#Импорт.

import pandas #Для работы с файлами.

import random #Для генерации случайный чисел.

import math #Для получения значений минус бесконечности и плюс бесконечности.

import numpy #Для работы со списками.

import matplotlib #Для работы с графиками.

from sklearn.preprocessing import StandardScaler

from sklearn.model\_selection import train\_test\_split #Метод разбиения на обучающую и тестовую выборки.

from sklearn.metrics import classification\_report #Метод оценки точности.

from sklearn.neighbors import KNeighborsClassifier #Модель КНН.

from sklearn.ensemble import RandomForestClassifier #Модель случайного леса.

from sklearn.neural\_network import MLPClassifier #Модель многослойного персептрона.

from sklearn.tree import DecisionTreeClassifier #Модель дерева решений.

print("Блиблиотеки подключены!")

#Чтение данных.

fileServices = pandas.read\_csv("/input/longtermcareservices/longTermCareServices.csv", sep=';') #Перечень услуг СДУ.

fileAssessment = pandas.read\_csv("/input/longtermcareservices/assessmentOfCareNeeds.csv", sep=';') #Оцениваемые действия при определении уровня нуждаемости.

fileLink = pandas.read\_csv("/input/longtermcareservices/assessmentLinkServices.csv", sep=';') #Связка услуг с оценкой.

fileExample = pandas.read\_csv("input/longtermcareservices/example.csv", sep=';') #Примеры принятых решений.

#Констнаты.

SERVICES\_COUNT = len(fileServices) #Количество оказываемых услуг.

ASSESSMENT\_COUNT = len(fileAssessment) #Количество оцениваемых действий.

EXAMPLE\_COUNT = len(fileExample) #Количество примеров принятых решений.

print("Данные загружены! Количество услуг СДУ: {servicesCount}; Количество оцениваемых действий: {assessmentCount}; Количество примеров: {exampleCount}".format(

servicesCount = SERVICES\_COUNT,

assessmentCount = ASSESSMENT\_COUNT,

exampleCount = EXAMPLE\_COUNT

))

#Сбор входных данных.

columnX = [columnName for columnName in (fileExample.columns) if columnName.find("able") >= 0]

dataX = fileExample[columnX]

#Заменяем строковые значения числовыми значениями.

pandas.options.mode.chained\_assignment = None

for i in range(ASSESSMENT\_COUNT):

#dataX[columnX[i]] = dataX[columnX[i]].map({'all' : 0, 'part': 0.5, 'no': 1})

dataX[columnX[i]] = dataX[columnX[i]].map({'all' : 1, 'part': 0.5, 'no': 0})

#Каждую строку примера дублируем количеством услуг.

dataX = dataX.loc[dataX.index.repeat(SERVICES\_COUNT)].reset\_index(drop=True)

dataX["service"] = fileServices["number"].to\_list() \* EXAMPLE\_COUNT # Вставляем индексы

dataX["service"] = [data / SERVICES\_COUNT for data in (dataX["service"])] #Нормализация услуг.

#Сбор выходных данных.

columnY = [columnName for columnName in (fileExample.columns) if columnName.find("need") >= 0]

dataY = []

for i in range(EXAMPLE\_COUNT):

for j in range(SERVICES\_COUNT):

dataY.append(fileExample[columnY[j]][i])

#Демонстрация данных.

dataXY = dataX

dataXY["answer"] = dataY

display(dataXY)

print("Данные загружены и обработаны!")

#Функция получения процента точности из отчета.

def getAccuracy(accuracy):

positionStart = accuracy.find("accuracy") #Начало строки с точностью.

positionStart = accuracy.find("0", positionStart) #Начало значения.

positionEnd = accuracy.find(" ", positionStart) #Конец значения.

return accuracy[positionStart:positionEnd] #Подстрока.

#Получение списка строк с услугами для удаления, не относящихся к перечню оцениваемых действий.

def getRowForDelete(data, iterationIndex):

serviceNeeded = []

rowForDelete = []

if iterationIndex >= 1:

serviceNeeded = serviceNeeded + [0.02, 0.04]

if iterationIndex >= 2:

serviceNeeded = serviceNeeded + [0.06, 0.08]

if iterationIndex >= 3:

serviceNeeded = serviceNeeded + [0.10, 0.12]

if iterationIndex >= 4:

serviceNeeded = serviceNeeded + [0.14, 0.40]

if iterationIndex >= 5:

serviceNeeded = serviceNeeded + [0.38, 0.50, 0.52, 0.54, 0.56]

if iterationIndex >= 6:

serviceNeeded = serviceNeeded + [0.16, 0.18, 0.34, 0.36, 0.42, 0.44, 0.46]

if iterationIndex >= 7:

serviceNeeded = serviceNeeded + [0.20, 0.22, 0.24, 0.26, 0.28, 0.30, 0.32]

if iterationIndex >= 8:

serviceNeeded = serviceNeeded + [0.68, 0.70]

if iterationIndex >= 9:

serviceNeeded = serviceNeeded + [0.62, 0.64, 0.66]

if iterationIndex >= 11:

serviceNeeded = serviceNeeded + [0.58, 0.60, 0.72, 0.74, 0.76, 0.78]

if iterationIndex >= 12:

serviceNeeded = serviceNeeded + [0.80]

if iterationIndex >= 14:

serviceNeeded = serviceNeeded + [0.48]

if iterationIndex >= 16:

serviceNeeded = serviceNeeded + [0.90]

if iterationIndex >= 18:

serviceNeeded = serviceNeeded + [0.86, 0.88]

for i in range(len(data)):

if data["service"][i] not in serviceNeeded:

rowForDelete.append(i)

return rowForDelete

#Результаты тестирования.

accuracyKNN = []

accuracyRF= []

accuracyMLP = []

accuracyDT = []

#Тестирование моделей на разных количествах признаков (оцениваемых действий).

for iterationIndex in range(1, 22):

#Исходные данные только с услугами, относящимися к оцениваемым действиям.

dataFilted = dataXY.copy()

rowForDelete = getRowForDelete(dataFilted, iterationIndex)

dataFilted = dataFilted.drop(rowForDelete)

dataY = dataFilted["answer"]

columnForDelete = dataFilted.columns[iterationIndex:len(dataFilted.columns)].to\_list()

columnForDelete.remove("service")

dataX = dataFilted.drop(columnForDelete, axis = 1)

#Предобработка данных.

scaler = StandardScaler()

scaler.fit(dataX)

scaledFeatures = scaler.transform(dataX)

scaledData = pandas.DataFrame(scaledFeatures, columns = dataX.columns)

x = scaledData

y = dataY

print("X: ",x.shape, "Y:", y.shape)

#Разбиение на обучающую и тестовую выборки.

trainX, testX, trainY, testY = train\_test\_split(x, y, test\_size = 0.3)

#Метод КНН.

modelKNN = KNeighborsClassifier(n\_neighbors = 6)

modelKNN.fit(trainX, trainY)

predictionKNN = modelKNN.predict(testX)

report = classification\_report(testY, predictionKNN)

accuracy = getAccuracy(report)

accuracyKNN.append(float(accuracy))

#Метод случайного леса.

modelRF = RandomForestClassifier()

modelRF.fit(trainX, trainY)

predictionRF = modelRF.predict(testX)

report = classification\_report(testY, predictionRF)

accuracy = getAccuracy(report)

accuracyRF.append(float(accuracy))

#Метод многослойного персептрона.

modelMLP = MLPClassifier(hidden\_layer\_sizes = 5, max\_iter = 5000, activation = "logistic")

modelMLP.fit(trainX, trainY)

predictionMLP = modelMLP.predict(testX)

report = classification\_report(testY, predictionMLP)

accuracy = getAccuracy(report)

accuracyMLP.append(float(accuracy))

#Метод дерева решений.

modelDT = DecisionTreeClassifier(criterion = "gini", max\_depth = 15)

modelDT.fit(trainX, trainY)

predictionDT = modelDT.predict(testX)

report = classification\_report(testY, predictionDT)

accuracy = getAccuracy(report)

accuracyDT.append(float(accuracy))

#Результат

print("КНН:", accuracyKNN)

print("Случайный лес:", accuracyRF)

print("Многослойный персептрон:", accuracyMLP)

print("Дерево решений:", accuracyDT)

#Ген особи.

class Gene:

#Атрибуты.

\_\_value = 0 #Значение.

\_\_valueMin = 0 #Максимально допустимое значение.

\_\_valueMax = 0 #Минимально допустимое значение.

\_\_weight = 0 #Вес гена.

\_\_cost = 0 #Ценность гена.

#Инициализация.

def \_\_init\_\_(self, value = 0, valueMin = -1 \* math.inf, valueMax = math.inf, weight = 0, cost = 0):

if (valueMin > valueMax):

raise ValueError("Минимальное устанавливаемое значение ({valueMin}) больше максимального ({valueMax})!".format(

valueMin = valueMin, valueMax = valueMax)

)

self.\_\_valueMin = valueMin

self.\_\_valueMax = valueMax

self.\_\_weight = weight

self.\_\_cost = cost

self.setValue(value)

#Получение значения.

def getValue(self):

return self.\_\_value

#Получение минимально возможного значения гена.

def getValueMin(self):

return self.\_\_valueMin

#Получение максимально возможного значения гена.

def getValueMax(self):

return self.\_\_valueMax

#Получение веса гена.

def getWeight(self):

return self.\_\_weight

#Получение ценности гена.

def getCost(self):

return self.\_\_cost

#Получение списка всех атрибутов гена.

def getListAttribute(self):

attribute = {'value' : [self.\_\_value],

'valueMin' : [self.\_\_valueMin],

'valueMax' : [self.\_\_valueMax],

'weight' : [self.\_\_weight],

'cost' : [self.\_\_cost]

}

return attribute

#Установка значения.

def setValue(self, value):

if (value < self.\_\_valueMin or value > self.\_\_valueMax):

raise ValueError("Устанавливаемое значение ({value}) не входит в допустимый диапазон ({valueMin} - {valueMax})!".format(

value = value, valueMin = self.\_\_valueMin, valueMax = self.\_\_valueMax)

)

else:

self.\_\_value = value

#Устанвока минимального допустимого значения.

def setValueMin(self, valueMin):

if (self.\_\_value < valueMin):

raise ValueError("Устанавливаемое минимальное значение ({valueMin}) больше текущего значения ({valueSelf})!".format(

valueMin = valueMin, valueSelf = self.\_\_value)

)

else:

self.\_\_valueMin = valueMin

#Устанвока максимально допустимого значения.

def setValueMax(self, valueMax):

if (self.\_\_value > valueMax):

raise ValueError("Устанавливаемое максимальное значение ({valueMax}) меньше текущего значения ({valueSelf})!".format(

valueMax = valueMax, valueSelf = self.\_\_value)

)

else:

self.\_\_valueMax = valueMax

#Установка веса гена.

def setWeight(self, weight):

self.\_\_weight = weight

#Установка ценности гена.

def setCost(self, cost):

self.\_\_cost = cost

#Мутяция гена (генерация другого значения из допустимого диапазона)

def mutateValue(self):

if (self.\_\_valueMin != self.\_\_valueMax):

if (self.\_\_value == self.\_\_valueMin):

self.setValue(random.randint(self.\_\_valueMin + 1,self.\_\_valueMax))

elif (self.\_\_valueMax == self.\_\_value):

self.setValue(random.randint(self.\_\_valueMin,self.\_\_valueMax - 1))

elif (random.random() <= 0.5): #Случайно выбираем выбрать значение из левого или правого диапазона от значения.

self.setValue(random.randint(self.\_\_valueMin, self.\_\_value - 1))

else:

self.setValue(random.randint(self.\_\_value + 1, self.\_\_valueMax))

#Клонирование гена.

def clone(self):

return Gene(self.\_\_value, self.\_\_valueMin, self.\_\_valueMax, self.\_\_weight, self.\_\_cost)

print("Класс гена особи загружен!")

#Класс особи.

class Individ:

#Атрибуты.

\_\_gens = [] #Гены особи.

\_\_fittness = 0 #Приспособленность особи.

\_\_weight = 0 #Вес особи.

\_\_weightMin = 0 #Минимально допустимое значение веса особи для приспособленности.

\_\_weightMax = 0 #Максимально допустимое значение веса особи для приспособленности.

#Инициализация.

def \_\_init\_\_(self, gens, weightMin = -1 \* math.inf, weightMax = math.inf):

self.\_\_gens = gens

self.\_\_weightMin = weightMin

self.\_\_weightMax = weightMax

self.calculateFitness()

#Получение количество генов.

def getCountGens(self):

return len(self.\_\_gens)

#Получение списка значений генов особи.

def getListValueGens(self):

gens = []

for iGene in range(self.getCountGens()):

gens.append(self.\_\_gens[iGene].getValue())

return gens

#Получение списка полностью клонированных генов особи.

#Гены клонируются для того, чтобы они не изменялись вне особи.

#Так как если возвращать просто список генов, то они передадутся по ссылке.

def getListClonedGens(self):

gens = []

for iGene in range(self.getCountGens()):

gens.append(self.\_\_gens[iGene].clone())

return gens

#Получение приспособленности особи.

def getFittness(self):

return self.\_\_fittness

#Получение веса особи.

def getWeight(self):

return self.\_\_weight

#Получение минимально допустимого значения веса особи для приспособленности.

def getWeightMin(self):

return self.\_\_weightMin

#Получение максимально допустимого значения веса особи для приспособленности.

def getWeightMax(self):

return self.\_\_weightMax

#Установка минимально допустимного значения веса особи для приспособленности.

def setWeightMin(self, weightMin):

self.\_\_weightMin = weightMin

self.calculateFitness()

#Установка максимально допустимного значения веса особи для приспособленности.

def setWeightMax(self, weightMax):

self.\_\_weightMax = weightMax

self.calculateFitness()

#Вычисление приспособленности особи (особь, веса генов, стоимость генов, максимальное допустимый вес особи)

#Приспособленность высчитывается как сумма произведений значения гена на его стоимость.

#При этом, если сумма произведений значений генов на их веса будут превышать допустимый вес, то особь отбрасывается.

def calculateFitness(self):

costSum, weightSum = 0, 0

for iGene in range(0, self.getCountGens(), 2):

costSum = costSum + (self.\_\_gens[iGene].getValue() \* self.\_\_gens[iGene + 1].getValue()) \* self.\_\_gens[iGene + 1].getCost()

weightSum = weightSum + (self.\_\_gens[iGene].getValue() \* self.\_\_gens[iGene + 1].getValue()) \* self.\_\_gens[iGene + 1].getWeight()

#for iGene in range(self.getCountGens()):

#costSum = costSum + self.\_\_gens[iGene].getValue() \* self.\_\_gens[iGene].getCost()

#weightSum = weightSum + self.\_\_gens[iGene].getValue() \* self.\_\_gens[iGene].getWeight()

if weightSum < self.\_\_weightMin or weightSum > self.\_\_weightMax:

self.\_\_fittness = 0

else:

self.\_\_fittness = costSum

self.\_\_weight = weightSum

#Скрещивание особи с другой особью (Другая особь для скрещивания).

def cross(self, individAnother):

if (individAnother.getCountGens() != self.getCountGens()):

raise ValueError("У скрещиваемых особей разное количество генов!")

gensChild1 = self.getListClonedGens()

gensChild2 = individAnother.getListClonedGens()

startCut = random.randint(1, self.getCountGens()/2 - 2) \* 2

#startCut = random.randint(1, self.getCountGens() - 2)

for iGene in range(startCut, self.getCountGens()):

tempGeneChild1 = gensChild1[iGene].getValue()

gensChild1[iGene].setValue(gensChild2[iGene].getValue())

gensChild2[iGene].setValue(tempGeneChild1)

return Individ(gensChild1, self.\_\_weightMin, self.\_\_weightMax), Individ(gensChild2, self.\_\_weightMin, self.\_\_weightMax)

#Мутировать особь путем изменения значения одного гена.

def mutate(self):

iGene = random.randint(0, self.getCountGens() - 1)

self.\_\_gens[iGene].mutateValue()

self.calculateFitness()

#Клонирование особи.

def clone(self):

gensClone = []

for iGene in range(self.getCountGens()):

gensClone.append(self.\_\_gens[iGene])

return Individ(gensClone, self.\_\_weightMin, self.\_\_weightMax)

print("Класс особи загружен!")

#Генерация особи (интервал возможных значений для каждого гена).

#Особь генерируется на оснвое шаблона, который содержит интервал допустимых значений для каждого гена.

def generateIndivid(individPattern, weightMin = -1 \* math.inf, weightMax = math.inf):

gensCount = len(individPattern)

gens = []

for iGene in range(gensCount):

valueMin = individPattern["valueMin"][iGene]

valueMax = individPattern["valueMax"][iGene]

weight = individPattern["weight"][iGene]

cost = individPattern["cost"][iGene]

gene = Gene(valueMin, valueMin, valueMax, weight, cost)

gene.mutateValue()

gens.append(gene)

individ = Individ(gens, weightMin, weightMax)

return individ

#Генерация популяции (размер популяции, интервал возможных значений для каждого гена).

#Популяция генерируется на оснвое шаблона особей, который содержит интервал допустимых значений для каждого гена особи.

def generatePopulation(populationSize, individPattern, weightMin = -1 \* math.inf, weightMax = math.inf):

population = []

for iIndivid in range(populationSize):

population.append(generateIndivid(individPattern, weightMin, weightMax))

return population

#Отбор особей для скрещивания (популяция).

def chooseIndividForCross(population):

individCount = len(population)

choosedIndivids = []

for iIndivid in range(individCount):

individInTour = []

for iContender in range(0, 3):

individInTour.append(population[random.randint(0, individCount - 1)])

individMax = max(individInTour, key = lambda x: x.getFittness())

choosedIndivids.append(individMax)

return choosedIndivids

#Скрещивание особей (популяция, вероятность скрещивания).

def crossIndivid(population, crossPercent):

populationNew = []

random.shuffle(population)

for iIndivid in range(0, len(population), 2):

if iIndivid + 1 == len(population):

populationNew.append(population[iIndivid])

elif crossPercent >= random.random():

child1, child2 = population[iIndivid].cross(population[iIndivid + 1])

populationNew.append(child1)

populationNew.append(child2)

else:

populationNew.append(population[iIndivid])

populationNew.append(population[iIndivid + 1])

return populationNew

#Мутация особей (популяция, вероятность мутации)

def mutateIndivid(population, mutatePercent):

for iIndivid in range(len(population)):

if mutatePercent >= random.random():

population[iIndivid].mutate()

#Генетический алгоритм (размер популяции, шаблон особи, количество поколений, минимальное и максимальное допустимое значение весов)

def geneticAlgorithm(populationSize, individPattern, numberGeneration, Tmin, Tmax):

fitnessHistory = []

populationNew = generatePopulation(populationSize, individPattern, Tmin, Tmax)

fitnessHistory.append([individ.getFittness() for individ in populationNew])

for iGeneration in range(numberGeneration):

individForCross = chooseIndividForCross(populationNew)

populationNew = crossIndivid(individForCross, mutatePercent)

mutateIndivid(populationNew, crossPercent)

fitnessHistory.append([individ.getFittness() for individ in populationNew])

populationNew = sorted(populationNew, key = lambda x: x.getFittness(), reverse = 1)

return populationNew, fitnessHistory

print("Методы загружены!")

#display(fileExample.iloc[22][:]) #1 уровень 111854.

#display(fileExample.iloc[136][:]) #2 уровень 112374.

#display(fileExample.iloc[39][:]) #3 уровень 111914.

#Сбор данных примера.

iExample = 39

assessment = [] #Оценка нуждаемости.

need, day, week = [], [], [] #Решение эксперта.

for iColumn in range(len(fileExample.columns)):

if (fileExample.columns[iColumn].find("able") >= 0):

assessment.append(fileExample.iloc[iExample][iColumn])

elif (fileExample.columns[iColumn].find("need") >= 0):

need.append(fileExample.iloc[iExample][iColumn])

elif (fileExample.columns[iColumn].find("day") >= 0):

day.append(fileExample.iloc[iExample][iColumn])

elif (fileExample.columns[iColumn].find("week") >= 0):

week.append(fileExample.iloc[iExample][iColumn])

expertDecision = pandas.DataFrame({

"need" : need,

"countInDay": day,

"countDaysInWeek": week

})

print(assessment)

#Формирование перечня услуг (по эксперту)

serviceIndex = []

serviceCost = []

for iService in range(len(expertDecision)):

if expertDecision["need"][iService] == 1:

serviceIndex.append(iService)

serviceIndex.append(iService)

serviceCost.append(fileServices["t"][iService])

serviceCost.append(fileServices["t"][iService])

serviceName = []

serviceValueMin = []

serviceValueMax = []

serviceWeight = []

for iService in range(len(serviceIndex)):

serviceName.append(fileServices["name"][serviceIndex[iService]])

serviceWeight.append(fileServices['t'][serviceIndex[iService]])

if (iService % 2) == 0:

serviceValueMin.append(fileServices["minCountInDay"][serviceIndex[iService]])

serviceValueMax.append(fileServices["maxCountInDay"][serviceIndex[iService]])

else:

serviceValueMin.append(fileServices["minDaysInWeek"][serviceIndex[iService]])

if fileServices["maxDaysInWeek"][serviceIndex[iService]] > 5:

serviceValueMax.append(5)

else:

serviceValueMax.append(fileServices["maxDaysInWeek"][serviceIndex[iService]])

individPattern = pandas.DataFrame({

"number": serviceIndex[:],

"serviceName": serviceName[:],

"valueMin" : serviceValueMin[:],

"valueMax" : serviceValueMax[:],

"weight" : serviceWeight[:],

"cost": serviceCost[:]}

)

populationSize = 50

numberGeneration = 100

Tmin = 0

Tmax = 28\*60 #14\*60 21\*60 28\*60

mutatePercent = 0.3

crossPercent = 0.7

#display(individPattern)

#Инициализация истории поколений.

fitness\_history = []

#Генерация начальной популяции.

populationNew = generatePopulation(populationSize, individPattern, Tmin, Tmax)

fitness\_history.append([individ.getFittness() for individ in populationNew])

#Основной алгоритм.

for iGeneration in range(numberGeneration):

#Отбор.

individForCross = chooseIndividForCross(populationNew)

#Скрещивание.

populationNew = crossIndivid(individForCross, mutatePercent)

#Мутация.

mutateIndivid(populationNew, crossPercent)

fitness\_history.append([individ.getFittness() for individ in populationNew])

#Сортировка последней популяции.

populationNew = sorted(populationNew, key = lambda x: x.getFittness(), reverse = 1)

#График лучших особей каждого поколения.

fitness\_history\_mean = [numpy.mean(fitness) for fitness in fitness\_history]

fitness\_history\_max = [numpy.max(fitness) for fitness in fitness\_history]

matplotlib.pyplot.plot(list(range(numberGeneration + 1)), fitness\_history\_mean, label = 'Mean Fitness')

matplotlib.pyplot.plot(list(range(numberGeneration + 1)), fitness\_history\_max, label = 'Max Fitness')

matplotlib.pyplot.legend()

matplotlib.pyplot.title('Fitness through the generations')

matplotlib.pyplot.xlabel('Generations')

matplotlib.pyplot.ylabel('Fitness')

matplotlib.pyplot.show()

#Извлечение информации из самой лучшей особи.

gens = populationNew[0].getListValueGens()

algorithmResult = pandas.DataFrame({

"index" : individPattern["number"].unique(), #К одной услуге относится два гена, из-за чего индексы повторяются, поэтому берем уникальные.

"countInDay" : gens[0::2], #Количество услуг в день находится по нечетным.

"countDaysInWeek": gens[1::2] #Количество дней в неделю находится по четным.

}

)

#Формируем итоговый результат.

result = fileServices[["number", "name", "t"]]

result["expertCountInDay"] = expertDecision["countInDay"]

result["expertcountDaysInWeek"] = expertDecision["countDaysInWeek"]

result.insert(len(result.columns), "countInDay", 0)

result.insert(len(result.columns), "countDaysInWeek", 0)

result.insert(len(result.columns), "countInWeek", 0)

result.insert(len(result.columns), "timeInWeek", 0)

#Заполняем итоговый результат информацией, извлеченной из особи.

for iService in range(len(algorithmResult)):

#Формируем данные.

index = algorithmResult["index"][iService]

countInDay = algorithmResult["countInDay"][iService]

countDaysInWeek = algorithmResult["countDaysInWeek"][iService]

countInWeek = countInDay \* countDaysInWeek

#Вставляем данные.

result.at[index, "countInDay"] = countInDay

result.at[index, "countDaysInWeek"] = countDaysInWeek

result.at[index, "countInWeek"] = countInWeek

result.at[index, "timeInWeek"] = countInWeek \* result["t"][index]

#Демонстрация результата.

display(result)

print("Итого: {totalCount} услуг ({totalHourse} ч. {totalMinute} мин.).".format(

totalCount = result["countInWeek"].sum(),

totalHourse = round(result["timeInWeek"].sum() // 60, 0),

totalMinute = result["timeInWeek"].sum() % 60

))