

Motion Sensor Technology

Presented by:

Yuge (Kucina) Li

Chuqian Yin

Chuyu Chen

Ziwen Ding



CONTENTS

01

PROBLEM STATEMENT

Business Cases
Our goals

02

MODELING

Data Cleaning
Exploratory Analysis
Model Selection

03

SERVER

Connection
Server Implementation

04

CLIENT

Client Implementation
Running Demo

05

Future Application Scenarios

Resources



01

Problem Statement

Business Problem:

Based on the analysis from WHO, worldwide obesity has nearly **tripled** since 2000. In 2016, more than **1.9 billion** adults, were overweight, with **650 million** obese ones. **378 million** children became overweight or obese in 2019.

Obesity is preventable!



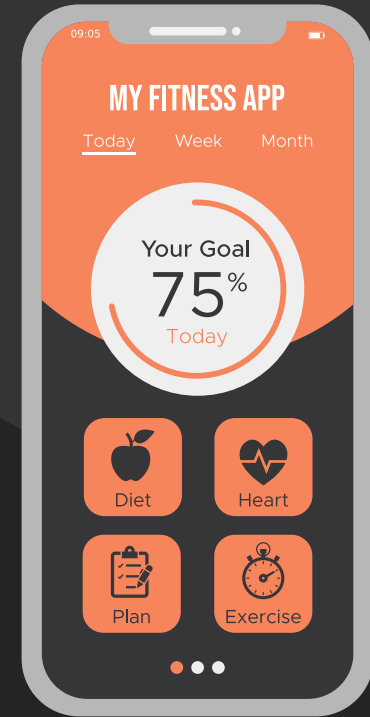
**Motion
Sensor Device**


Goal:

In order to prevent people from getting more weight and help them become healthier, we plan to corporate with local fitness companies and rehabilitation centers to create more personalized fitness apps and launch more customized products and services for their customers and patients.

To achieve the goal, our “**Motion Sensor Device**” could collect users data of exercises or physical activities by:

- detecting and predicting whether the person is “running” or “walking”.
- preventing users from repeating the same action for multiple times by sending health warnings.





02 Modeling

Data Cleaning & Exploratory Analysis:

Nominal Features:

--"wrist": refers to the hand on which the device was worn; 0 for "left" and 1 for "right"

--"activity": refers to the physical activity being performed; 0 for "walk" and 1 for "run"

Ratio Features:

--(x, y, z) acceleration & (x, y, z) gyro(orientation) values

	wrist	activity	acceleration_x	acceleration_y	acceleration_z	gyro_x	gyro_y	gyro_z
count	88588.000000	88588.000000	88588.000000	88588.000000	88588.000000	88588.000000	88588.000000	88588.000000
mean	0.522170	0.500801	-0.074811	-0.562585	-0.313956	0.004160	0.037203	0.022327
std	0.499511	0.500002	1.009299	0.658458	0.486815	1.253423	1.198725	1.914423
min	0.000000	0.000000	-5.350500	-3.299000	-3.753800	-4.430600	-7.464700	-9.480000
25%	0.000000	0.000000	-0.381800	-1.033500	-0.376000	-0.920700	-0.644825	-1.345125
50%	1.000000	1.000000	-0.059500	-0.759100	-0.221000	0.018700	0.039300	0.006900
75%	1.000000	1.000000	0.355500	-0.241775	-0.085900	0.888800	0.733700	1.398200
max	1.000000	1.000000	5.603300	2.668000	1.640300	4.874200	8.498000	11.266200

Model Process:

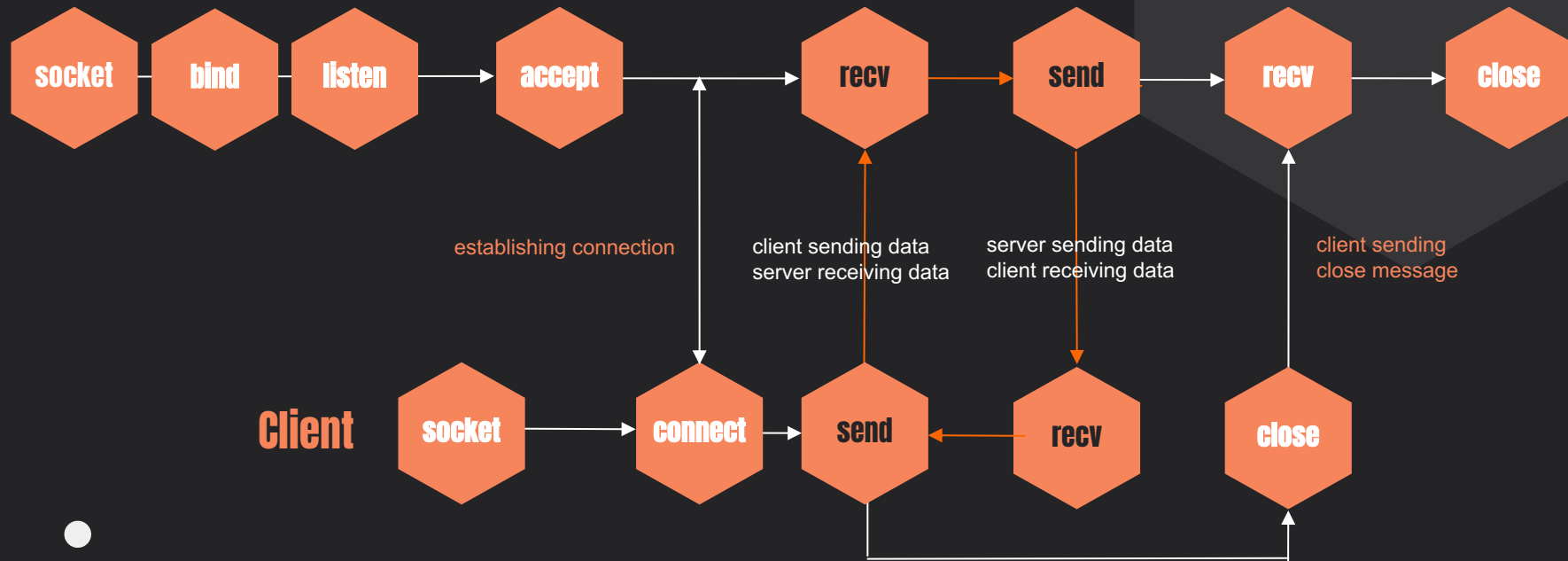
Model	Training Accuracy	Testing Accuracy	F-Measure	Overfitting
SVM	0.9884	0.9880	0.9880	No
Decision Tree	1.0	0.9832	0.9833	No
Random Forest	1.0	0.9900	0.9900	No
Logistic Regression	0.8589	0.8600	0.8541	No
Gradient Boosting	0.9854	0.9836	0.9835	No
Stochastic Gradient Descent	0.8613	0.8609	0.8485	No
Perceptron	0.8499	0.8486	0.8394	No
Naïve Bayesian	0.9561	0.9567	0.9554	No



03 **Server**

Server & Client Connection

Server



Data Exchange



Server



SendCSVfile()
HandleCustomData()
SendStreamToClient()
ListenToClient()
HandleClientAnswer()

Client



CorrectError()
HandleDictPredict()
ActionWarning()

Testing data
Previous result



Prediction

Server Side Implementation

Step 1:

We first load testing dataset from model preparation. Each row in the dataset is loaded into dictionary format like shown on the right.

Step 2:

We store the actual run/walk status into variable **state['label']**. We will use this to check if the prediction returned by client is valid in the future steps.

Server loaded data example

```
{"wrist": "0", "acceleration_x": "0.265", "acceleration_y": "-0.7814",  
"acceleration_z": "-0.0076", "gyro_x": "-0.059", "gyro_y": "0.0325",  
"gyro_z": "-2.9296", "label": "0"}
```

```
def sendCSVfile(self):  
    out = []  
    for f in self.opt.files:  
        print('reading file %s...' % f)  
        csvfile = open(f, 'r')  
        reader = csv.DictReader(csvfile)  
        for row in reader:  
            out += [row]  
    return out
```

handleCustomData Funtion

```
def handleCustomData(self, buffer):  
    if self.opt.mode is not None and self.opt.mode == 'label':  
        self.lock.acquire()  
        self.state['label'] = int(buffer['label'])  
        buffer['label'] = self.check  
        self.lock.release()
```

Server Side Implementation

Step 3:

We encoded the streaming data as JSON format and send it to the client one row by a time. The server will print '*End of stream*' when all data had been sent.

Step 4:

We use the *ListenToClient* function to receive the predictions returned by the client side. At the meantime, we show how many records we correctly predicted and the accuracy rate so far.

```
def sendStreamToClient(self, client, buffer):
    for i in buffer:
        print(i)
        self.handleCustomData(i)
        try:
            client.send((self.convertStringToJSON(i) + '\n').encode('utf-8'))
            time.sleep(self.opt.interval)
        except:
            print('End of stream')
            return False
    client.send((self.convertStringToJSON(self.state) + '\n').encode('utf-8'))
    return False
```

ListenToClient Funtion

```
while True:
    try:
        data = client.recv(size).decode()

        if data:
            a = json.loads(data.rstrip('\n\r '))
            self.handle_client_answer(a)
            total += 1
            print(f"Correctly predicted: {self.state['points']} records.")
            print(f"Accuracy rate: {self.state['points']/total*100}%")
        else:
            print('Client disconnected')
            return False
```

Server Side Implementation

Step 5:

The *check* variable stores if previous prediction is correct/incorrect. The server will send back this information to client on the next stream.

handle_client_answer Function

```
def handle_client_answer(self, obj):  
    if self.opt.mode is not None and self.opt.mode == 'label':  
        if 'label' not in obj:  
            return  
        self.lock.acquire()  
        if self.state['label'] == int(obj['label']):  
            self.state['points'] += 1  
            self.check = 1  
        else:  
            self.check = 0  
        self.lock.release()  
    return
```



04 Client

Client Side Implementation

```
print(cnt)
received = str(sock.recv(1024), "utf-8")
received = json.loads(received)
```

Step 1: Received predictor data as a dictionary

```
def correct_error(indication):
    global correct
    global predict_list
    if indication == 0:
        old = predict_list[-1]
        predict_list[-1] = abs(old-1)
        print('Your previous prediction is wrong')
    else:
        correct += 1
        print('Your previous prediction is correct')
```

```
predict_df = handle_dict_predcit(received)
result = predict_rf(predict_df, rf)
current_action(result)
```

Step 2: Use random forest model to predict current action and print out the result

Step 3: Starting from second received information, client will adjust previous prediction according to server response for previous prediction, and global variable <correct> will count accurate predictions.

Client Side Implementation

```
def action_warning():  
    global predict_list  
    global current_act  
    global times  
    if current_act == predict_list[-1]:  
        times += 1  
    else:  
        current_act = predict_list[-1]  
        times = 1  
    if times >= 5:  
        if current_act == 1:  
            print('Healthy Warning: You run over 5 times, you should try to walk instead.')  
        else:  
            print('Healthy Warning: You walk over 5 times, you should try to run instead.')
```

Step 4: In the meantime, < action_warning > function and global variable <current_act> and <times> will track if user repeats the same action for more than 5 times. If yes, the client will print out the healthy warning message to notify user.

```
if cnt == n-1:  
    print('Prediction Accuracy rate:', (correct / (n-1)) * 100, '%')
```

Step 5: Finally, after finishing the last prediction, client will calculate the accuracy rate of all predictions and print it out. This will be a crucial indication of our model's performance.

Demo

Running Server & Client



05 **Future Application**

Future Application Scenarios:



Monitor

Human performance & measurement application



Home gym

Fitness at Home



IoT & Kits

Body-sensing analytics kits featuring apparel.



Running

Intelligent running exercise and technique coach

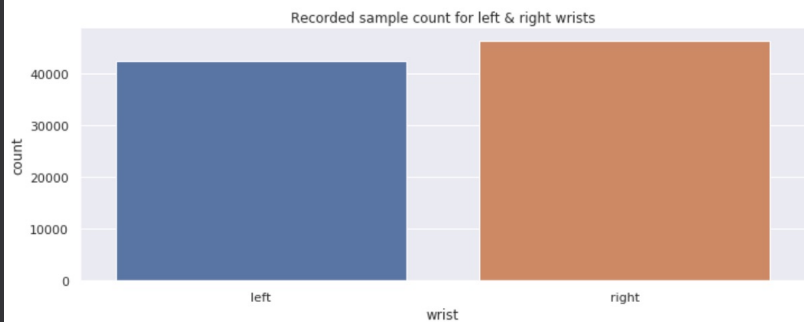
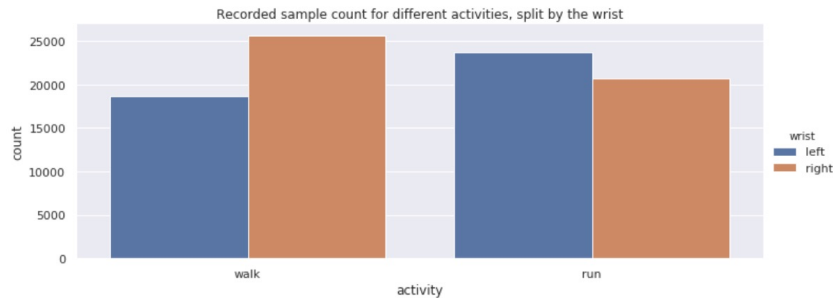
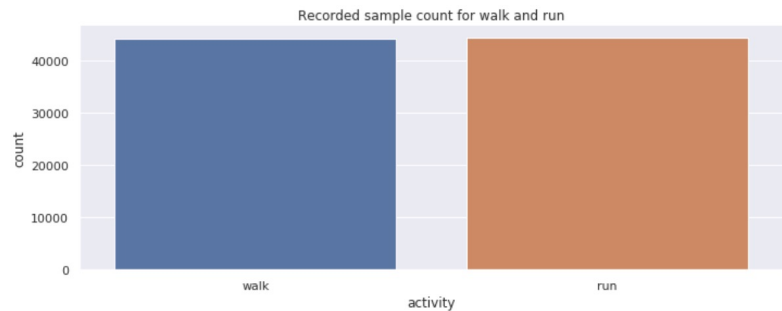
RESOURCES

Data from:

- <https://www.kaggle.com/vmalyi/run-or-walk>

Data Cleaning & Exploratory Analysis:

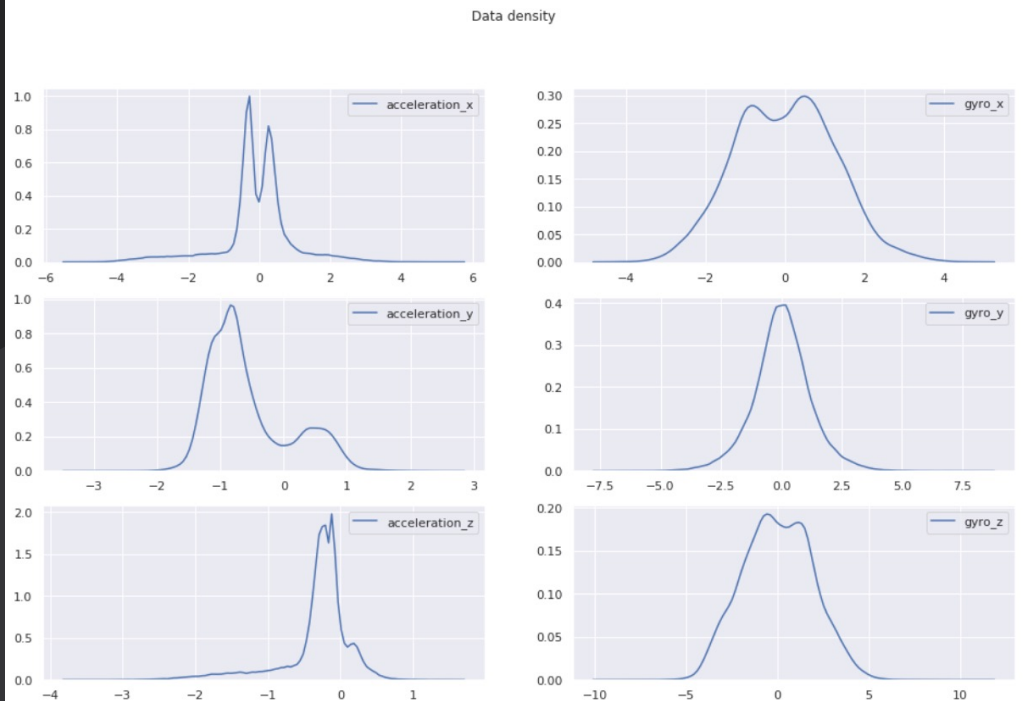
Analysis of activity and wrist features: count plots



- Sample distribution is roughly even for *activity* and *wrist*
- *Walk* have more samples for right wrist and vice versa for *run*

Data Cleaning & Exploratory Analysis:

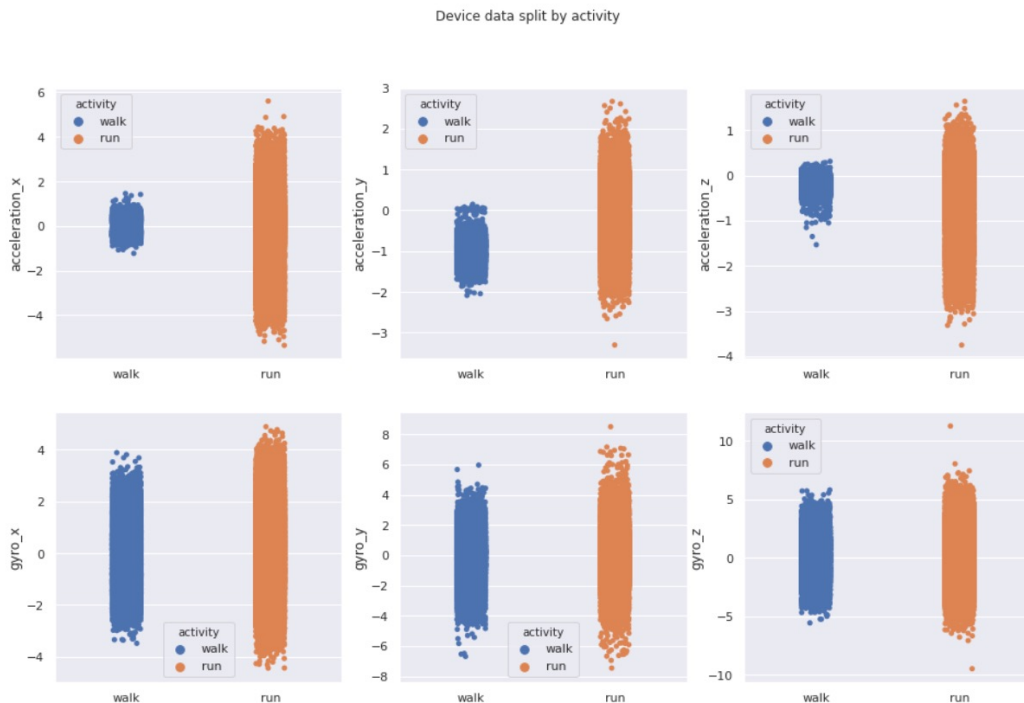
Analysis of accelerometer and gyroscope features: density plots (skewness & inconsistency)



- For **x-axis**, accelerometer data is roughly symmetric and the "wrist" value causes the double peak pattern. Gyroscope data has similar patterns.
- For **y-axis**, gyroscope data has normal distribution with mean = 0. Accelerometer data on the other hand looks skewed, and has the most inconsistent distribution among all the ratio features.
- For **z-axis**, gyroscope data looks symmetric. Accelerometer data is slightly skewed but not as much as y-axis data.

Data Cleaning & Exploratory Analysis:

Analysis of accelerometer and gyroscope features by activity: *walk* & *run*:



- "acceleration_x", "acceleration_z" show clear differentiation between walking and running, with running yielding much higher values.
- "gyroscope" data on the other hand look quite similar for walking and running.



THANKS!

Do you have any questions?

motionsensor@technology.com

+1 666 777 8888

motionsensor.com

