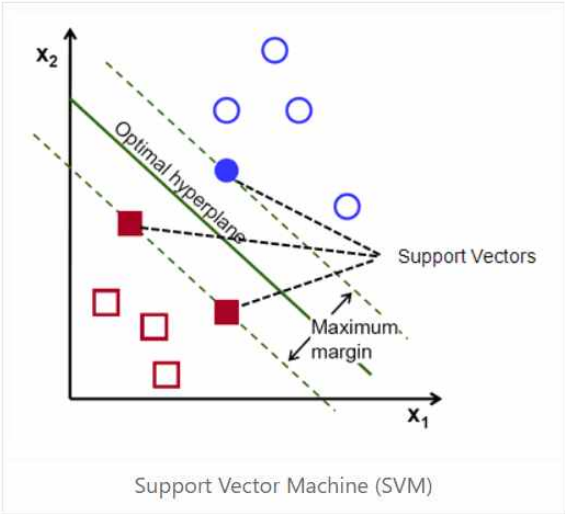



그림 18. 각 feature의 feature importance를 나타낸 그래프

상관계수가 중요한 이유는, 상관계수가 큰 두 변수를 모두 feature로 사용하게 되면 학습 성능이 낮아지기 때문이다. 그래서 먼저 feature importance가 큰 상위 15개의 feature를 추출했다. 이는 (58, 57, 66, 9, 8, 50, 51, 62, 67, 10, 7, 14, 15, 2, 61)이다. 이 중요도가 가장 높은 58을 선택하고, 58과의 상관계수가 0.75이상이거나 -0.75이하인 feature을 모두 제거하였다. 남은 (2, 14, 15) 중 feature importance가 큰 14를 선택하였고, 14와의 상관계수가 큰 15는 제거하였다. 남은 feature 2는 14와의 상관계수가 0.43으로 낮기 때문에, feature로 선택하였다. 최종적으로 선택한 feature은 (2, 14, 58) 3개이다.

## 3-3-(3) SVM(Support Vector Machine) 분류

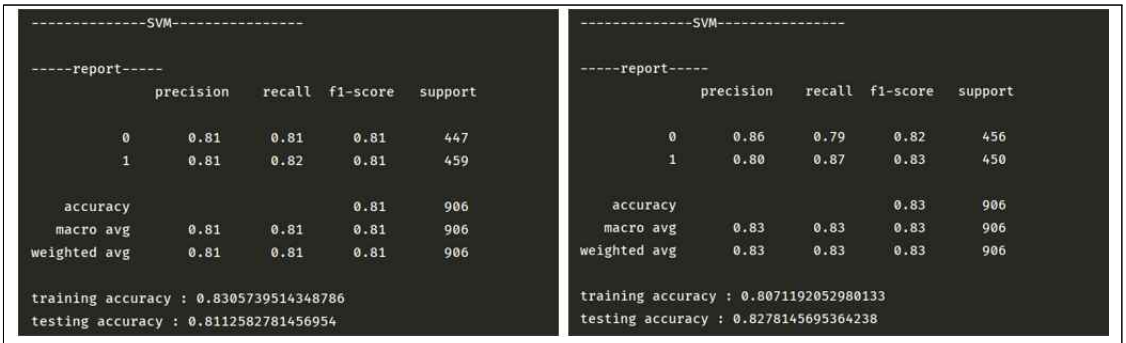



Support Vector Machine (SVM)

SVM(Support Vector Machine)은 자료를 분류하는 Optimal한 hyperplane을 찾고 margin이 최대가 되게 분류하도록 학습하는 방법이다. 학습데이터로 학습을 시킨 후 테스트 데이터로 학습 결과를 확인해봐야 한다. 본 연구에서는 전체 데이터를 기준으로 랜덤으로 20%의 데이터를 뽑아서 테스트 데이터로, 나머지를 학습데이터로 하는 것을 첫 번째로 진행한다.

```
right_mov['pos'] = 'right'
mov = pd.concat([left_mov, right_mov])
mov.loc[mov.pos=='left', 'pos'] = 0
mov.loc[mov.pos=='right', 'pos'] = 1
mov = mov.astype({'pos':'int'})

select = ['58', '14', '2']

data = mov.loc[:, select]
target = mov['pos']

#학습 데이터와 테스트 데이터 나누기
data_train, data_test, label_train, label_test\
    = train_test_split(data, target, test_size=0.2)

print('--------------SVM----------------')
clf = svm.SVC(kernel='linear')
clf.fit(data_train, label_train)
# Prediction
predict = clf.predict(data_test)

# Testing
ac_score = metrics.accuracy_score(label_test, predict)
cl_report = metrics.classification_report(label_test, predict)
print("\n-----report-----\n", cl_report)
print('training accuracy : ', end="")
print(clf.score(data_train, label_train))
print('testing accuracy : ', end="")
print(clf.score(data_test, label_test))
```

sklearn.svm 라이브러리의 함수를 활용하여 SVM 학습 결과를 출력하는 코드를 구현한 것이다. select에 feature가 될 랜드마크 값을 입력하고, train_test_split을 활용하여 전체 데이터의 0.2만을 테스트 데이터로 추출한다. data_train은 학습 데이터 중 입력값, label_train은 학습 데이터 중 출력값, data_test는 테스트 데이터 중 입력값, label_test는 테스트 데이터 중 출력값을 나타낸다. 그리고 svm.SVC(kernel='linear')을 활용하여 선형 SVM 분류기를 clf로 저장한다. 이때, 'rbf', 'poly', 'linear' 커널 중 'linear' 커널의 학습 결과가 가장 좋았고, c값과 gamma의 값은 default이 가장 성능이 좋아서 default로 지정하였다. clf를 통해 predict한 결과를 바탕으로 training accuracy와 testing accuracy를 출력하였다.



```
--------------SVM----------------

-----report-----
              precision    recall  f1-score   support

           0       0.81      0.81      0.81       447
           1       0.81      0.82      0.81       459

    accuracy                           0.81       906
   macro avg       0.81      0.81      0.81       906
weighted avg       0.81      0.81      0.81       906

training accuracy : 0.8305739514348786
testing accuracy : 0.8112582781456954
```

```
--------------SVM----------------

-----report-----
              precision    recall  f1-score   support

           0       0.86      0.79      0.82       456
           1       0.80      0.87      0.83       450

    accuracy                           0.83       906
   macro avg       0.83      0.83      0.83       906
weighted avg       0.83      0.83      0.83       906

training accuracy : 0.8071192052980133
testing accuracy : 0.8278145695364238
```

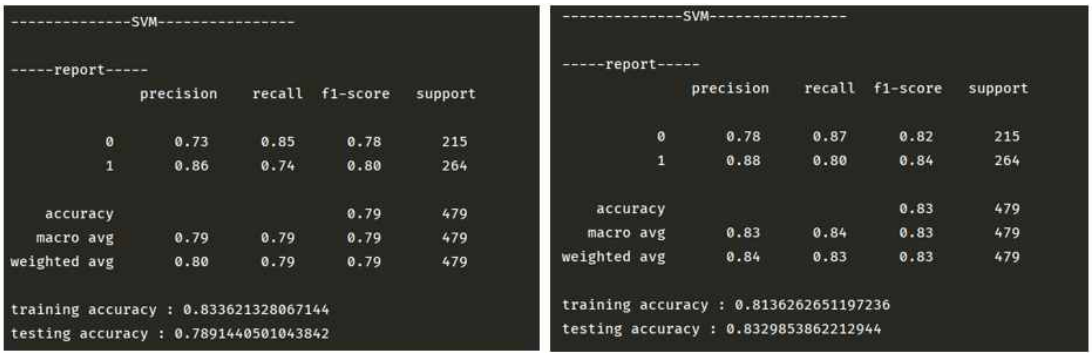
그림 19. 랜드마크 [1, ..., 17, 48, ..., 68]를 기반으로 한 SVM 결과(좌)와 랜드마크 [58, 14, 2]를 기반으로 한 SVM 결과 (train_test_split)

학습 결과는 위 그림 중 오른쪽 그림과 같다. 학습 정확도는 약 81%, 테스트 정확도는 83%가 나왔다. 학습데이터와 테스트 데이터 생성이 랜덤이다 보니 실행할 때마다 약간의 차이는 있지만, 거의 비슷하게 나타난다. 또한, 위 그림 중 왼쪽 결과는 feature 추출을 하지 않았을 때의 결과를 나타낸 것이다. 거의 비슷하게 나타나기 때문에 feature를 줄이는 것이 유의함을 알 수 있다.

하지만 학습데이터의 같은 참여자의 데이터로 학습하고 테스트할 경우 성능이 올라가는 효과가 날 수 있기 때문에, 10명의 참여자의 데이터로 학습을 한 후 다른 1명의 참여자 데이터로 테스트를 하는 방법을 두 번째로 진행하였다.

```python
left_train_mov = pd.read_csv("../move_dir/vector_data/left_x_train1.csv")
left_mov_test = pd.read_csv("../move_dir/vector_data/id1_left_x.csv")
right_train_mov = pd.read_csv("../move_dir/vector_data/right_x_train1.csv")
right_mov_test = pd.read_csv("../move_dir/vector_data/id1_right_x.csv")

left_train_mov['pos'] = 'left'
right_train_mov['pos'] = 'right'
left_mov_test['pos'] = 'left'
right_mov_test['pos'] ='right'

train_mov = pd.concat([left_train_mov, right_train_mov])
train_mov = train_mov.reset_index(drop=True)
test_mov = pd.concat([left_mov_test, right_mov_test])
test_mov = test_mov.reset_index(drop=True)

train_mov.loc[train_mov.pos=='left', 'pos'] = 0
train_mov.loc[train_mov.pos=='right', 'pos'] = 1
train_mov = train_mov.astype({'pos':'int'})
test_mov.loc[test_mov.pos=='left', 'pos'] = 0
test_mov.loc[test_mov.pos=='right', 'pos'] = 1
test_mov = test_mov.astype({'pos':'int'})
```

구현 방법은 거의 비슷하지만, 학습 데이터와 테스트 데이터 생성시 train_test_split을 사용하지 않고 한명의 데이터를 test 데이터로, 나머지 참여자의 데이터를 train 데이터로 저장하여 사용하였다.



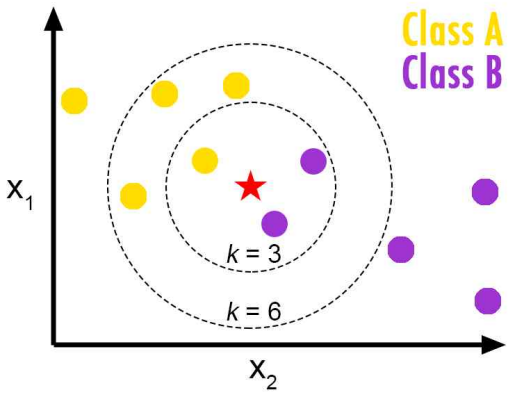
```
--------------SVM----------------

-----report-----
              precision    recall  f1-score   support

           0       0.73      0.85      0.78       215
           1       0.86      0.74      0.80       264

    accuracy                           0.79       479
   macro avg       0.79      0.79      0.79       479
weighted avg       0.80      0.79      0.79       479

training accuracy : 0.833621328067144
testing accuracy : 0.7891440501043842
```

```
--------------SVM----------------

-----report-----
              precision    recall  f1-score   support

           0       0.78      0.87      0.82       215
           1       0.88      0.80      0.84       264

    accuracy                           0.83       479
   macro avg       0.83      0.84      0.83       479
weighted avg       0.84      0.83      0.83       479

training accuracy : 0.8136262651197236
testing accuracy : 0.8329853862212944
```

그림 20. 랜드마크 [1, ..., 17, 48, ..., 68]를 기반으로 한 SVM 결과(좌)와 랜드마크 [58, 14, 2]를 기반으로 한 SVM 결과 (참여자 10명 train, 1명 test)

새로운 참여자 데이터로 테스트한 결과도 약 83%의 테스트 정확도를 나타낸다. 이 결과에서는 랜드마크를 모두 활용한 경우에 테스트 데이터가 79%로 조금 낮게 나오는 것을 확인할 수 있다. 또한, 위 결과는 id1를 새로운 참여자 데이터로 하여 진행한 결과이다. 다른 참여자를 테스트 데이터로 했을 때 약간의 정확도 차이가 발생하긴 하지만 80±3%의 정확도를 나타냈다.

### 3-3-(4) KNN(K-Nearest Neighbors)을 이용한 분류

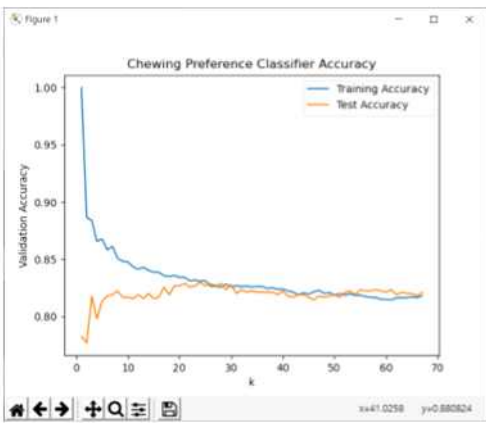



K분류할 데이터와 가장 가까이에 있는 학습 데이터가 속한 그룹을 찾는다. k=3이라면, 3개의 데이터가 발견될 때까지 원 모양을 확장하면서 발견된 3개의 데이터 중 가장 많은 그룹을 새로운 데이터의 그룹으로 분류한다. 또한, k는 항상 홀수여야 한다. 위 그림에서, k = 3일 때 데이터는 class B로 분류되고, k = 6일 때 class A로 분류된다. SVM 학습 방법과 동일하게 train_test_split으로 나눈 test 데이터와 새로운 참여자 데이터를 테스트 데이터로 하는 두가지 방식으로 진행한다.

```
print('---------------KNN-------------------')
classifier = KNeighborsClassifier(n_neighbors = 25)
classifier.fit(training_data, training_labels)

print(classifier.score(validation_data, validation_labels))

predict=classifier.predict(validation_data)
ac_score=metrics.accuracy_score(validation_labels, predict)
cl_report=metrics.classification_report(validation_labels, predict)
print("\n-----report-----\n", cl_report)
print('training accuracy : ', end="")
print(classifier.score(training_data, training_labels))
print('testing accuracy : ', end="")
print(classifier.score(validation_data, validation_labels))
```

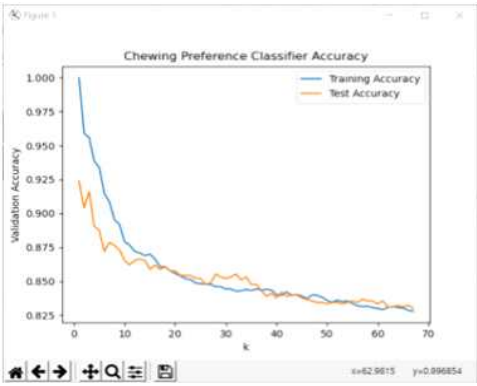
```
training_accuracy = []
test_accuracy = []
k_list = range(1,68)
for k in k_list:
  classifier = KNeighborsClassifier(n_neighbors = k)
  classifier.fit(training_data, training_labels)
  training_accuracy.append(classifier.score(training_data, training_labels))
  test_accuracy.append(classifier.score(validation_data, validation_labels))
plt.plot(k_list, training_accuracy, label="Training Accuracy")
plt.plot(k_list, test_accuracy, label="Test Accuracy")
plt.xlabel("k")
plt.ylabel("Validation Accuracy")
plt.title("Chewing Preference Classifier Accuracy")
plt.legend()
plt.show()
```

KNN은 sklearn.neighbors의 KNeighborsClassifier을 이용하여 분류한다. n_neighbors는 k의 수를 바꿔가면서 training accuracy와 test accuracy의 변화를 관찰하여 선택한다. 이 때의 n_neighbors는 25일 때 train과 test가 모두 적절한 정확도를 나타냈기 때문에 25로 선택하였다. 25개의 k를 이용하여 분류하는 분류기를 classifier에 저장하고, 이를 training 데이터와 라벨을 통해 fit한 후에, predict 함수를 이용하여 예측하였다. 예측한 결과를 바탕으로 training accuracy와 testing accuracy를 출력하였다.




```
3624
906
---------------KNN-------------------
0.826710816777042

-----report-----
              precision    recall  f1-score   support

           0       0.83      0.82      0.82       442
           1       0.83      0.84      0.83       464

    accuracy                           0.83       906
   macro avg       0.83      0.83      0.83       906
weighted avg       0.83      0.83      0.83       906

training accuracy : 0.8314017660044151
testing accuracy : 0.826710816777042
```

그림 21. 랜드마크[58, 2, 14]를 이용한 KNN 학습 결과



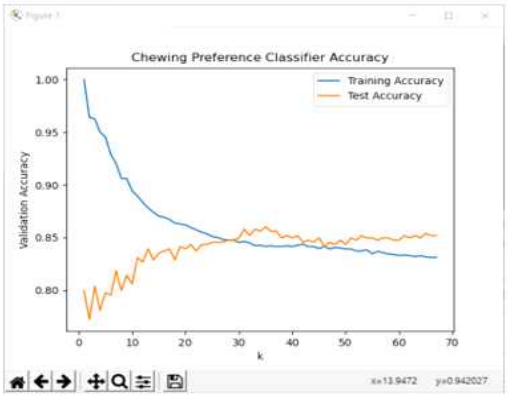

```
3624
906
---------------KNN-------------------
0.8520971302428256

-----report-----
              precision    recall  f1-score   support

           0       0.86      0.83      0.85       442
           1       0.85      0.87      0.86       464

    accuracy                           0.85       906
   macro avg       0.85      0.85      0.85       906
weighted avg       0.85      0.85      0.85       906

training accuracy : 0.8482339955849889
testing accuracy : 0.8520971302428256
```

그림 22. 랜드마크[1~17, 49~68]을 이용한 KNN 학습 결과

train_test_split을 통해서 학습 데이터와 테스트 데이터를 나누어 학습한 결과는 다음과 같다. 3개의 feature을 통해 학습한 결과 학습 정확도는 약 83%, 테스트 정확도는 약 82.7%로 나타났고, 이는 랜덤으로 추출된 데이터이기 때문에 약간의 차이가 발생한다.
아래 그림은 전체 feature을 통하여 KNN을 진행한 결과인데, training accuracy와 test accuracy가 매우 높게 나왔다. 이는 같은 참여자의 데이터가 학습 데이터와 테스트 데이터에 모두 포함되어 발생한 결과라고 예상할 수 있다.


```
2020
215
---------------KNN-------------------

-----report-----
              precision    recall  f1-score   support

           0       0.78      0.85      0.81       215
           1       0.87      0.80      0.83       264

    accuracy                           0.82       479
   macro avg       0.82      0.83      0.82       479
weighted avg       0.83      0.82      0.82       479

training accuracy : 0.82670945445569
testing accuracy : 0.8225469728601252
```

그림 23. 랜드마크 [58, 14, 2]를 기반으로 한 KNN 결과 (참여자 10명 train, 1명 test)

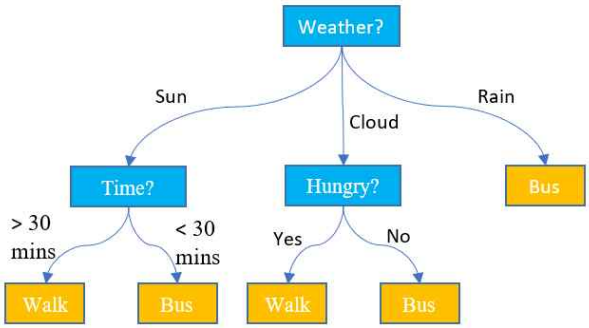



```
2020
215
---------------KNN-------------------

-----report-----
              precision    recall  f1-score   support

           0       0.82      0.89      0.85       215
           1       0.90      0.84      0.87       264

    accuracy                           0.86       479
   macro avg       0.86      0.86      0.86       479
weighted avg       0.86      0.86      0.86       479

training accuracy : 0.8415206121945199
testing accuracy : 0.860125260960334
```

그림 24. [1~17, 49~68]를 기반으로 한 KNN 결과 (참여자 10명 train, 1명 test)

새로운 참여자 데이터로 테스트한 결과도 약 82%의 테스트 정확도를 나타낸다. 아래 그림에서 랜드마크를 모두 활용한 경우에 테스트 데이터가 86%로 더 높게 나오는 것을 확인할 수 있다. 또한, 새로운 참여자 데이터를 테스트로 했기 때문에, 동일한 데이터가 학습과 테스트에 동시에 사용되지 않아서 정상적으로 결과가 나타난다.

## 3-3-(5) Decision Tree, Random Forest, Bagging을 이용한 분류




Decision Tree, Random Forest, Bagging 3가지 방법으로 결과를 예측해보았다. 이 방법들은 모두 랜드마크의 일부를 사용하는 것보다 전체를 사용하는 것이 성능이 더 좋았다. 이에 아래에서는 랜드마크 전체를 기반으로 한 학습에 대한 결과를 출력하였다. 또한, SVM, KNN과 같이 train_test_split을 이용한 학습과 테스트, 그리고 새로운 참여자 데이터를 테스트로 하는 학습과 테스트 두 가지를 모두 진행한 결과 후자의 성능이 더 좋아서 이에 대한 결과만 출력하려고 한다.

### 1) Decision Tree

Decision Tree는 트리 구조로 각 노드에서 분류 기준을 통해 분류해나가면 leaves 가 라벨 데이터가 되는 형식이다. parent node를 후보 질문을 통해 분기하여 children node를 만들고, information gain을 계산하여 가장 높은 information gain을 나타내는 질문을 선택하여 분기한다. 이를 impurity가 0이 되거나 노드 샘플 수가 threshold 이하가 될 때까지 반복한다.

```
print('--------------- DECISION TREE -------------------')
tree_clf = DecisionTreeClassifier(random_state=42)
tree_clf.fit(training_data, training_labels)
predict_tree = tree_clf.predict(validation_data)
ac_score2 =metrics.accuracy_score(validation_labels, predict_tree)
cl_report2 =metrics.classification_report(validation_labels, predict_tree)
print("\n-----report-----\n", cl_report)
print('training accuracy : ', end="")
print(tree_clf.score(training_data, training_labels))
print('testing accuracy : ', end="")
print(tree_clf.score(validation_data, validation_labels))
```

sklearn.tree 라이브러리의 DecisionTreeClassifier를 이용하여 학습시켜 분류해보았다. random_state를 42로 정한 DecisionTreeClassifier를 tree_clf로 만들고, 이를 이용하여 학습 데이터로 학습 시킨 후 테스트 데이터를 예측하여 정확도를 출력해보았다. 약 76%의 정확도를 나타냈다.

```
--------------- DECISION TREE -------------------

-----report-----
              precision    recall  f1-score   support

           0       0.73      0.82      0.77       215
           1       0.84      0.75      0.79       264

    accuracy                           0.78       479
   macro avg       0.78      0.79      0.78       479
weighted avg       0.79      0.78      0.78       479

training accuracy : 1.0
testing accuracy : 0.7620041753653445
```

## 2) Bagging

Decision tree는 variance가 크다는 약점을 가지고 있었다. Variance가 크면 데이터 셋에 따라 모델이 심하게 변동한다. Bagging, 즉 Bootstrap aggregation은 전체 train data를 sampling 해서 새로운 데이터를 만들어내어 Variance를 줄여주기 위하여 사용한다. random sampling을 통해 추출한 여러 개의 subset data를 생성하여 모델링한 후 결합한다.

```
print('------------------ BAGGING --------------------')
bag_clf = BaggingClassifier(DecisionTreeClassifier(), n_estimators=100,
                            bootstrap=True, n_jobs=-1, oob_score=True)
bag_clf.fit(training_data, training_labels)

predict = bag_clf.predict(validation_data)
ac_score=metrics.accuracy_score(validation_labels, predict)
cl_report=metrics.classification_report(validation_labels, predict)
print("\n-----report-----\n", cl_report)
print('training accuracy : ', end="")
print(bag_clf.score(training_data, training_labels))
print('testing accuracy : ', end="")
print(bag_clf.score(validation_data, validation_labels))
```

sklearn.ensemble 라이브러리의 BaggingClassfier를 이용하여 학습시켜 분류해보았다. Bagging을 하기 위해서 sampling할 base_estimator가 있어야 하는데, 이를 DecisionTreeClassifier로 한다. n_estimators=100, 즉 100개의 모형을 만드는 방식을 수행한다. BaggingeClassifier를 bag_clf에 담고, 이를 이용하여 학습 데이터로 학습 시킨 후 테스트 데이터를 예측하여 정확도를 출력해보았다. 약 78%의 정확도를 나타냈다. Decision Tree의 단점을 개선한 모델이기 때문에 Decision Tree보다는 정확도가 높게 나오는 것을 확인할 수 있다.

```
---------- BAGGING ----------

-----report-----
              precision    recall  f1-score   support

           0       0.73      0.82      0.77       215
           1       0.84      0.75      0.79       264

    accuracy                           0.78       479
   macro avg       0.78      0.79      0.78       479
weighted avg       0.79      0.78      0.78       479

training accuracy : 1.0
testing accuracy : 0.7828810020876826
```

### 3) Random Forest

Random Forest는 Decision Tree를 기반으로 하는 앙상블 모델이다. 앙상블이란 학습 모델을 여러 개 사용해서 종합적으로 결론을 내리는 방식이다. Decision tree는 학습 데이터가 조금만 바뀌어도 decision tree가 완전히 달라질 수 있다는 단점이 있다. 이를 random forest를 통하여 combine 하여 일반화할 수 있다.

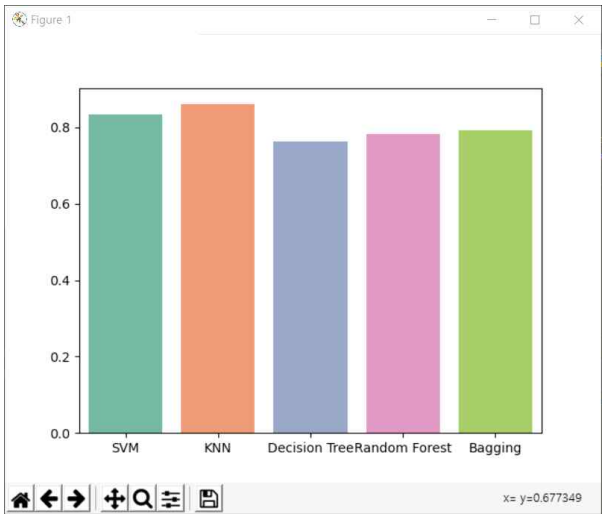
```python
print('---------- RANDOM FOREST ----------')
rand_clf = RandomForestClassifier(random_state=42)
rand_clf.fit(training_data, training_labels)
predict_rand = rand_clf.predict(validation_data)
ac_score3 =metrics.accuracy_score(validation_labels, predict_rand)
cl_report3 =metrics.classification_report(validation_labels, predict_rand)
print("\n-----report-----\n", cl_report)
print('training accuracy : ', end="")
print(rand_clf.score(training_data, training_labels))
print('testing accuracy : ', end="")
print(rand_clf.score(validation_data, validation_labels))
```

sklearn.ensemble 라이브러리의 RandomForestClassfier를 이용하여 학습시켜 분류해보았다. random_state를 42로 정한 RandomForestClassfier를 rand_clf에 담고, 이를 이용하여 학습 데이터로 학습시킨 후 테스트 데이터를 예측하여 정확도를 출력해보았다. 약 79%의 정확도를 나타냈다. Decision Tree의 단점을 개선한 모델이기 때문에 Decision Tree보다는 정확도가 높게 나오는 것을 확인할 수 있다.

```
---------- RANDOM FOREST ----------

-----report-----
              precision    recall  f1-score   support

           0       0.73      0.82      0.77       215
           1       0.84      0.75      0.79       264

    accuracy                           0.78       479
   macro avg       0.78      0.79      0.78       479
weighted avg       0.79      0.78      0.78       479

training accuracy : 1.0
testing accuracy : 0.7933194154488518
```

## 3. 결론

### (1) 최종 종합 그래프




KNN을 이용한 분류가 성능이 가장 좋게 나타났고, SVM Bagging이 뒤를 이었다. 이 결과는 id1을 테스트 데이터로 했을 때의 결과이고, 다른 참여자를 테스터 데이터로 했을 때의 결과는 약간의 차이가 있었다.

### (2) Future Works

- RetinaFace라는 얼굴 검출 기법은 얼굴의 전체를 덮는 형태의 랜드마크를 추출한다고 한다. [17] 이를 통해서 얼굴 전체를 뒤덮는 랜드마크 활용하면 보다 정확한 움직임을 얻을 수 있을 것으로 기대한다. 이를 활용하면 좀더 높은 정확도를 얻을 수 있을 것이라고 판단된다.
- 남성 1명 여성 10명으로 여성의 데이터가 많다. 또한, 좀더 많은 참여자 데이터가 있을 때 정확도가 올라갈 것으로 예상된다.
- Dlib을 통한 얼굴의 랜드마크 추출이 예상보다 정확하게 나타나지 않아서 좌표값이나 움직임 양으로 좌측 저작과 우측 저작을 구별하기가 어려웠다. Dlib의 랜드마크가 좀더 정확하게 추출된다면 다른 feature을 이용할 수 있을 것으로 예상된다.
- 균형 있게 씹는 데이터를 추가한다면 3단계의 분류가 가능해서 균형 있게 씹는 것까지 예측할 수 있을 것이다. 또한 unsupervised clustering을 통해 특정 클래스로 분류될 확률에 대해서 나타낸다면 이분법적인 결과가 아닌 확률로 나타낸 결과를 얻을 수 있을 것으로 예상된다.
- 편측 저작 환자의 데이터를 구할 수 없어서 일반인을 대상으로 좌측과 우측으로 의도적으로 음식을 씹도록 요구하였다. 편측 저작 환자의 데이터가 있다면 이를 바탕으로 같은 과정을 진행한다면 편측 저작 여부에 대해서 좀더 정확한 결과가 나타날 것으로 판단된다.

## 4. Reference

[1] Pond LH, Barghi N, Barnwell GM. Occlusion and chewing side preference. J Prosth et Dent 1986;35:498-500.

[2] Kumai Toshifumi. Difference in chewing patterns between involved and opposite si des in patients with unilateral temporomandibular joint and myofascial pain-dysfunctio

n. Archs oral Biol 1993;38(6):467-478.
[3] Mi-Soon Lee, Influence of Unilateral Mastication Habit on Alveolar Bone Density,
[4] Takahashi M, Takahashi F, Morita O. Evaluation of the masticatory part and the habitual chewing side by wax cube and bite force measuring system, J Jpn Prosthodont Soc, 2008;52:513-20
[5] A Study On the Maximum Bite Force And Facial Morphology Acoording to Chewing Side Preference, Mi-ra Jeong, Woo-sung Son, Korea. J. Orthod., Vol.25, No.3, 1995.
[6] Relationship between occlusion analysis using the T-scan III® system and oral behavior checklist according to temporomandibular joint disorder in female college students
[7] McNeill C. Management of temporomandibular disorders: concepts and controversies. J Prosthet Dent 1997;77(5):510-22.
[8] Neil DJ, Howell PGT, Computerized Kinesiography in the study of mastication in dentate subjects. J Prosthet Dent 1986; 55: 629-638.
[9] Jemt T, Karlsson S, Computer-analysed movements in three dimensions recorded by light-emitting diodes. A study of methodological errors and of evaluation of chewing behaviour in a group of young adults, Journal of Oral RehabilitationVolume 9, Issue 4, July 1982.
[10] A Study on the Effects of Unilateral Chewing Habit on the Chewing movements, Tae-Yeon Hyun, Kyung-Soo Han, Journal of Wonkwang Dental Research Institute, Vol. 7, No.2, 1997.
[11] D. E. King, "Dlib-ml: A machine learning toolkit," Journal of Machine Learning Research, vol. 10, no. Jul, pp. 1755-1758, 2009.
[12] Y. Sun, X. Wang, and X. Tang, "Deep convolutional network cascade for facial point detection," Proceedings of the IEEE conference on computer vision and pattern recognition, 2013, pp. 3476-3483.
[13] M. Valstar, B. Martinez, X. Binefa, and M. Pantic. Facial point detection using boosted regression and graph models. In Proc. CVPR, 2010
[14] Face++, Face++ [Internet], https://www.faceplusplus.com.
[15] Zhang, K., Zhang, Z., Li, Z., Qiao, Y.: Joint face detection and alignment using multi-task cascaded convolutional networks. arXiv preprint (2016). arXiv:1604.02878
[16] P. B M Thomas, T. Baltrusaitis, "The Cambridge Face Tracker: Accurate, Low Cost Measurement of Head Posture Using Computer Vision and Face Recognition Software", Translational Vision Science and Technology. 5. 10.1167, 2016.
[17] J. Deng, J. Guo, Y. Zhou, J. Yu, I. Kotsia, and S. Zafeiriou. Retinaface: Single-stage dense face localisation in the wild. arXiv preprint arXiv:1905.00641, 2019.
[18] Deng, W., & Wu, R.:Real-Time Driver-Drowsiness Detection System Using Facial Features. IEEE Access. 7, 118727-118738 (2019)
[19] Aggarwal V, Asadi H, Gupta M, Lee JJ, Yu D (2018) Covfefe: a computer vision approach for estimating force exertion. arXiv preprint arXiv:1809.09293