**빅데이터응용보안 팀프로젝트 보고서**

기계학습을 활용한 차량 주행 데이터기반 도난방지



사이버보안학과 201320878 박용채

사이버보안학과 201321493 이정관

사이버보안학과 201320926 유진형

사이버보안학과 201320921 조윤상

**목차**

1. 주제 선정

1.1. 선정 배경

1.2. 프로젝트 목표

1.3. 기대효과

1.4. 관련연구

1.5. 프로젝트 진행

1.5.1. 프로세스

1.5.2. 개발 환경

1.5.3. 팀원 역할

2. 데이터셋

2.1. 데이터셋 선정

2.2. 데이터셋 구조

3. 전처리

3.1. 전처리 과정

3.2. 처리 개발

3.3. 전처리 결과

4. 기계학습

4.1. 알고리즘 선정

4.2. 기계학습 개발

4.3. 정확도 측정

5. Simulation

5.1. Methodology Flowchart

5.2. Simulation 구현

5.3. Simulation 결과

6. 참고문헌

**1. 주제 선정**

**1.1. 선정 배경**

스마트 키와 각종 경보 장치에 자동차의 현재 위치까지 실시간으로 확인 가능한 기능도 갖출 수 있는 지금 시대에 ‘자동차 도난’ 이란 단어가 멀게만 느껴진다. 하지만 이런 기능들을 오히려 역으로 이용한 자동차 도난 사건이 생각보다 빈번하게 일어나고 있다. 오늘날 ICT가 발전하면서 차량도 많은 혜택을 받고 있다. 자동차도 인터넷에 연결되어 점점 더 다양한 기능을 제공해 사용자에게 많은 편의성을 주고 있다. 하지만 이에 따라 차량의 원거리 및 직접 접근을 통해 도난이 일어나는 경우도 증가하고 있다. 사이버 공격이 다양하고 지능화됨에 따라 단순한 보호 기술만으로는 대응할 수 없다. 새로운 공격을 탐지하기 위해 데이터 마이닝 기술이 사용되는데 이 기술은 운전자의 운전 습관을 이용한다. 운전자가 차량 간격, 최대 주행 속도, 의도하지 않은 급가속 등의 운전 습관을 가지고 있어 고유한 주행패턴을 나타내기 때문에 이를 활용하면 차량도난을 탐지하는데 유용하다.

**1.2. 프로젝트 목표**

차량에 임베디드된 다양한 센서 데이터들의 실제 수집된 운전 데이터셋을 전처리하여 가공하고 기계학습을 적용하여 학습 데이터셋를 통한 Training과 테스트 데이터셋를 통한 Testing을 거친다. 이 때 사용되는 알고리즘은 KNN, Decision Tree, SVM이다. 마지막으로 기계학습에서 나온 정확도를 기준으로 Arena로 Simulation을 수행하여 기계학습 알고리즘을 통한 운전자 식별 시나리오를 작성 및 평가한다.

**1.3. 기대효과**

본 프로젝트는 차량에 탑재된 센서를 통해 운전자의 주행 데이터를 수집하고 기계학습을 기반으로 운전자를 식별한다. 이를 통해 키를 습득하거나 차량의 전자장비의 취약점을 악용하여 차량을 탈취하더라도 운전자의 주행 습관을 바탕으로 현재 주행중인 운전자를 식별하여 차량 도난을 탐지 및 예방 할 수 있다.

**1.4. 관련연구**

**1) Kwak, Byung Il, JiYoung Woo, and Huy Kang Kim. "Know your master: Driver profiling-based anti-theft method."**

내용

차량 도난을 방지하기 위한 새로운 기술이 개발됨에도 차량에 들어가는 전자장비의 취약점을 악용하는 새로운 도난 방법이 나오고 있다. 본 논문에서는 기계학습과 차량의 센서를 활용하는 새로운 도난 탐지 기법을 소개하고 있다. 차량 센서를 통해 축적한 데이터를 바탕으로 기계학습을 진행하고 이를 활용하여 운전자를 탐지한다.

차이점

단순히 기계학습의 정확도와 정밀도 향상을 위한 연구뿐 아니라 시뮬레이션을 활용해 사용자를 탐지하는 일련의 과정을 수행한다.

**2) Virojboonkiate, Nuttun, Peerapon Vateekul, and Kultida Rojviboonchai. "Driver identification using histogram and neural network from acceleration data."**

내용

최근 센서 기술이 좋아지고 가격이 저렴해지면서 차량에 다양한 센서가 장착되고 있다. 이 중 acceleration 센서가 존재한다. 본 연구에서는 악셀레이터의 센서 값과 기계학습을 통해 운전자를 구별하기위한 방법을 제시한다.

차이점

차량에 51개의 센서를 부착한 후 51개의 센서 중 운전자의 행동 패턴을 분석하기 위해 사용할 수 있는 패턴을 선별하여 기계 학습에 활용하였다**.**

**1.5. 프로젝트 진행**

**1.5.1. 프로세스**

프로젝트를 진행하는 프로세스는 크게 전처리, 기계학습, 시뮬레이션의 3단계로 나눠 진행한다.

1) 전처리

전처리에서는 분석 및 분류 알고리즘에 적용 가능하도록 가공하는 단계이다. 이 과정에서는 전체 주행 데이터셋에서 필요한 속성들을 선택하고 추출하고 슬라이딩 윈도우를 적용하여 신뢰성 있는 탐지를 제공하여 가능한 빨리 소유주한테 알릴 수 있게 한다. 또한 평균, 중간, 표준편차와 같은 통계적 특징도 도출하여 집합을 다양하게 하여 신뢰할 수 있는 모델을 생성한다.

2) 기계학습

기계학습 단계에서는 K-nearest neighbors algorithm(KNN), Decision Tree, Random Forest, Support vector machine(SVM) 4개의 알고리즘에 가공된 데이터를 적용하여 운전자에 대한 분류 정확도를 표현하고 운전자 인증 모델을 생성한다. 분류에 대한 성능평가는 10-fold cross validation을 적용하여 수행하였다.

3) 시뮬레이션

시뮬레이션 단계에서는 기계학습 단계에서 사용한 3개의 알고리즘의 정확도를 기반으로 실제 차량 주행 시에 수행되는 운전자 식별 시나리오를 수행한다. 처음 주행 데이터를 받아와 3개의 알고리즘 중 정확도가 높은 순서대로 알고리즘을 직렬로 배치하여 검증을 수행하고 이 3개에 대한 알고리즘에 대해 모두 식별이 완료가 되는 경우에 운전자 검증이 완료된다.

**1.5.2.**  **개발 환경**

1) Python

Python 은 라이브러리가 풍부하고 객체 지향 프로그래밍과 구조적 프로그래밍을 완벽하게 지원하며 함수형 프로그래밍, 관점 지향 프로그래밍 등도 주요 기능에서 지원된다. 또한 타언어에 비해 문법이 쉬우며 제작자가 의도적으로 간결하게 만들어 읽고 사용하기 쉽다는 특징이 있다. Python 은 시스템 유틸리티 제작, GUI 프로그래밍, 웹 프로그래밍, 데이터베이스 프로그래밍 등 다양한 기능을 지원하지만 속도가 느리고 모바일 컴퓨팅에는 취약하다는 단점이 있다.

본 프로젝트에서는 데이터 전처리, 기계학습 알고리즘 등의 개발에 Python을 활용한다.

2) GitLab

GitLab 은 Git의 원격 저장소 기능과 이슈 트래커 기능등을 제공하는 소프트웨어다. 설치형 Github라는 컨셉으로 시작된 프로젝트이기 때문에 Github와 비슷한 면이 많이 있다. 서비스 형 원격저장소를 운영하는 것에 대한 비용이 부담되거나, 소스코드의 보안이 중요한 프로젝트에게 적당하다.

본 프로젝트에서는 데이터 전처리, 기계학습 알고리즘, 시뮬레이션 등의 개발에서 사용되는 소스코드 및 파일을 GitLab에서 활용한다.

3) Arena Simulation Software

Arena 는 프로젝트 수행을 위해서 다양한 기능들을 가지고 있다. 입력 데이터의 통계적인 분석, 모델 구축을 위한 구성 요소들의 제공, 프로세스의 처리와 다른 작업과 연관된 시스템을 보여주는 애니메이션 기능, 의사 결정을 위한 시뮬레이션 실행 결과의 통계적인 비교를 도와주는 출력 분석 도구, 여러 가지 대안들 중 최적의 대안을 찾아주는 최적화 기능 등을 가지고 있다.

본 프로젝트에서는 기계학습 알고리즘을 통한 운전자 식별 시나리오에 활용된다.

**1.5.3. 팀원 역할**

박용채: 기계학습 알고리즘 활용 정확도 및 정밀성 측정

유진형: 전처리 계획 수립 및 개발

이정관: 시뮬레이션 구성도 작성 및 실행

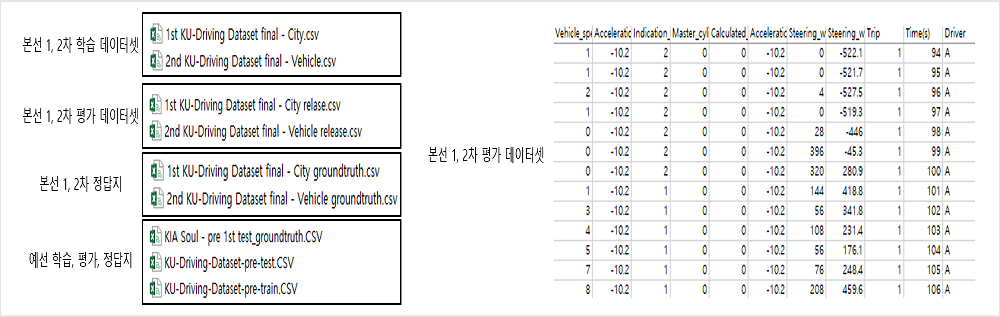
조윤상: 전처리 계획 수립 및 개발

**2. 데이터셋**

**2.1. 데이터셋 수집**

사용된 데이터는 고려대학교 해킹대응기술연구실에서 연구한 차량 주행 데이터를 이용한 운전자 프로파일링의 성능검증에 기반이 된 데이터셋으로, Kia Soul 및 Hyundai Sonata 두 개의 차량으로 두 개 코스에 대해서 예선 10명, 본선 5명에 대해서 데이터를 수집했으며, 예선에서 사용한 주행 데이터는 610km이며, 본선에서 사용한 주행데이터는 약 950km의 데이터를 사용하였다. 모든 주행 데이터 파일은 주행한 후 운전자의 라벨링을 수행했으며, 운전자들의 주행 시간대를 제한하여 데이터를 수집하였다.

본 프로젝트에서는 아래의 샘플데이터 중 본선 1,2차에 대한 데이터 셋으로 프로젝트를 수행한다.



**[그림 1] 데이터셋 샘플데이터**

**2.2. 데이터셋 구조**

데이터셋은 51개의 센서에서 측정된 값과 구간 반복 횟수(Trip), 시간(Time(s)), 운전자(Driver)로 총 54개의 feature로 이뤄져 있다.

|  |  |  |
| --- | --- | --- |
| **구분** | **데이터 수** | **사이즈** |
| 본선 1차 학습 데이터셋 | 34,568 | 6,089KB |
| 본선 2차 학습 데이터셋 | 108,224 | 19,690KB |
| 본선 1차 평가 데이터셋 | 7,068 | 1,287KB |
| 본선 2차 평가 데이터셋 | 12,756 | 2,336KB |

**[표 1] 데이터셋 사이즈**

본 프로젝트에서는 51개의 센서 중 사례연구에서 진행한 실제 운전자의 주행패턴과 직접적인 연관이 있는 15개의 feature들로 진행한다.

|  |  |  |  |
| --- | --- | --- | --- |
| **Feature** | **Type of vehicle data** | **Range** | **Description** |
| Long-term fuel trim bank1 | Fuel | -100~100(%) | 주행에 따른 연료 조절 시스템의 조정 값 |
| Intake air pressure | Fuel | 0~255(kPA) | 엔진 흡기압 |
| Accelerator Pedal value | Fuel | 0~100(%) | 가속 페달 압력의 정도 |
| Fuel consumption | Fuel | 0~10000(mcc) | 엔진의 연료 소모량 |
| Friction torque | Engine | 0~100(%) | 엔진의 마찰 토크 |
| Maximum indicated engine torque | Engine | 0~100(%) | 최대 엔진 토크 |
| Engine torque | Engine | 0~100(%) | 현재 엔진 토크 |
| Calculated load value | Engine | 0~100(%) | 엔진에 대한 부하량의 계산 값 |
| Activation of Air compressor | Engine | 0 or 1 | 차량 에어 컴프레셔의 작동 여부 |
| Engine coolant temperature | Engine | -40~215(℃) | 엔진 냉각수 온도 |
| Transmission oil temperature | Transmission | -40~215(℃) | 변속기 오일 온도 |
| Wheel velocity, front, left-hand | Transmission | 0~511.75(km/h) | 차량 전방 좌측 휠 속도 |
| Wheel velocity, front, right-hand | Transmission | 0~511.75(km/h) | 차량 전방 우측 휠 속도 |
| Wheel velocity, rear, left-hand | Transmission | 0~511.75(km/h) | 차량 후방 좌측 휠 속도 |
| Torque converter speed | Transmission | 0~16383.75(rpm) | 토크 컨버터 속도 |

**[표 2] 주행패턴 분석에 사용되는 Features**

**3. 전처리**

**3.1. 전처리 과정**

전처리 과정은 사용자 분류, 주행 구간 분류, Feature 분류, Sliding Window 적용, 통계값 적용의 5단계로 분류된다.

1. **사용자 분류**

데이터셋에서는 A~E의 5명의 운전자로 라벨링 되어 있다. 이 5명의 운전자에 대한 데이터를 각각 분류하여 운전자별 데이터를 생성한다.

1. **주행 구간 분류**

데이터를 수집할때 운전자가 한 구간에 대한 주행을 여러번 반복 수행하였고 데이터셋에서는 이를 Trip이라는 Feature로 구분하였다. 한 구간에 대한 데이터가 필요하므로 이 역시 분류한다. 위에서 분류된 운전자별 데이터에 각 구간별 데이터로 더 상세 분류되어 생성한다.

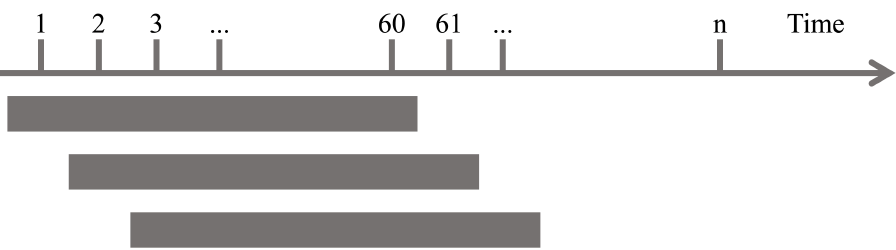
1. **Feature 분류**

데이터셋의 51개의 모든 Feature가 주행패턴 분석에 사용되는 것이 아니기에 이 중 [표 2]에 해당되는 Feature만을 출력하기 위한 분류가 필요하다.

a, b, c 과정을 거쳐 전처리된 파일은 “운전자명\_구간반복횟수.csv”의 파일명으로 15개의 Feature로 이뤄진 파일이 ./normal 디렉토리에 생성된다.

1. **Sliding Window 적용**

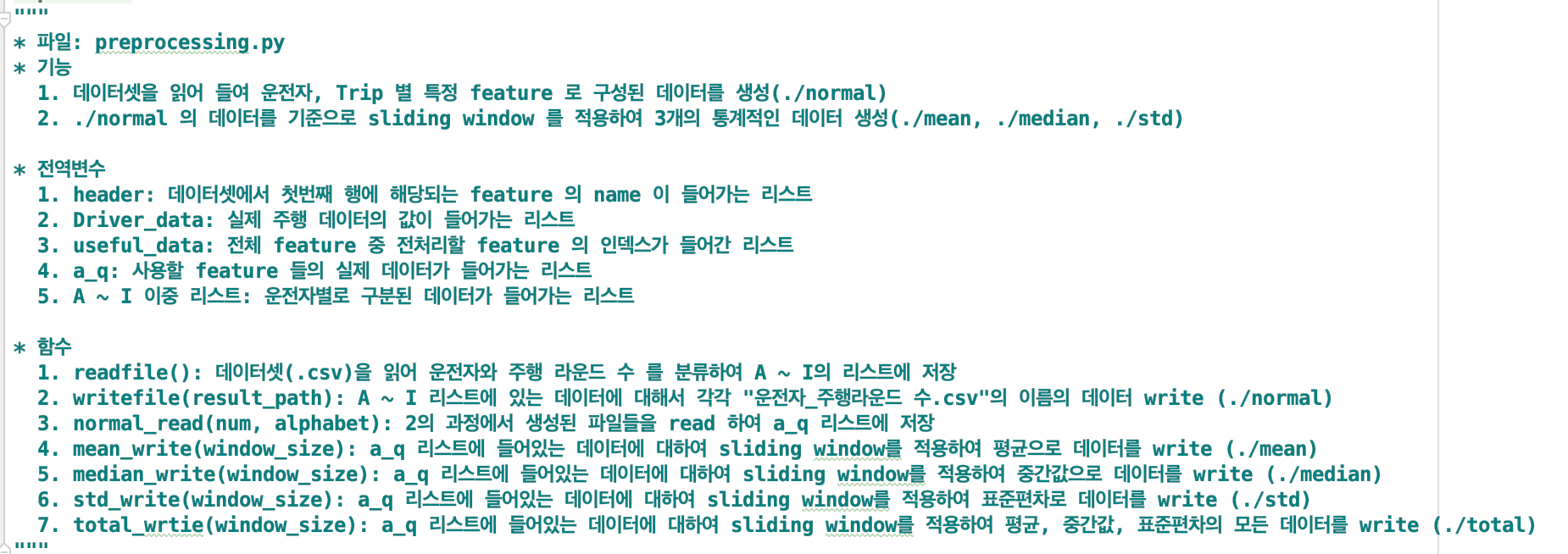
분류 알고리즘을 통한 성능향상을 위해 각 Feature들에 통계적인 특성을 적용한다. 데이터셋의 데이터는 시계열 기반의 데이터로 1초마다 각 센서 값이 저장된 것 이다. 하지만 이 값들은 주위 상황이나 노이즈 값들에 의해 이상데이터가 같이 수집될 수 있기에 이러한 상황을 고려해야 하기에 통계적인 특성을 추가 구성한다. 사용하려는 통계값은 평균, 중간값, 표준편차이다. 이를 적용하기 위해 Sliding Window를 아래 [그림 2]와 같이 적용한다.



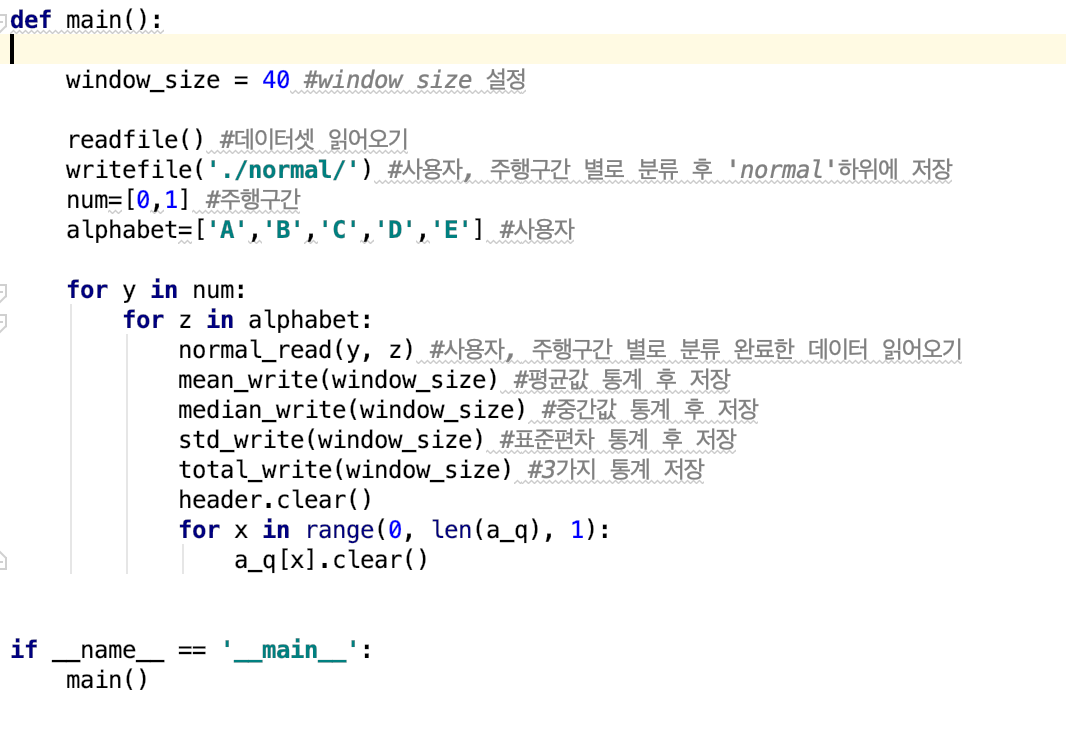
**[그림 2] 통계적 특성 적용을 위한 Sliding Window**

1. **통계 적용**

c에서 생성된 normal데이터를 d에서 적용된 sliding window size 만큼의 데이터에 대한 평균, 중간값, 표준편차를 적용하여 추가적인 통계적인 파일을 생성한다. 각각 ./mean, ./medium, ./std 디렉토리에 파일이 생성된다.

**3.2. 전처리 개발**

**[그림 3] 주요 변수 및 함수 설명**

****

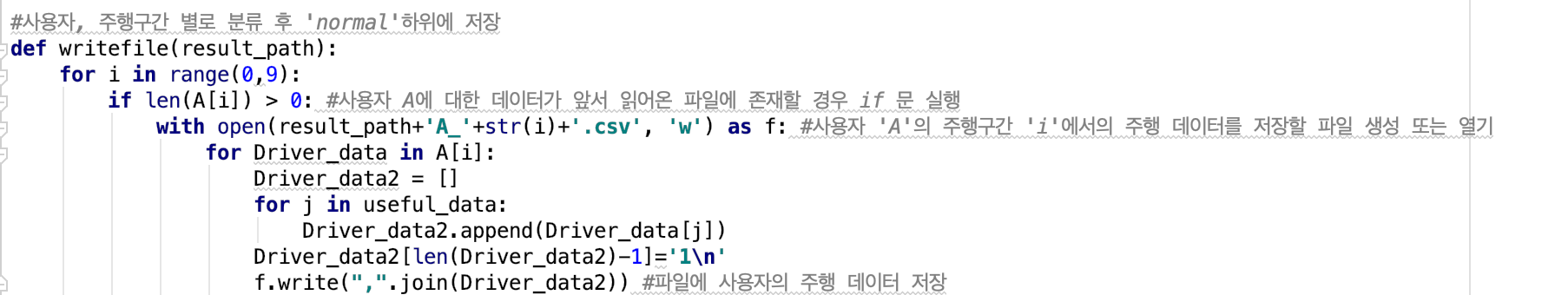
**[그림 4] main 함수**

main 함수는 윈도우 사이즈를 설정하고 전처리 과정을 수행한다. 먼저 데이터셋을 읽어 들인 후 사용자, 주행구간 별로 데이터를 분류하여 ‘normal’ 폴더 하위에 데이터를 저장한다.이후 사용자, 주행구간 별로 파일을 읽어 들여 평균, 중간값, 표준편차 3가지 종류로 통계를 진행하여 각각 파일로 저장하며 3가지 통계치를 모두 저장하는 파일도 생성한다.

****

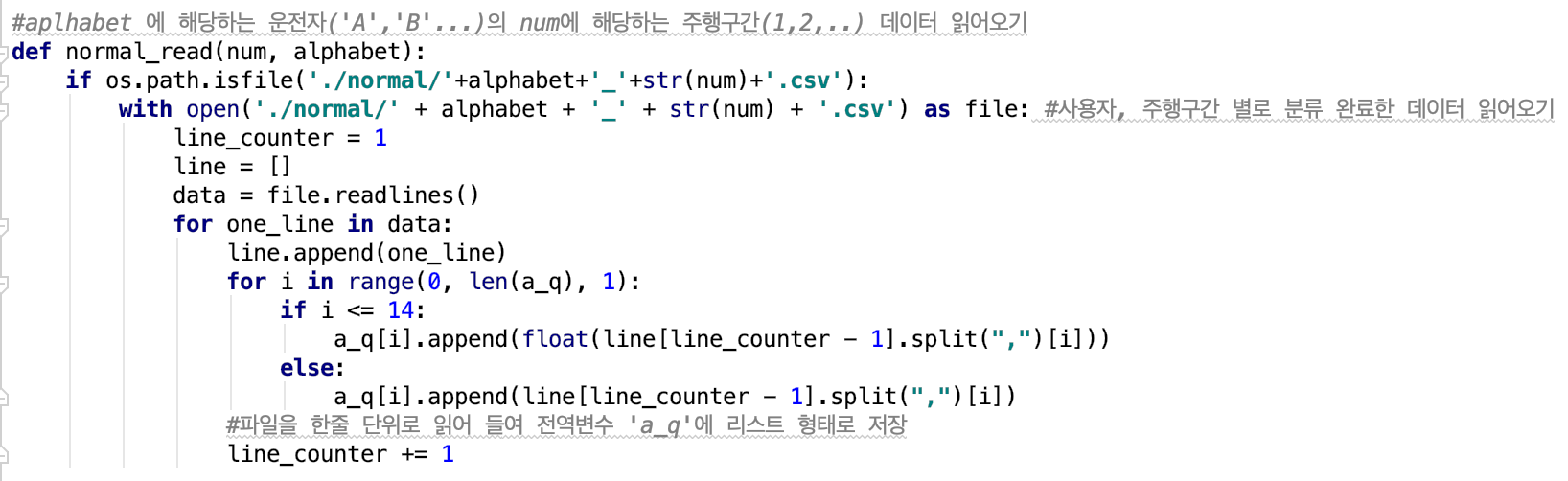
**[그림 5] readfile 함수**

readfile은 데이터셋을 읽어 들인 후 사용자, 주행도로 별로 데이터를 분리하는 기능을 수행한다. readline()함수를 통해 엑셀파일의 내용을 한 줄 단위로 읽어 들이며 리스트 형태의 변수 ‘Driver\_data’ 에 임시 저장한다. 사용자와 주행도로를 구분하기 위해 ‘Driver\_data’ 리스트에서 사용자를 의미하는 53번째 인자와 주행구간을 의미하는 51번째 인자를 활용한다. 구분 된 데이터는 이중리스트(A,B,....)에 저장한다.



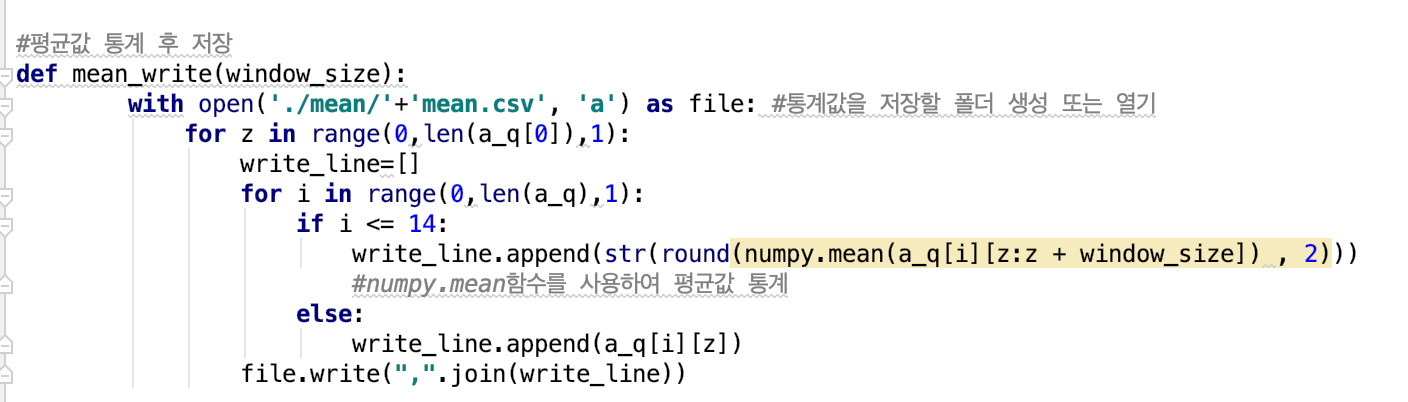
**[그림 6] writefile 함수**

writefile 함수는 앞서 사용자, 주행구간 별로 분류한 데이터를 ‘normal’ 이라는 디렉토리 밑에 ‘사용자\_주행구간.csv’ 과 같은 이름의 엑섹파일로 저장한다. 이때 전역변수 ‘userful\_data’를 활용하여 전체 feature 중 전처리 할 feature만 선별한다.



**[그림 7] normal\_read 함수**

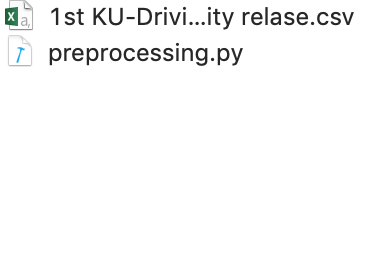
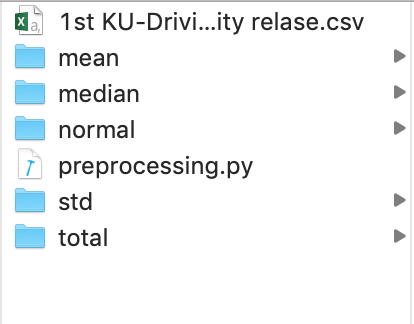
normal\_read 함수는 통계값을 산출하기 위해 사용자, 주행구간 별로 저장된 데이터를 읽어와 전역변수 ‘a\_q’에 저장하는 기능을 한다. 함수의 인자로 넘어온 사용자, 주행구간을 바탕으로 ‘사용자\_주행구간.csv’ 파일을 읽어 들인 후 ‘a\_q’ 변수에 파일의 내용을 계속해서 추가한다.



**[그림 8] mean\_write 함수**

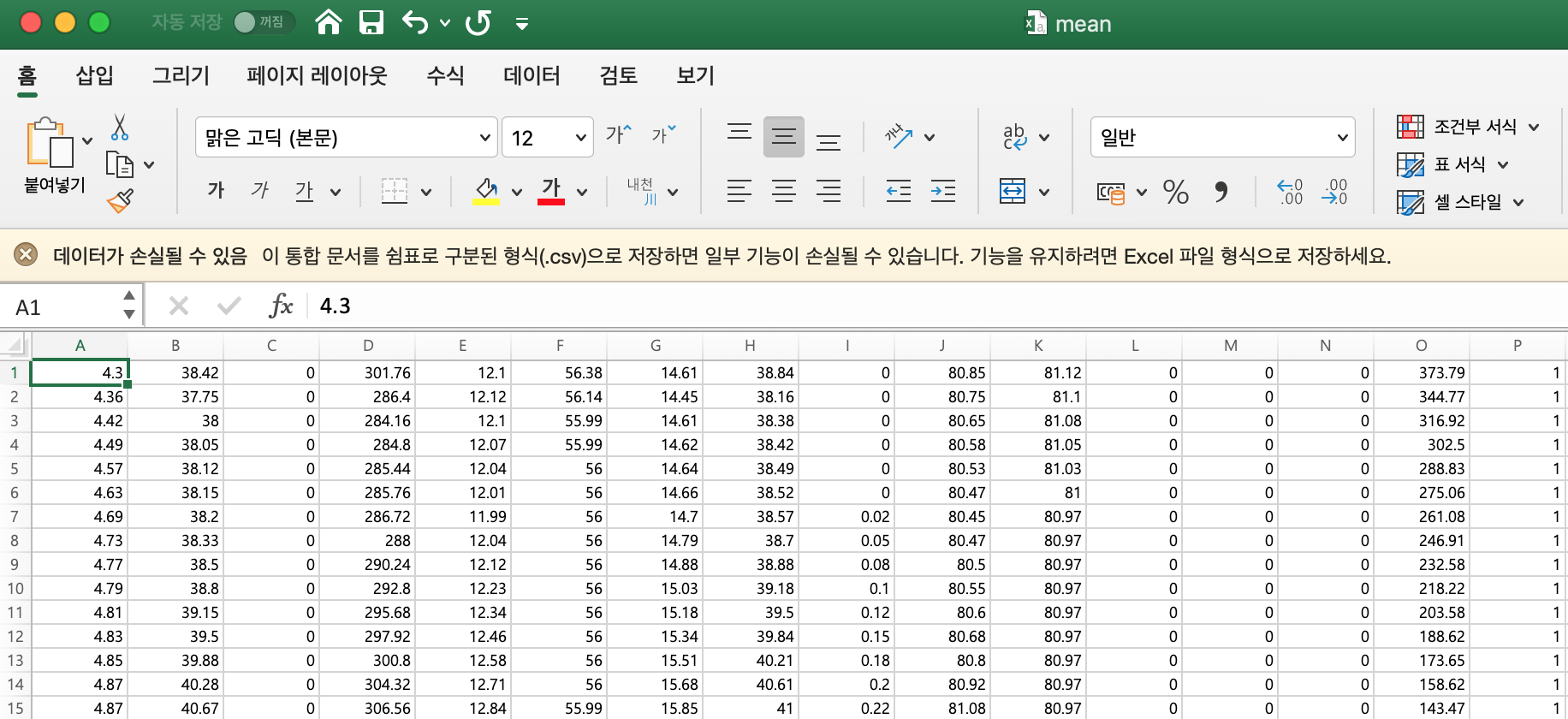
mean\_write 함수는 평균값을 산출하는 함수이다. 평균값은 ‘mean.csv’에 저장된다. 통계값 산출은 numpy 모듈의 mean 메소드를 사용하며, normal\_read 함수를 통해 데이터를 저장한 ‘a\_q’변수를 활용한다. 중간값, 표준편차도 mean\_write와 같이 구성되며 numpy.mean 메소드 대신 numpy.media, numpy.std 를 사용한다.

**3.3. 전처리 결과**

** **

**[그림 9] 실행 전 [그림 10] 실행 후**

전처리 파일 실행 후 사용자, 주행구간 별로 분류한 파일들이 저장될 ‘normal’ 폴더와 각종 통계값들이 저장될 ‘mean,median,std,total’ 폴더가 생성된다.



**[그림 11] mean 파일**

평균 값이 저장된 mean.csv 파일이다. 중간값 표준편차도 이와 동일하게 csv파일로 저장된다.

|  |  |  |
| --- | --- | --- |
| **데이터 종류** | **데이터 개수** | **비고** |
| 1st KU-Driving Dataset final - City relase | 7068(헤더 제외) | 원본 |
| A | 1500 | 분류 |
| B | 1232 | 분류 |
| C | 1400 | 분류 |
| D | 1348 | 분류 |
| E | 1588 | 분류 |
| 총합 | =7068 |  |

**[표 3] 주행자별 데이터 분류**

1st KU-Driving Dataset final - City relase 데이터 셋에는 주행구간이 나누어져 있지 않아 사용자 별로만 구별되어다.

**4. 기계학습**

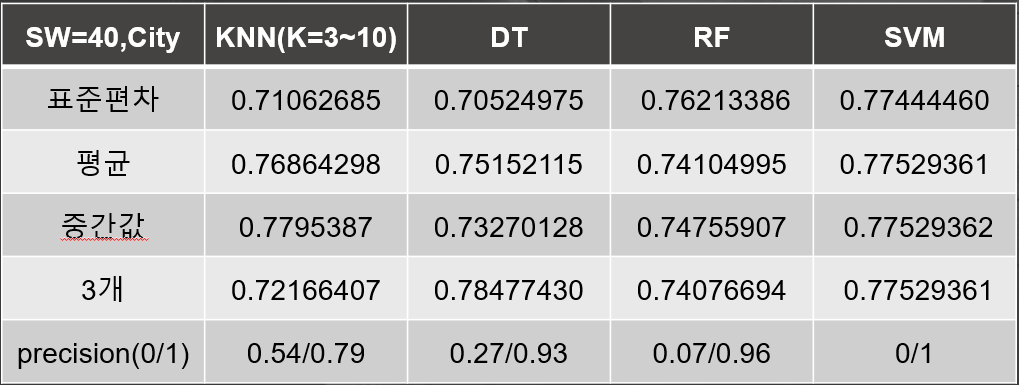
**4.1. 알고리즘 선정**

알고리즘은 운전자를 측정하는 것이 아닌 선별하기 위한 것이기 때문에 classifier 알고리즘인 KNN, DecisionTree, RandomForest, SVM 4가지를 사용하였다. 추가로 오탐지율을 줄이기 위해서 정확도와 정밀도를 측정하여 시뮬레이션에 활용하였다.

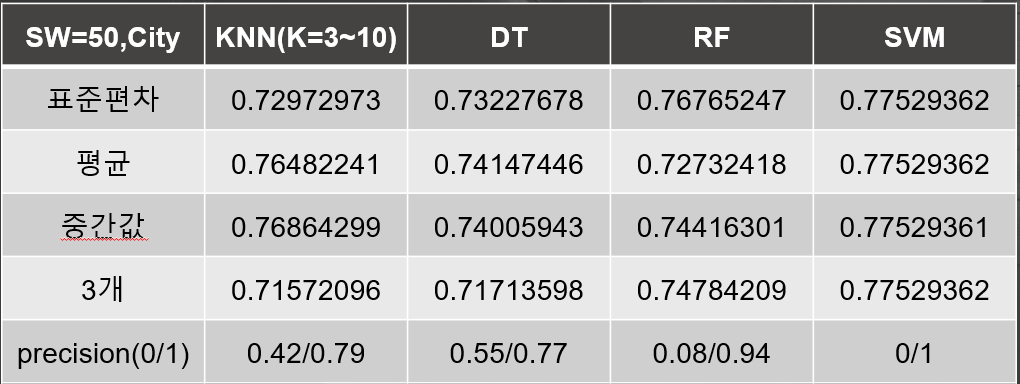
**4.2. 기계학습 개발**

|  |
| --- |
| **import pandas as pd**  **import numpy as np**  **import sklearn.metrics as metrics**  **from sklearn.neighbors import KNeighborsClassifier # sklearn 라이브러리에서 KNN을 사용하기 위한 import**  **from sklearn.tree import DecisionTreeClassifier # sklearn 라이브러리에서 DesicionTree을 사용하기 위한 import**  **from sklearn.ensemble import RandomForestClassifier # sklearn 라이브러리에서 RandomForest을 사용하기 위한 import**  **from sklearn import svm # sklearn 라이브러리에서 SVM을 사용하기 위한 import**  **from sklearn.metrics import confusion\_matrix, accuracy\_score, classification\_report, precision\_score, recall\_score**  **# sklearn 라이브러리에서 정확도와 정밀성, 재현율을 출력하기 위한 import**  **name = ['1','2','3','4','5','6','7','8','9','10','11','12','13','14','15', 'Driver'] # 데이터 셋을 넣기 위하여 만들어준 name array**  **train\_d = pd.read\_csv('train\std\std.csv', names = name) # train 데이터가 있는 위치로 이 데이터를 통해 학습한다. std 표준편차, mean 평균, median 중간값, total 3개**  **test\_d = pd.read\_csv('test\std\std.csv', names = name) # test데이터가 있는 위치로 이 데이터를 통해 학습한다. std 표준편차, mean 평균, median 중간값, total 3개**  **train\_x = train\_d[name[0:14]] # train데이터에서 15개의 데이터를 학습**  **train\_y = train\_d[name[15]] # 16번째에 있는 Driver를 라벨로 학습한다.**  **test\_x = test\_d[name[0:14]] # test데이터에서 15개의 데이터를 학습**  **test\_y = test\_d[name[15]] # test데이터에서 16번째에 있는 Driver를 라벨을 읽어온다. 답을 넣는 이유는 정확도와 정밀성을 측정하기 위함이다.**  기본적인 라이브러리 import와 데이터 넣는 순서  for k in range(3, 10, 1):  knnclf = KNeighborsClassifier(n\_neighbors = k, weights= 'uniform')  # weight는 uniform, distance 두가지 uniform은 그냥 거리 distance는 거리의 역수 weight에 따른 차이는 0.001정도의 차이  knnclf = Pipeline([('norm', StandardScaler()), ('knn', knnclf)])# nomarlization  knnclf.fit(train\_x, train\_y) #knn 트레이닝 데이터로 학습  knnpre = knnclf.predict(test\_x) #knnpre ? knn이 예측한 라벨 부분 이 라벨과 test\_y간 비교 test\_y는 정답  knnac\_score = metrics.accuracy\_score(test\_y, knnpre)# 답과 예측한 결과로 정확도 측정  print("SW= 40, KNN weights = uniform, Data = std, K의 값 ",k, knnac\_score)  print(metrics.classification\_report(test\_y, knnpre)) # 답과 예측한 결과로 metrics 출력  KNN 알고리즘 최적값 K를 찾기위해 for문을 사용하였다. 최적값은 넣은 데이터마다 다르지만 5, 9 가 많았는데 이 이유는 5명의 데이터를 기반으로 해서 그런 것임을 추측할 수 있다. 파라미터는 uniform, distance 가 있는데 uniform은 거리, distance는 1/거리로 역수로 각 데이터를 계산한다. 두 파라미터 간 결과의 차이는 별로 나지 않았다.  #DecisionTree  dt = DecisionTreeClassifier()  #DecisionTree 선언  dt.fit(train\_x, train\_y)  # Data train  dtpre = dt.predict(test\_x)  # DecisionTree가 학습한 데이터를 기반으로 예측한 결과  dtac\_score = metrics.accuracy\_score(dtpre, test\_y)  # 예측한 결과를 기반으로 구한 정확도  print("SW= 40, Data = std, DecisionTreeClassifier accuracy: ", dtac\_score )  # 정확도 출력  print(metrics.classification\_report(dtpre, test\_y))  # precision, recall 등 메트릭스 출력  DecisionTree알고리즘  #RandomForest  rfclf = RandomForestClassifier(n\_estimators=50)  # RandomForest 선언  rfclf.fit(train\_x, train\_y)  # RandomForest train  rfpre = rfclf.predict(test\_x)  # RandomForest가 학습한 데이터를 기반으로 예측한 결과  rfac\_score = metrics.accuracy\_score(test\_y, rfpre)  # 예측한 겨로가를 기반으로 구한 정확도  print("Randomforest accuracy : ", rfac\_score )  # 정확도 출력  print(classification\_report(rfpre, test\_y))  # precision, recall 등 메트릭스 출력  RandomForest알고리즘  for n in range (1, 102, 50):  # SVM의 파라미터 C를 1, 51, 101로 측정하기 위한 반복문  print('start rbf, gamma = 1000, C = ',n)  # gamma도 반복문으로 하려 했지만 시간이 너무 오래걸려 따로 정수값으로 주었다.  svmclf = svm.SVC(kernel='rbf', C=n, gamma=1000).fit(train\_x, train\_y)  # svm 선언 및 파라미터 설정, train  svmpre = svmclf.predict(test\_x)  # svm으로 예측한 결과  svmac\_score = metrics.accuracy\_score(svmpre, test\_y)  # 정확도 측정  print("SVM linear accuracy : ", svmac\_score )  # 정확도 출력  print(classification\_report(svmpre, test\_y))  # 메트릭스 출력 percision 등  print('end')  SVM 알고리즘 linear모델 rbf 모델 둘 다 큰 차이가 없었다. 파라미터 값을 C = 1, 51, 101 gamma = 10, 100, 1000으로 총 9가지를 비교해 보았지만 편차가 크게 나지 않았다. |

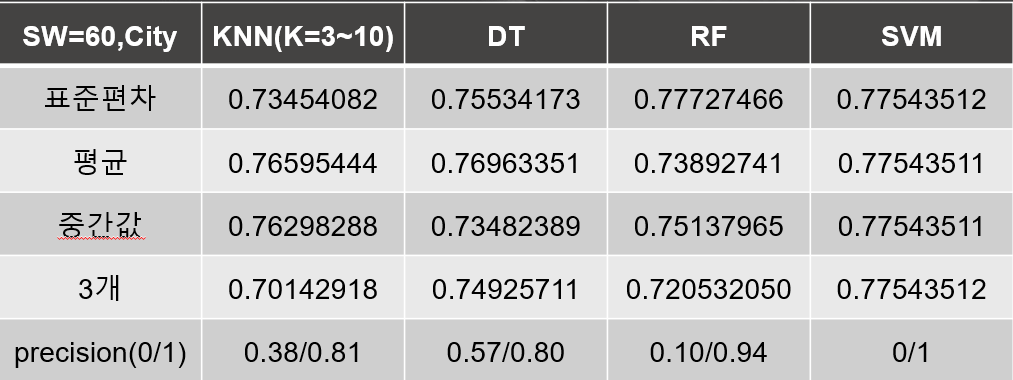
**4.3. 정확도 측정**



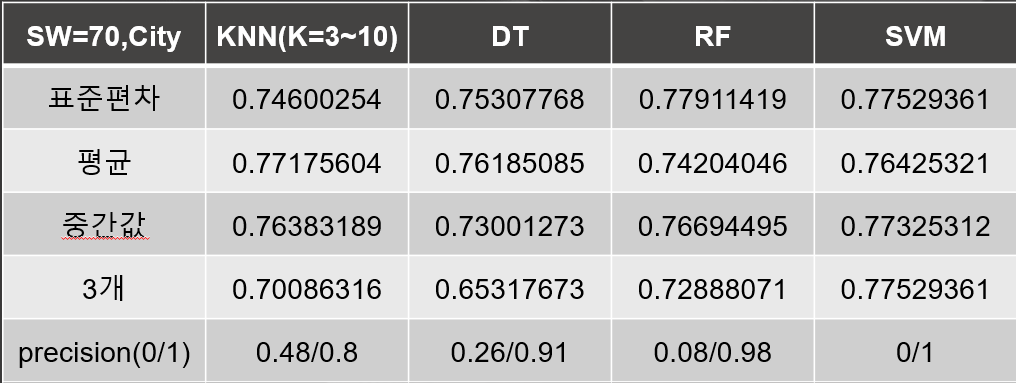
0.7795387 0.78477430 0.76213386 0.77529362

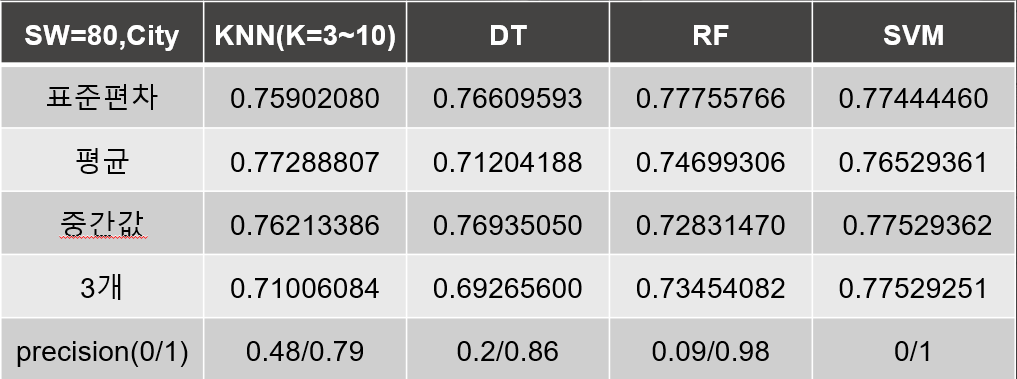


0.76864299 0.74147446 0.76765247 0.77529362

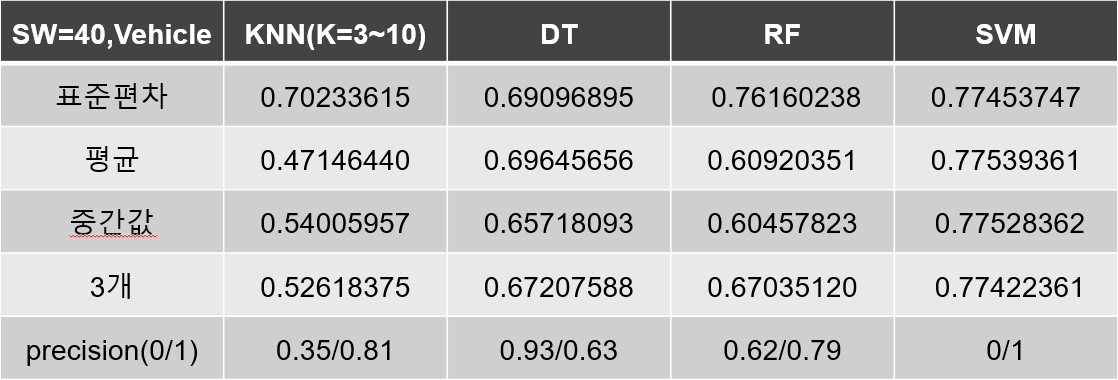


0.7659544 0.76963351 0.77727466 0.77543512

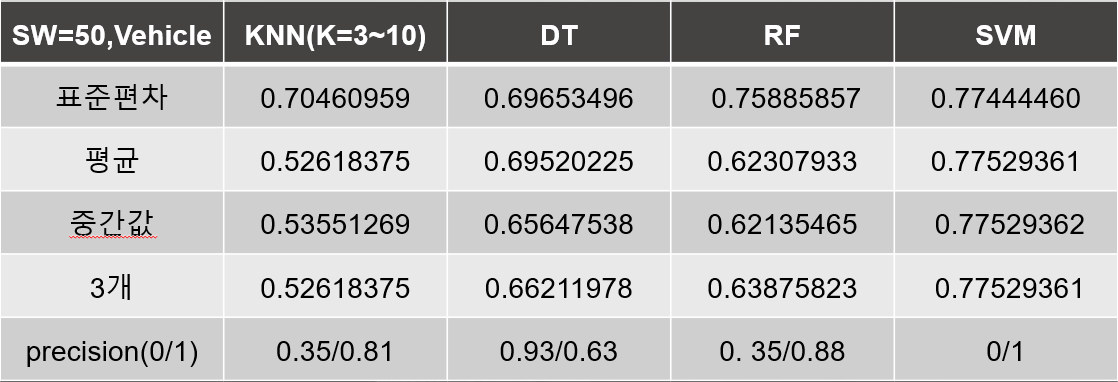
 0.77175604 0.76185085 0.77911419 0.77529361



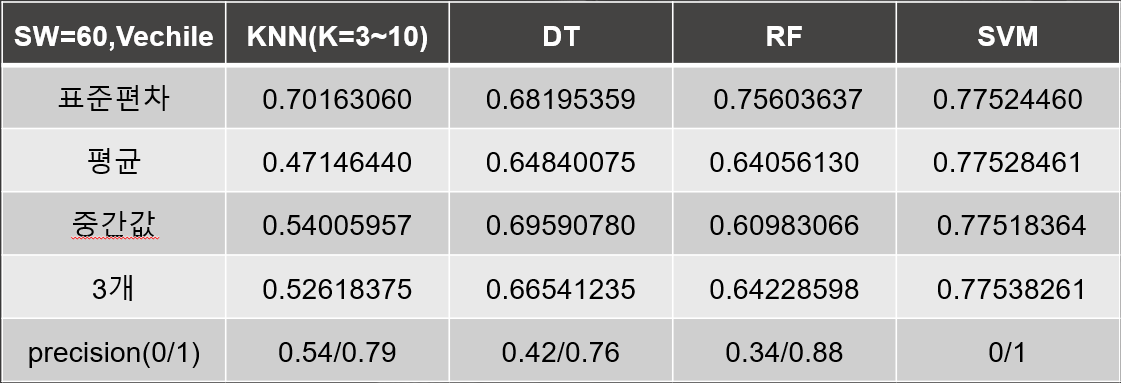
0.77288807 0.76935050 0.77755766 0.77529362



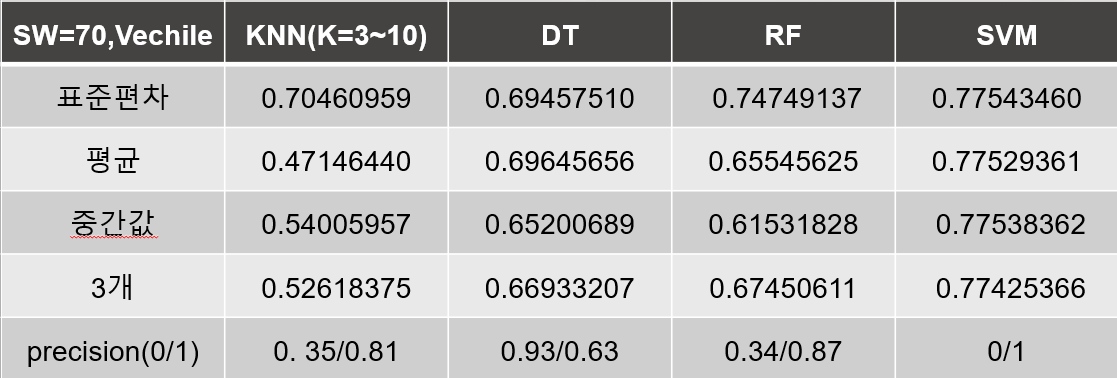
0.70233615 0.69645656 0.76160238 0.77528362



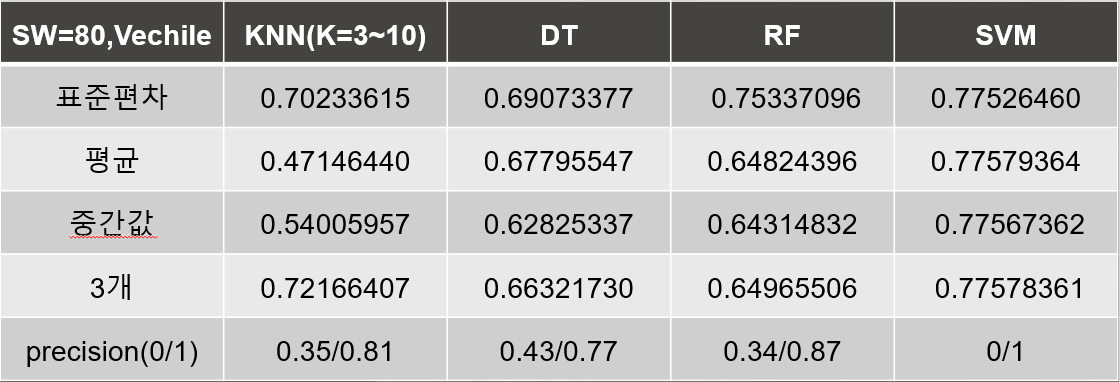
0.70460959 0.69653496 0.75885857 0.77529362



0.70163060 0.6959078 0.75603637 0.77538261



0.70460959 0.9645656 0.74749137 0.7754346



0.70233615 0.69073377 0.75337096 0.77579364

도심 데이터와 고속도로 데이터를 볼 때 전체적인 정확도는 도심이 높고 1에 대한 정밀도 또한 높다. 하지만 고속도로에서 0에대한 정밀도가 좀 더 높은 것을 볼 수 있다. 이는 도심에서 식별된 인원을 좀 더 잘 잡아내고 고속도로에서는 정확도가 비교적 낮지만 식별되지 않은 사람을 더 잘 잡아낸다는 것을 볼 수 있다.

SVM은 차이가 거의 없고 0에대한 정밀도가 0인것으로 보아 데이터 혹은 전처리 과정에 오류가 있다고 추측, 좀 더 수정이 필요한 것으로 보인다.

정확도가 제일 높을 때 들어가는 통계치가 sliding window 크기 별로 다른 것으로 보아 데이터간의 상관관계를 찾지 못했다.

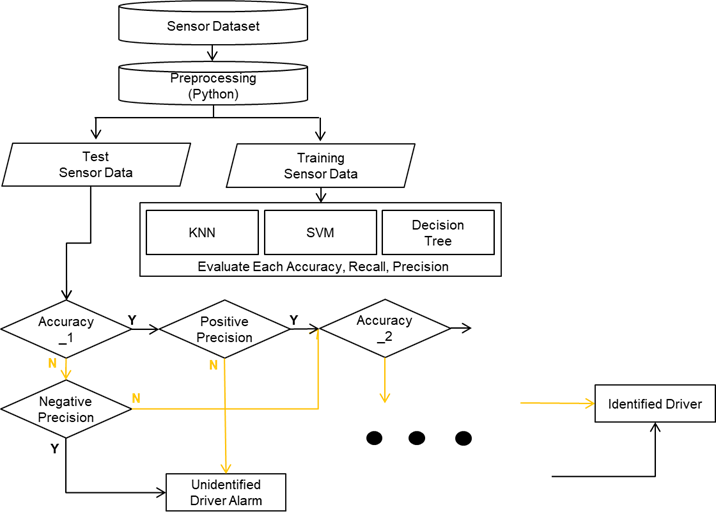
도심데이터는 제일 높은 정확도가 70% 중후반대로 비슷하게 나왔지만 고속도로 데이터에서는 Randomforest, SVM이 좀더 높은 정확도를 보였다.

전체적으로 볼때 sliding window 크기가 40일 때 정확도가 비교적 높은 수치를 보여 이를 기반으로 시뮬레이션을 구성하였다.

**5. Simulation**

**5.1.**  **Methodology Flowchart**

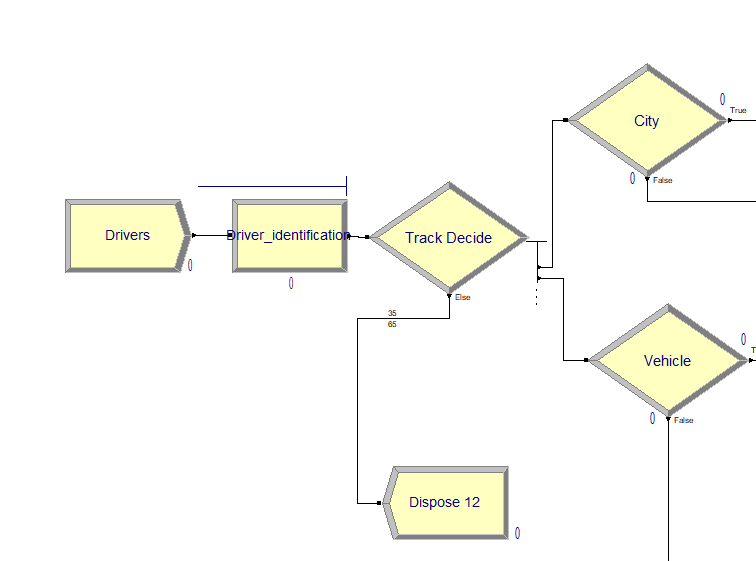
시뮬레이션 과정을 포함한 Methodology Flowchart는 아래와 같이 구성하였다.

****

1. 위의 데이터셋 파트에 [표1]의 데이터 셋을 살펴본 후 그 중에 이번 프로젝트에 사용할 데이터들을 추린다.
2. 추려낸 결과 데이터셋 파트의 [표2]에 나온 데이터들을 선정하여 기계 학습 알고리즘에 사용할 형태로 전처리 작업을 실행한다.
3. 각 알고리즘들을 실행하여 데이터 종류에 따른 정확도와 정밀도를 측정한다.
4. 학습 데이터로 기계 학습을 시킨 뒤, 테스트 데이터로 테스트하여 나온 결과(정확도, 정밀도)를 가지고 시뮬레이션을 진행한다.
5. 각 알고리즘 별로 정확도로 체크를 한 뒤 그 결과에 정밀도를 사용하여 한번 더 체크한다.
6. 이러한 과정을 직렬로 연결하여 여러 개의 알고리즘을 같이 사용한다.

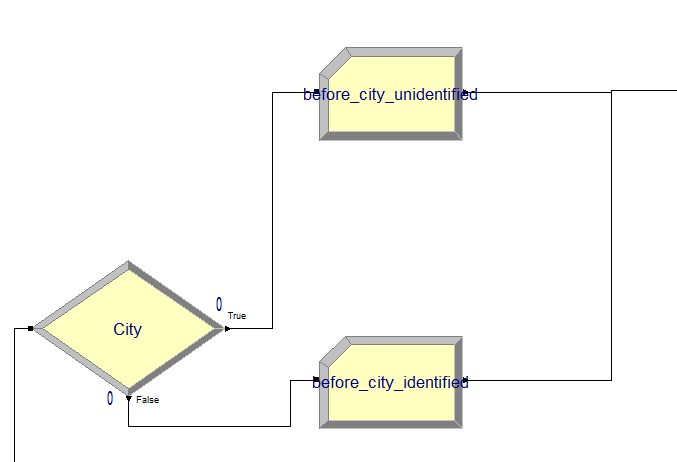
**5.2. Simulation 구현**

1. 주행 데이터를 가져와서 주행 유형별로 분류한다. 주행의 유형에는 city(도심)과 vehicle(고속도로), 2종류가 있다.

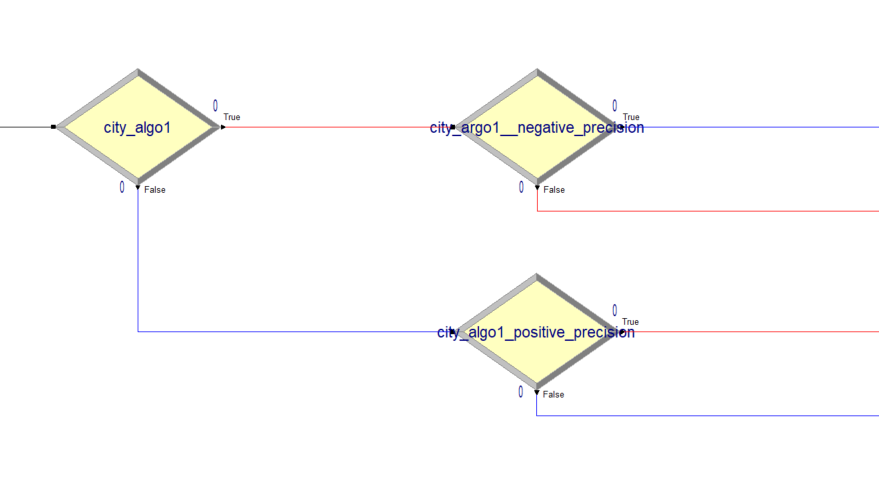


1. 결정된 도로 유형별로 기계학습된 알고리즘을 돌리기 전에 각 도로별 정상 운전자와 비정상 운전자의 수를 측정한다.

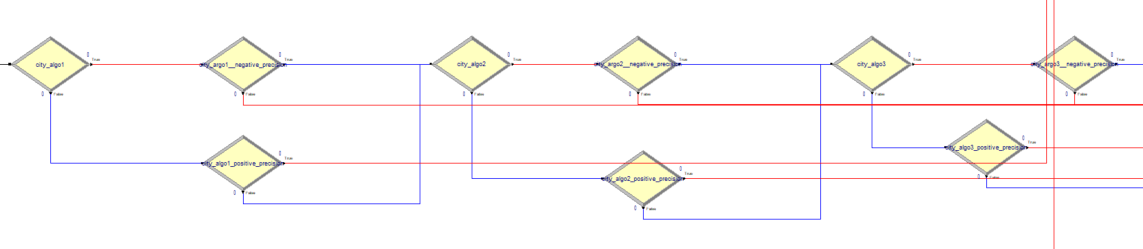
* 추후, 얼마나 탐지되었는지 체크해보기 위해



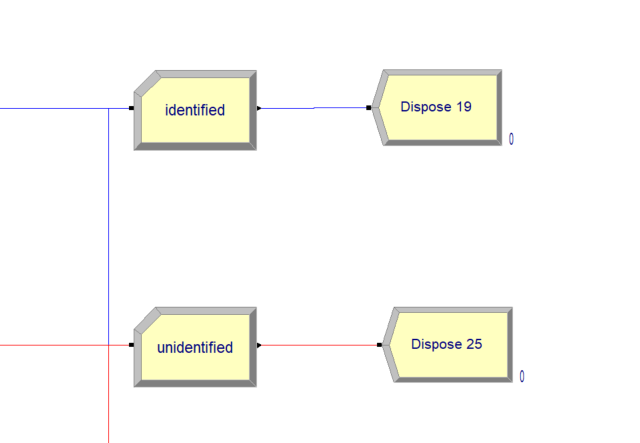
1. 정확도를 사용하여 1차적으로 체크하고 각 정상과 비정상에 대한 정밀도를 바탕으로 한번 더 체크한다.



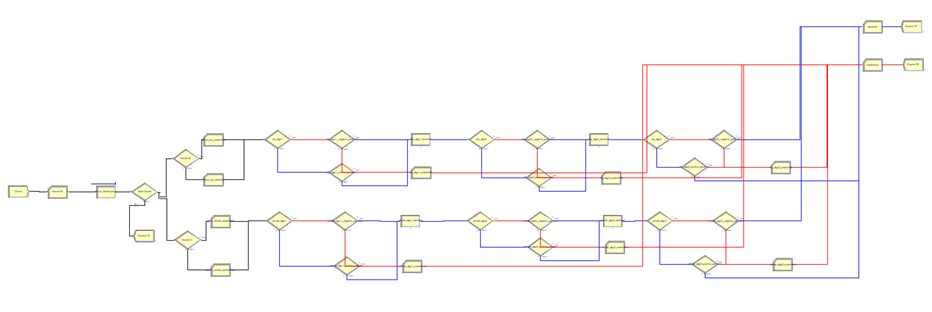
1. 3번의 과정을 직렬로 연결하여 여러번 알고리즘을 거치게 만들어서 어떤 순서로 배치했을 때 가장 높은 정확도가 나오는지 체크한다.



1. 마지막으로 최종 알고리즘을 통해 분류된 정상 운전자와 비정상 운전자의 개수를 저장한다.

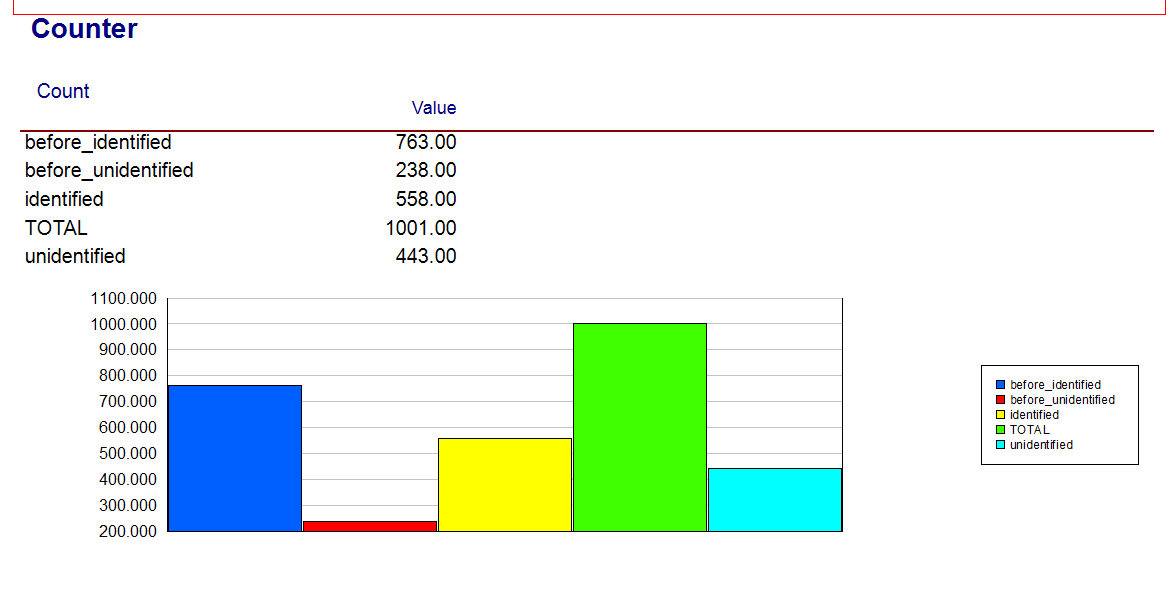


1. 전체 시뮬레이션 구성도는 아래와 같다.



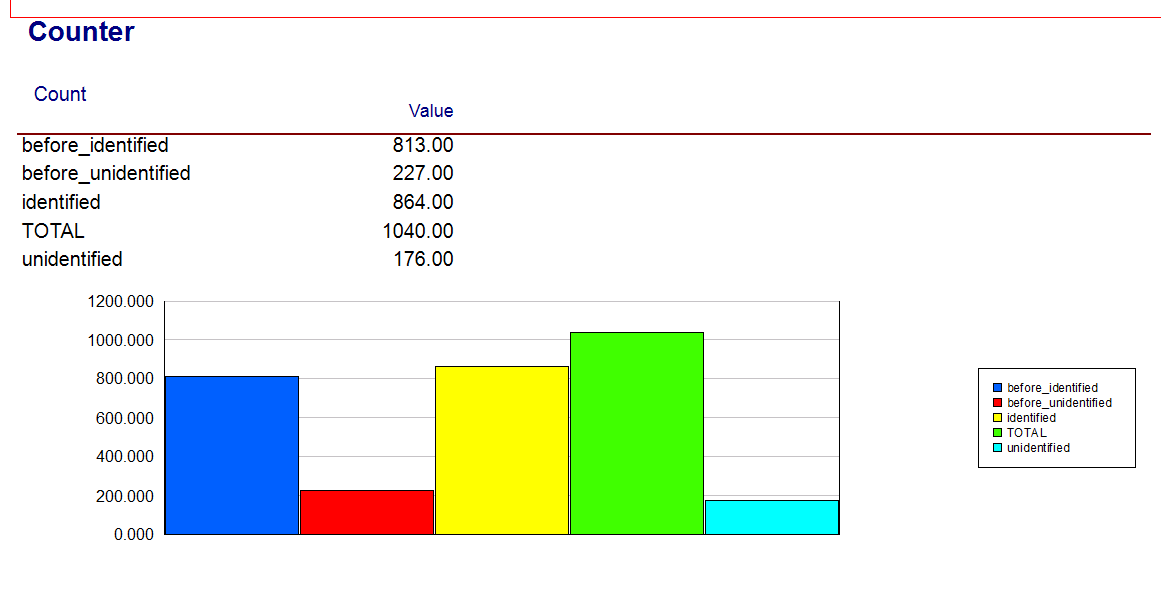
**5.3. Simulation 결과**

**<4개의 알고리즘을 사용한 보고서>**

****

처음에는 우선 4개의 알고리즘을 사용하여 시뮬레이션을 돌려보았다. 순서는 정확도가 높았던 순서대로 돌려보았는데 약 55% 내외의 정확도를 보였다. 정확도와 정밀도를 충분히 높게 확보하지 못한 상황에서 여러개의 알고리즘을 직렬로 사용했기 때문에 오히려 정확도가 떨어졌을지도 모른다는 생각이 들었고 3개의 알고리즘으로 다시 시뮬레이션을 실행해보았다.

**<3개의 알고리즘을 사용한 보고서>**



3개의 알고리즘을 사용했을 때 배치를 정확도를 기준으로 높은 순서대로 하였고 그 결과 4개의 알고리즘을 사용했을 때 보다 훨씬 향상된 80%전후의 정확도를 보였다.

각각 알고리즘의 정확도와 정밀도가 충분히 높은 상태라면 직렬의 방법을 사용하면 정확도가 높아질 것이라고 예상되지만, 충분한 정확도와 정밀도를 확보하지 못한다면 직렬의 방법을 사용할 때 오히려 정확도가 떨어지는 상황이 발생할 수 있는것으로 보인다.

**6.** **참고문헌**

[1] Fabio Martinellia, Francesco Mercaldoa, Albina Orlandob, Vittoria Nardonec, Antonella Santoned, Arun Kumar Sangaiahe. Human Behavior Characterization for Driving Style Recognition in Vehicle System

[2] Virojboonkiate, Nuttun, Peerapon Vateekul, and Kultida Rojviboonchai. "Driver identification using histogram and neural network from acceleration data.“

[3] Byung Il Kwak, JiYoung Woo, Huy Kang Kim. Know Your Master: Driver Profiling-based Anti-theft Method

[4] Arena Simulation Software, http://www.arenasimulation.com/

[5] 안드레아스 뮐러, 세라 가이도, Introduction to Machine Learning with Python, 한빛아카데미

[6] W. David Kelton, Simulation with ARENA, 교보문고

[7] 박응용, Do it! 점프 투 파이썬, 이지스퍼블리싱