

# 빅데이터 분석

산업정보시스템공학과

20162263 박윤아

## 중간고사 대체 보고서

: 통신사 이탈 여부 데이터 분석

<b>1.데이터 확인 및 전처리 .....</b>	<b>2</b>
1-1 데이터 특징 정리 및 전처리 정리.....	2
1-2 object형 변수 및 target 변수 전처리: 이진 변수화.....	3
1-3 연속형 변수의 상관관계.....	3
1-4 연속형 변수 전처리 .....	4
<b>2. 데이터 분석 .....</b>	<b>5</b>
2-1 사용 함수 정리 .....	5
2-2 Defalut값을 이용한 분류 분석 수행.....	6
2-3 GridSearchCV() 함수를 이용한 mlp 모델의 최적 파라미터 탐색 .....	6
2-4 GridSearchCV() 함수를 이용한 rf 모델의 최적 파라미터 탐색.....	7
2-5 결론 .....	8
<b>3. 고객의 특성 분석.....</b>	<b>8</b>
3-1 추가 함수 정리 .....	8
3-2 이탈 고객의 특성.....	8

## 1.데이터 확인 및 전처리

- .info()함수로 각 column의 변수 타입 확인, set함수와 value\_counts()함수를 통해 값의 종류 파악

### 1-1 데이터 특징 정리 및 전처리 정리

순서	Columns name	변수의 속성	기타 특징 및 전처리 방법
0	id		의미X ->삭제
1	고객 ID		의미X ->삭제
2	성별	Object형: '남', '여'	이진 변수로 변경
3	연령	12세~82세	(10대~80대)이진 변수로 변경
4	개시일	Object형: 날짜 (96년~99년 ->4개년도)	5와 동일 의미-> 삭제
5	서비스기간	연속형	4와 동일 의미
6	단선횟수	이산형	
7	지불방법		모두 같은 값 가짐->삭제
8	요금제	Object형: 'CAT 100', 'Play 100', 'CAT 50', 'Play 300', 'CAT 200'	이진 변수로 변경
9	이탈여부	Object형: '유지', '이탈'	target
10	핸드셋	Object형: 'S80', 'SOP10', 'ASAD170', 'BS110', 'ASAD90', 'S50', 'CAS60', 'BS210', 'WC95', 'CAS30', 'SOP20'	이진 변수로 변경
11	주간통화횟수	이산형	
12	주간통화시간_분	연속형	
13	야간통화횟수	이산형	
14	야간통화시간_분	연속형	
15	주말통화횟수	이산형	
16	주말통화시간_분	연속형	
17	국제통화시간_분	연속형	
18	국내통화요금_분	연속형	29와 중복
19	평균주간통화시간		12÷11 ->삭제(데이터 축소)
20	평균야간통화시간		14÷13 ->삭제(데이터 축소)
21	평균주말통화시간		16÷15 ->삭제(데이터 축소)
22	국내통화횟수	이산형	
23	국내통화시간_분	연속형	
24	평균국내통화시간		23÷22 ->삭제(데이터 축소)
25	총통화시간_분		17+23 ->삭제(데이터 축소)
26	통화량구분	Object형: '저', '고', '중저', '중', '중고', '무'	29의 na 값 제거 시 '무'는 사라짐 이진 변수로 변경
27	요금부과시간	연속형	
28	분당통화요금	연속형	na 값 제거 필요 (데이터 충분)
29	국내통화요금		18과 중복->삭제
30	총통화요금	연속형	
31	부과요금	연속형	
32	납부여부	Object형: 'High CAT 50', 'High CAT 100', 'High Play 100', 'OK'	Ok가 아닌 경우 ->(High '요금제이름')표시 이진 변수로 변경
33	평균납부요금		31÷25 ->삭제(데이터 축소)
34	주간통화비율,		34+35+36=1

			12÷(12+14+16) ->삭제(데이터 축소)
35	야간통화비율		14÷(12+14+16) ->삭제(데이터 축소)
36	주말통화비율		16÷(12+14+16) ->삭제(데이터 축소)
37	국제통화비율	연속형	
38	통화품질불만	Object형: 'T', 'F'	
39	미사용		26의 '?' 제거 시 모두 F 값을 가짐 ->삭제

### 1-2 object형 변수 및 target 변수 전처리: 이진 변수화

- object 형 변수인 성별, 요금제, 핸드셋, 통화량구분, 납부여부, 통화품질불만을 이진 변수화 했다.
- 추후 해석의 편의를 위해 연령을 10대부터 80대까지 8개 그룹으로 나누어 이진 변수화 하였다.
- 연령의 경우 10대~80대까지 이진변수화 하였다.
- 이진 변수화 한 특징을 합친 결과는 아래와 같다.

```

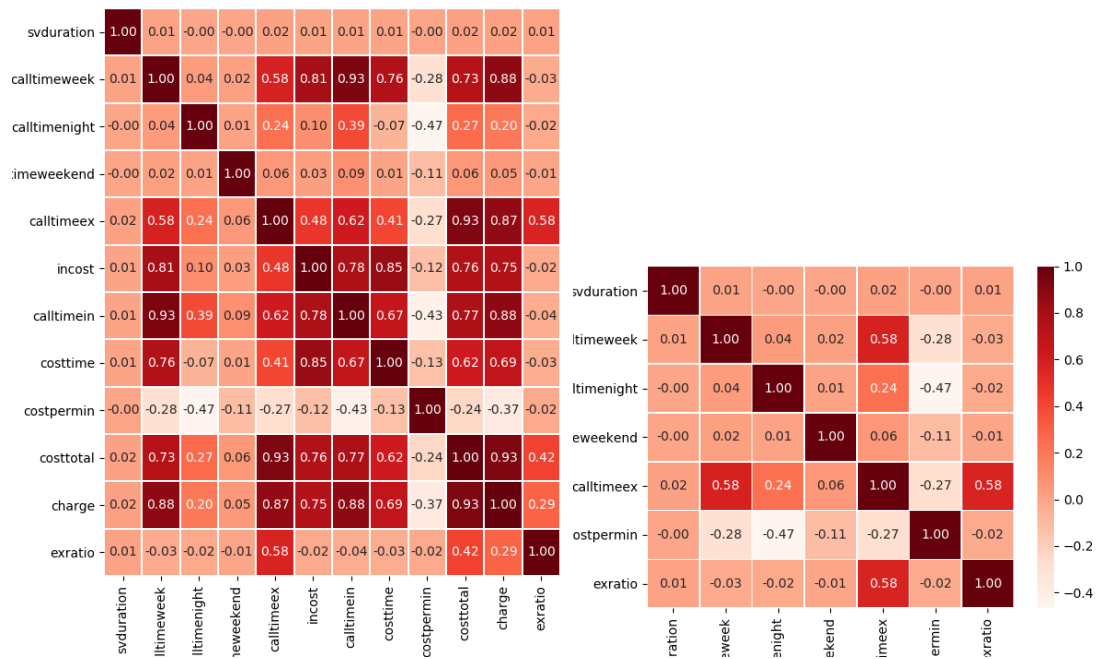
>>> binary_features = np.column_stack([np_gender, np_age, np_cplan, np_handset, np_callvol, np_pay, np_quality])
>>> binary_features
array([[1, 0, 1, ..., 0, 0, 1],
       [0, 1, 0, ..., 0, 0, 1],
       [0, 1, 0, ..., 0, 0, 1],
       ...,
       [0, 1, 1, ..., 0, 1, 0],
       [0, 1, 0, ..., 0, 1, 0],
       [0, 1, 0, ..., 0, 1, 0]])

```

- target 변수인 이탈여부 또한 '유지', '이탈'로 구분된 object형 변수이므로 이진 변수화 하였다.
- 이산형 변수인 단선횟수, 주간통화횟수, 야간통화횟수, 주말통화횟수, 국내통화횟수는 이진 변수화 하기에 너무 많은 더미 변수가 필요하기 때문에 값을 그대로 사용

### 1-3 연속형 변수의 상관관계

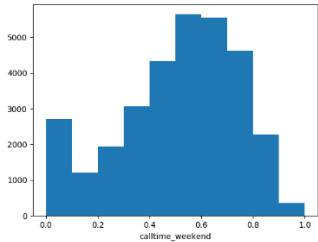
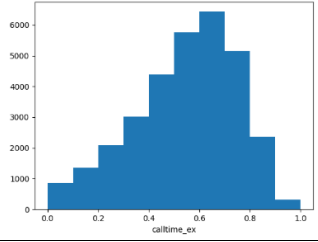
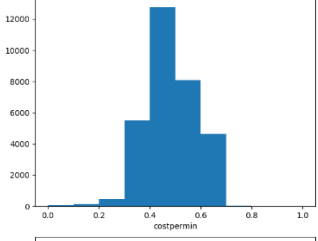
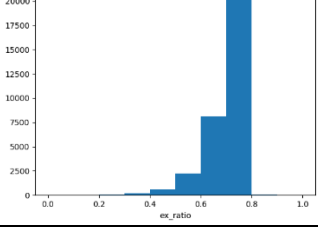
- 연속형 변수의 correlation을 corr()함수로 확인 후 seaborn의 heatmap을 통해 나타낸 결과는 아래 그림과 같다.
  - 타 변수와 강한 상관관계를 가지는 변수를 제거하며, 이때 절댓값 0.6을 기준으로 한다.
- 따라서 국내통화요금\_분, 국내통화시간\_분, 요금부과시간, 총통화요금, 부과요금을 삭제한다.
- 5개의 변수를 제거한 후 다시 heatmap으로 나타낸 결과는 오른쪽과 같으며 절댓값 0.6이상의 값은 없다.



#### 1-4 연속형 변수 전처리

- plt.hist() 함수를 통해 변수의 형태를 확인한 후 정규성을 만족하도록 보정하였다.
- log를 씌워서 보정하는 경우에 log0가 음의 무한대이므로 10을 더해서 진행하였다.

변수 이름	보정 방법	보정 후 그래프
서비스기간	0,1 normalization	
주간통화시간_분	0,1 normalization log씌워서 보정	
야간통화시간_분	0,1 normalization log씌워서 보정	

주말통화시간_분	0,1 normalization log씩워서 보정	
국제통화시간_분	0,1 normalization log씩워서 보정	
분당통화요금	0,1 normalization log씩워서 보정	
국제통화비율	100 곱해서 값을 받아왔다. 0,1 normalization log씩워서 보정	

## 2. 데이터 분석

### 2-1 사용 함수 정리

- 'UtilForHw'에 사용할 함수들을 따로 정리하였다. 사용함수는 아래와 같다.

- getxfold(data, turn, fold): data를 train데이터와 test데이터로 나눠 반환한다.

- runCV(clf, data, labels, fold=10), runCVshuffle(clf, data, labels, fold=10): 두 함수 모두 data와 labels를 shuffle하여 getxfold 함수를 통해 train용 data와 labels, test용 data와 labels로 나눈 후 해당 classifier로 cross validation을 수행한다. 또한 추가적으로 이탈을 맞게 예측한 index만 c\_index에 모아서 정확도와 함께 반환한다. 이때 Classifier의 데이터 처리 방식에 따라 코드가 약간 달라지며, runCV함수와 runCVshuffle함수의 차이는 이 부분에만 존재한다.

해당 부분의 코드는 아래와 같다. (좌: runCV, 우: runCVshuffle)

```

for x in range(len(labels_te)):
    if pred[x] == labels_te[x] & labels_te[x] == 1:
        c_index.append(x)
for x in range(len(labels_te)):
    if pred[x, 0] == labels_te[x, 0] & labels_te[x, 0] == 1:
        c_index.append(x)

```

## 2-2 Defalut값을 이용한 분류 분석 수행

-Decision Tree, K-Nearest Neighbor, Logistic Regression, Perceptron, Random Forest, Linear Discriminant Analysis, Quadratic Discriminant Analysis, Support Vector Machine, Multi-layer Perceptron를 수행한 결과는 아래와 같다.

```
>>> mean_acc_dt, mean_acc_knn, mean_acc_lr, mean_acc_pc
(0.8667926906112161, 0.5454316320100817, 0.8566477630749842, 0.6351291745431633)
>>> mean_acc_rf, mean_acc_lda, mean_acc_qda, mean_acc_lsvc, mean_acc_mlp
(0.9184625078764966, 0.8760869565217391, 0.7169187145557655, 0.74354127284184, 0.8953056080655324)
```

- Decision Tree의 결과값을 보면, 연속 값을 카테고리 변수로 범주화 하지 않았음에도 제대로 작동하고 상당히 높은 결과값이 나왔다.

- K-Nearest Neighbor 방법의 결과값이 낮게 나온 것을 확인할 수 있는데, 이는 n\_neighbors의 값의 default가 5로 크지 않은 값이라 노이즈의 영향을 많이 받았을 수 있고, weights의 default가 uniform으로 덜 중요한 attribute에도 동일한 가중치를 주었기 때문일 수 있다.

- Perceptron의 경우 single layer를 가지기 때문에 해당 데이터를 분류하는데 충분하지 않아 낮은 정확도가 나왔음을 예상할 수 있다.

- QDA보다 LDA가 더 적합하며, collinear error로 인해 QDA의 정확도가 낮은 값이 나왔음을 예상할 수 있다.

- 정확도가 높은 Random Forest와 Multi-layer-perceptron 방법을 튜닝해보았다.

## 2-3 GridSearchCV() 함수를 이용한 mlp 모델의 최적 파라미터 탐색

- Multi-layer Perceptron 모델을 cv=5로 설정하여 GridSearchCV 함수를 통해 튜닝해보았다.

### 2-3-1 내부 파라미터 조절1

- activation, hidden\_layer\_sizes, learning\_rate\_init, power\_t, max\_iter를 아래와 같이 조절하였다.

```
par1 = {'activation': ['identity', 'logistic', 'tanh', 'relu'], 'solver': ['adam'],
        'hidden_layer_sizes': np.array([100, 200, 300, 400, 500]), 'learning_rate_init': np.array([0.001, 0.01, 0.1]),
        'power_t': np.array([0.3, 0.4, 0.5, 0.6]), 'max_iter': np.array([100, 200, 300, 400, 500])}
```

- accuracy의 평균값이 0.909 이상인 결과 출력해보면 다음과 같다.

```
0.91003 for {'max_iter': 300, 'power_t': 0.3, 'learning_rate_init': 0.001, 'activation': 'logistic', 'hidden_layer_sizes': 100, 'solver': 'adam'}
0.90924 for {'max_iter': 400, 'power_t': 0.3, 'learning_rate_init': 0.001, 'activation': 'logistic', 'hidden_layer_sizes': 100, 'solver': 'adam'}
0.90968 for {'max_iter': 500, 'power_t': 0.3, 'learning_rate_init': 0.001, 'activation': 'logistic', 'hidden_layer_sizes': 100, 'solver': 'adam'}
0.90911 for {'max_iter': 500, 'power_t': 0.5, 'learning_rate_init': 0.001, 'activation': 'logistic', 'hidden_layer_sizes': 100, 'solver': 'adam'}
0.90918 for {'max_iter': 300, 'power_t': 0.5, 'learning_rate_init': 0.001, 'activation': 'logistic', 'hidden_layer_sizes': 200, 'solver': 'adam'}
0.90981 for {'max_iter': 400, 'power_t': 0.4, 'learning_rate_init': 0.001, 'activation': 'logistic', 'hidden_layer_sizes': 200, 'solver': 'adam'}
0.90937 for {'max_iter': 500, 'power_t': 0.6, 'learning_rate_init': 0.001, 'activation': 'logistic', 'hidden_layer_sizes': 200, 'solver': 'adam'}
0.90930 for {'max_iter': 300, 'power_t': 0.4, 'learning_rate_init': 0.001, 'activation': 'logistic', 'hidden_layer_sizes': 500, 'solver': 'adam'}
0.90921 for {'max_iter': 200, 'power_t': 0.5, 'learning_rate_init': 0.001, 'activation': 'tanh', 'hidden_layer_sizes': 100, 'solver': 'adam'}
```

-> activation: logistic 또는 tanh, learning\_rate\_init 모두 0.001

-> 가장 높은 정확도를 가지는 파라미터 조합은 첫번째이고, 정확도는 0.91003이다.

## 2-3-2 내부 파라미터 조절2

- activation, hidden\_layer\_sizes, learning\_rate\_init, power\_t, max\_iter를 아래와 같이 조절하였다.

```
par2 = {'activation': ['logistic', 'tanh'], 'solver': ['adam'], 'hidden_layer_sizes': np.array([50, 60, 70, 80, 90, 100]),  
        'learning_rate_init': np.array([0.0005, 0.00075, 0.001]), 'power_t': np.array([0.1, 0.2, 0.3]),  
        'max_iter': np.array([50, 60, 70, 80, 90, 100])}  
  
max(acc_means_p2)  
0.9082315877061845
```

-> 이때의 정확도의 최대값은 0.908로 2.1.1에서 수행한 결과가 더 좋다.

## 2-4 GridSearchCV() 함수를 이용한 rf 모델의 최적 파라미터 탐색

- Random Forest 모델을 cv를 5로 설정하여 GridSearchCV 함수를 통해 튜닝해보았다.

### 2-4-1 내부 파라미터 조절1

- criterion, n\_estimators, max\_features, bootstrap을 아래와 같이 조절하였다.

```
par1_rf = {'criterion': ['gini', 'entropy'], 'n_estimators': [80, 90, 100, 110, 120],  
           'max_features': ['log2', 'auto', 0.1, 0.2, 0.3], 'bootstrap': ['TRUE', 'FALSE']}
```

- accuracy의 평균값이 0.92 이상인 결과를 출력해보면 아래와 같다.

```
0.92014 for {'max_features': 0.3, 'n_estimators': 100, 'bootstrap': 'TRUE', 'criterion': 'gini'}  
0.92005 for {'max_features': 0.3, 'n_estimators': 90, 'bootstrap': 'TRUE', 'criterion': 'entropy'}  
0.92027 for {'max_features': 0.3, 'n_estimators': 100, 'bootstrap': 'FALSE', 'criterion': 'gini'}  
0.92005 for {'max_features': 0.3, 'n_estimators': 110, 'bootstrap': 'FALSE', 'criterion': 'entropy'}
```

-> max\_features는 모두 0.3이다. 가장 높은 정확도는 0.92027이다.

### 2-4-2 내부 파라미터 조절2

- 다른 파라미터는 2-4-1과 동일하게 하고, max\_features를 0.3, 0.4, 0.5, 0.6으로 수행하였다.

```
par2_rf = {'criterion': ['gini', 'entropy'], 'n_estimators': [80, 90, 100, 110, 120],  
           'max_features': [0.3, 0.4, 0.5, 0.6], 'bootstrap': ['TRUE', 'FALSE']}
```

- accuracy의 평균값이 0.9208 이상인 결과를 출력해보면 아래와 같다.

```
0.92105 for {'max_features': 0.6, 'n_estimators': 90, 'bootstrap': 'TRUE', 'criterion': 'entropy'}  
0.92096 for {'max_features': 0.6, 'n_estimators': 110, 'bootstrap': 'TRUE', 'criterion': 'entropy'}  
0.92108 for {'max_features': 0.6, 'n_estimators': 120, 'bootstrap': 'FALSE', 'criterion': 'gini'}  
0.92080 for {'max_features': 0.6, 'n_estimators': 90, 'bootstrap': 'FALSE', 'criterion': 'entropy'}  
0.92080 for {'max_features': 0.6, 'n_estimators': 100, 'bootstrap': 'FALSE', 'criterion': 'entropy'}
```

-> max\_features는 모두 0.6이다. 가장 높은 정확도는 0.92108이다.

### 2-4-3 내부 파라미터 조절3

- 다른 파라미터는 2-4-1과 동일하게 하고, max\_features를 0.7, 0.8, 0.9로 수행하였다.

```
par3_rf = {'criterion': ['gini', 'entropy'], 'n_estimators': [80, 90, 100, 110, 120],
           'max_features': [0.7, 0.8, 0.9], 'bootstrap': [True, False]}
```

- accuracy의 평균값이 0.9208 이상인 결과를 출력해보면 아래와 같다.

```
0.92102 for {'max_features': 0.7, 'n_estimators': 120, 'bootstrap': True, 'criterion': 'entropy'}
0.92083 for {'max_features': 0.8, 'n_estimators': 90, 'bootstrap': True, 'criterion': 'entropy'}
0.92090 for {'max_features': 0.8, 'n_estimators': 110, 'bootstrap': True, 'criterion': 'entropy'}
0.92080 for {'max_features': 0.7, 'n_estimators': 110, 'bootstrap': False, 'criterion': 'entropy'}
0.92105 for {'max_features': 0.8, 'n_estimators': 120, 'bootstrap': False, 'criterion': 'entropy'}
```

- > 가장 높은 정확도는 0.92105로 2-4-2의 결과보다는 좋지 않다.

## 2-5 결론

- Random Forest 방법을 이용하고 내부 파라미터를 max\_features=0.6, n\_estimators=120, bootstrap='FALSE', criterion='gini'로 설정한 결과가 정확도 0.92108로 가장 좋다.

## 3. 고객의 특성 분석

### 3-1 추가 함수 정리

- 'UtilForMid'에 사용할 함수를 정리하였다.

- featurenum(findcol, c\_index, kind): 카테고리형 변수 중 하나를 findcol, 이탈을 옳게 예측한 인덱스 정보가 담긴 c\_index(runCVshuffle함수를 통해 받아온), 카테고리형 변수의 종류의 개수를 kind로 넘겨 맞게 예측된 데이터와 종류별개수를 반환한다.

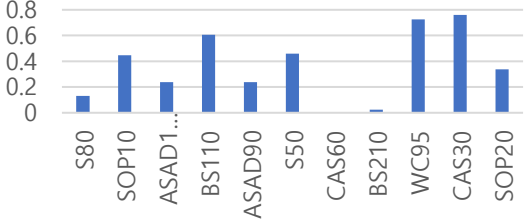
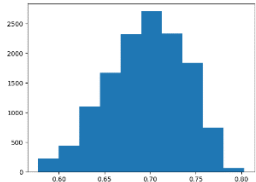
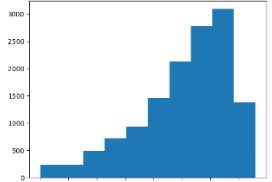
- runCVshuffle 함수를 사용하여 옳게 예측된 인덱스의 정보를 받아올 때는 GridSearchCV에서 cross validation을 5회 수행한 것과 달리 10회를 수행하며 함수의 작동 방식이 다르기 때문에 약간의 오차는 발생할 수 있다.

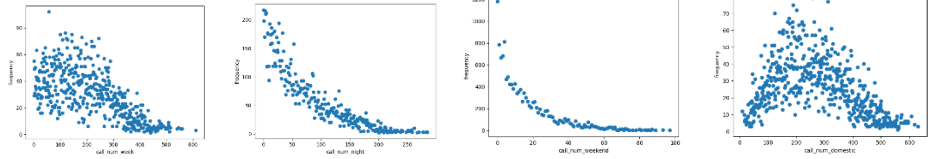
- 이진화 하지 않은 이산형과 연속형의 경우 c\_index를 통해 직접 특징을 찾아보았다.

### 3-2 이탈 고객의 특성

Columns name	특성	기존 대비
성별	성별과 무관하다.	
연령	80대의 이탈0 but, 데이터가 적다.	80대를 제외하고 모든 연령대의 약 40% 정도가 이탈
단선횟수	단선횟수 0,1,2가 대부분의 경우를 차지 -> 원래 데이터가 많아서	모든 횟수가 기존 대비 약 35%에서 53% 정도의 이탈을 보임.



		단선횟수 12회인 경우 약 53%의 이탈율로 가장 높음
요금제	모두 'CAT 200'을 사용	기존 'CAT 200' 요금제를 사용하는 고객의 98%가 이탈
핸드셋	기존 대비 이탈 비율은 오른쪽과 같으며, 'CAS30', 'WC95', 'BS110'의 이탈 비율은 60% 이상이다. 'CAS60'의 경우 아무도 이탈하지 않았다.	<p>이탈 비율</p> 
주간통화시간_분		기존 그래프 대비 0.6~0.8 구간에 데이터가 몰려 있다. -> 주간통화시간이 상위 60%에서 80%인 사용자의 이탈이 많다.
국제통화시간_분		기존 그래프 대비 분포가 오른쪽으로 갈수록 데이터가 많아진다. -> 국제통화시간이 많은 경우의 이탈이 많다.
통화량구분	이탈은 통화량이 '중', '중저'에서만 발생하였으며, 각각 99%, 1%를 차지한다.	통화량이 '중'인 경우 기존 대비 약 80%가 이탈하였다. 통화량이 '중저'인 경우 기존 대비 약 0.03%이탈하였으며
납부여부	이탈은 'OK'와 'High CAT 100'에서만 발생하였으며, 각각 99%, 1%를 차지한다.	'OK'는 기존 대비 약 43%가 이탈하였고, 'High CAT 100'은 기존 대비 약 24%가 이탈하였다.
서비스기간 야간통화시간 주말통화시간 분당통화요금 국제통화비율		기존의 분포와 비슷한 모양을 가짐 -> 뚜렷한 특징이 보이지 않음
통화품질불만	약 0.08%는 True에서 약 92%는 False에서 이탈이 발생하였다.	True는 기존 대비 약 51%가 이탈하였고, False는 기존 대비 약 41%가 이탈하였다.

Columns name	특성
주간통화횟수 야간통화횟수 주말통화횟수 국내통화횟수	 <p>주간/야간/주말의 경우 통화 횟수가 증가할수록 이탈이 줄어드는 경향이 있다. 국내통화횟수의 경우 통화 횟수가 증가할수록 이탈이 증가하다가 약 200회에서 정점을 찍고 점점 이탈이 줄어드는 경향이 있다.</p>