Mobile Price Classification

Saeid Cheshmi - 98222027

1. Data Exploration

Mobile price dataset has 2000 rows and 21 columns where each column is a feature of a mobile (e.g RAM, battery power, …). Our goal here is predict price range of a mobile.

The dataset has the following attributes:

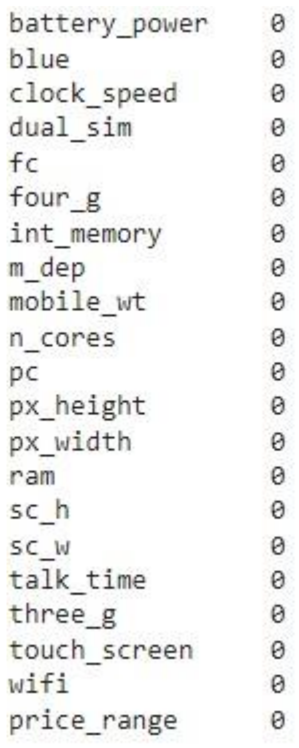
- **"id"**: id of each record
- **"battery_power"**: total energy a battery can store in one time measured in mAh
- **"blue"**: has bluetooth or not
- **"clock_speed"**: speed at which microprocessor executes instructions
- **"dual_sim"**: has dual sim support or not
- **"fc"**: front camera mega pixels
- **"four_g"**: has 4G or not
- **"int_memory"**: internal memory in gigabytes
- **"m_dep"**: mobile depth in cm
- **"mobile_wt"**: weight of mobile phone
- **"n_cores"**: cumber of cores of processor
- **"pc"**: primary camera mega pixels
- **"px_height"**: pixel resolution height
- **"px_width"**: pixel resolution width
- **"ram"**: random access memory in megabytes
- **"sc_h"**: screen height of mobile in cm
- **"sc_w"**: screen width of mobile in cm
- **"talk_time"**: longest time that a single battery charge will last when you are
- **"three_g"**: has 3G or not
- **"touch_screen"**: has touch screen or not
- **"wifi"**: has wifi or not

Also, our target feature (price_range) has 4 different classes (0,1,2,3) which we want to predict.

2. Preprocessing

We preprocess our dataset in following steps

1) **Null values:** As you can see in figure 1, we don't have any null value in the dataset.
2) **Outliers:** Outliers were dropped from the dataset. We first detected outliers by computing z-score, then deleted them. By using this method 99.7% of each feature are considered as normal record and rest of them are considered as outlier.



```
battery_power    0
blue             0
clock_speed      0
dual_sim         0
fc               0
four_g           0
int_memory       0
m_dep            0
mobile_wt        0
n_cores          0
pc               0
px_height        0
px_width         0
ram              0
sc_h             0
sc_w             0
talk_time        0
three_g          0
touch_screen     0
wifi             0
price_range      0
```

Figure 1. Number of null values for

each feature

3. Exploratory Data Analysis

3.1. Target Distribution

At first, we look at target distribution to find out the dataset is imbalanced or not? The Fig.2 shows that the dataset has around 500 records of each class so, the dataset is balanced.

But if the dataset isn't balanced, what should we do??

3 possible solutions are mentioned as following:

1) Proper Evaluation Metric

   Accuracy of a classifier is the number of correct predictions by classifier divided by total number of predictions. But this is good for a balanced dataset not for imbalanced one. There are other metrics like **precision** which measures how accurately the prediction of a specific class and **recall** which measures the classifier's ability to detect a class. There is also another metric, **F1-Score**, the harmonic mean of precision and recall, which is more appropriate here. F1-score improves when classifier identifies more of a certain class correctly.

2) Resampling (Oversampling and Undersampling)

   This method is used to downsample or upsample the majority or minority class. When we dealing with imbalanced dataset, we can upsample minority class with replacement, this method is called oversampling. Also, we can delete randomly records of majority class to have equal size with minority class. This method is called undersampling. When the dataset has balanced classes, we can assume that classifier will give equal importance to both classes.

3) SMOTE

   Synthetic Minority Oversampling Technique is another method to oversample minority class. Instead of adding duplicate sample to increase minority class, new records are synthesized from the existing data. SMOTE looks in minority class and uses k nearest neighbor to select a random nearest neighbor to synthesize an instance.

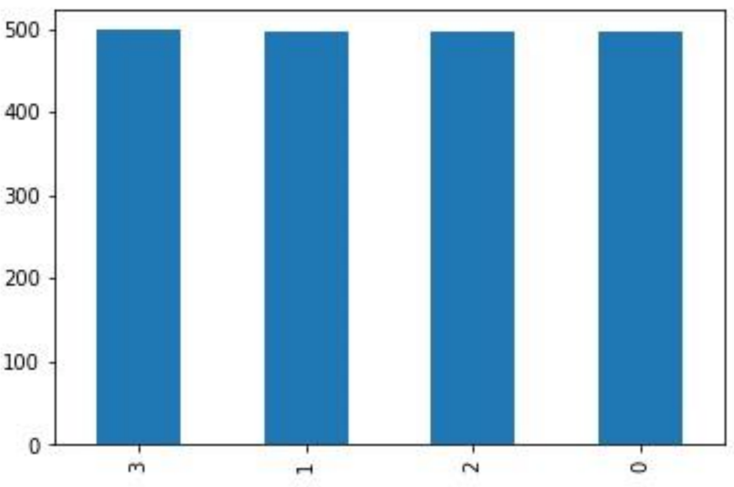


Figure 2. Distribution of target feature

3.2. Distributions of Numerical Features

According to figure 3, it can be seen that none of feature has normal distribution. Also records in battery_power, int_memory, n_cores, pc, mobile_wt, px_width, talk_time, sc_h and ram are approximately equally distributed.

Most of records in the dataset have 0.5 of clock speed and number of records with other clock speed are almost equal. Majority of mobile in the dataset doesn’t have front camera or has an ordinary front camera.

Px_height and sc_w have skewed distribution, most of mobiles in the dataset have small height and low width resolution.

3.3. Distributions of Categorical Features

According to figure 4 , it can be seen that all categorical features of the dataset exept 3G , are equally distributed. In the dataset nearly 75% of mobiles have 3G and other don’t.

As figure 5 shows also in each class of price range categorical feature are distributed equally.

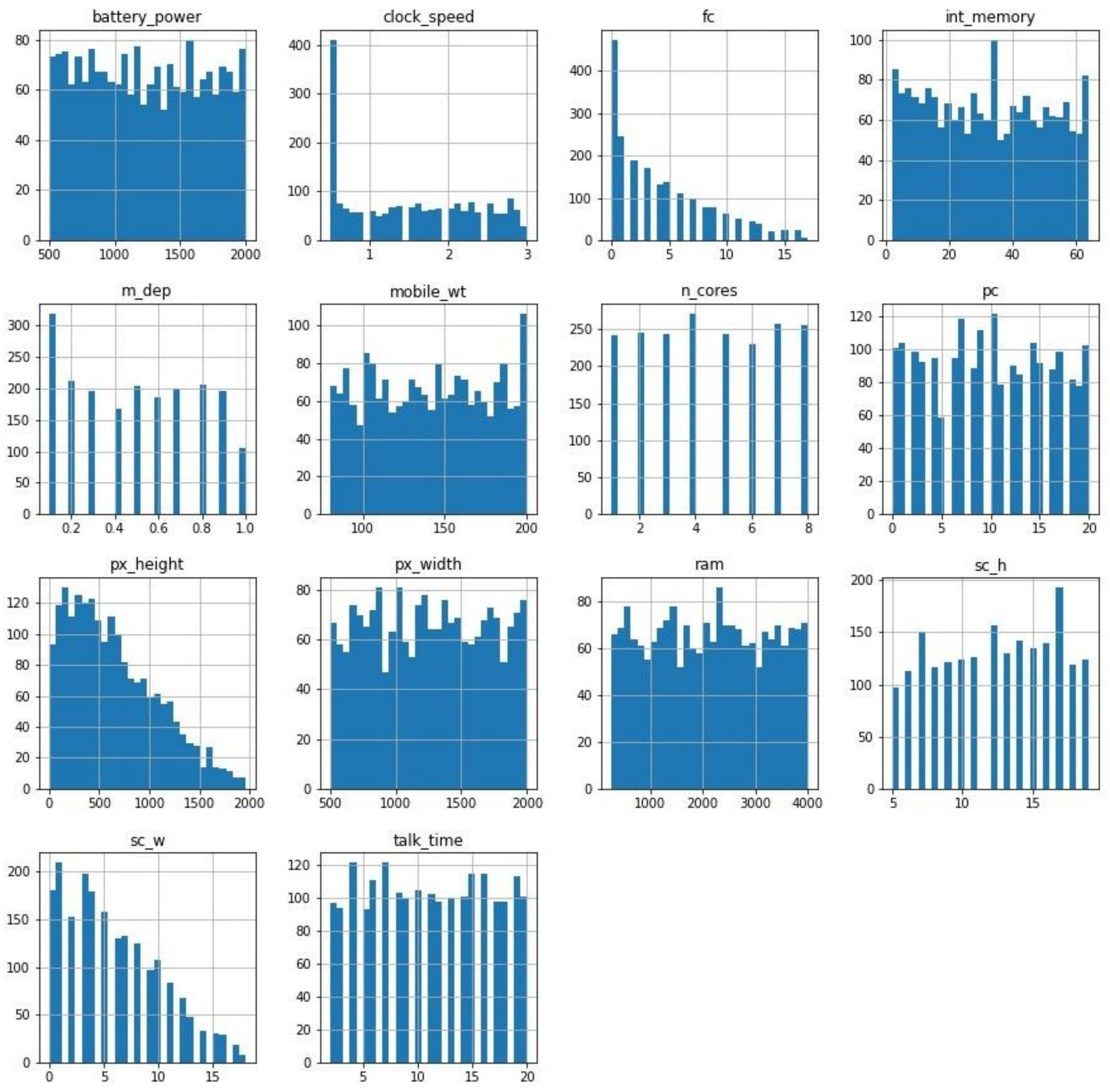


Figure 3. Histograms of numerical features

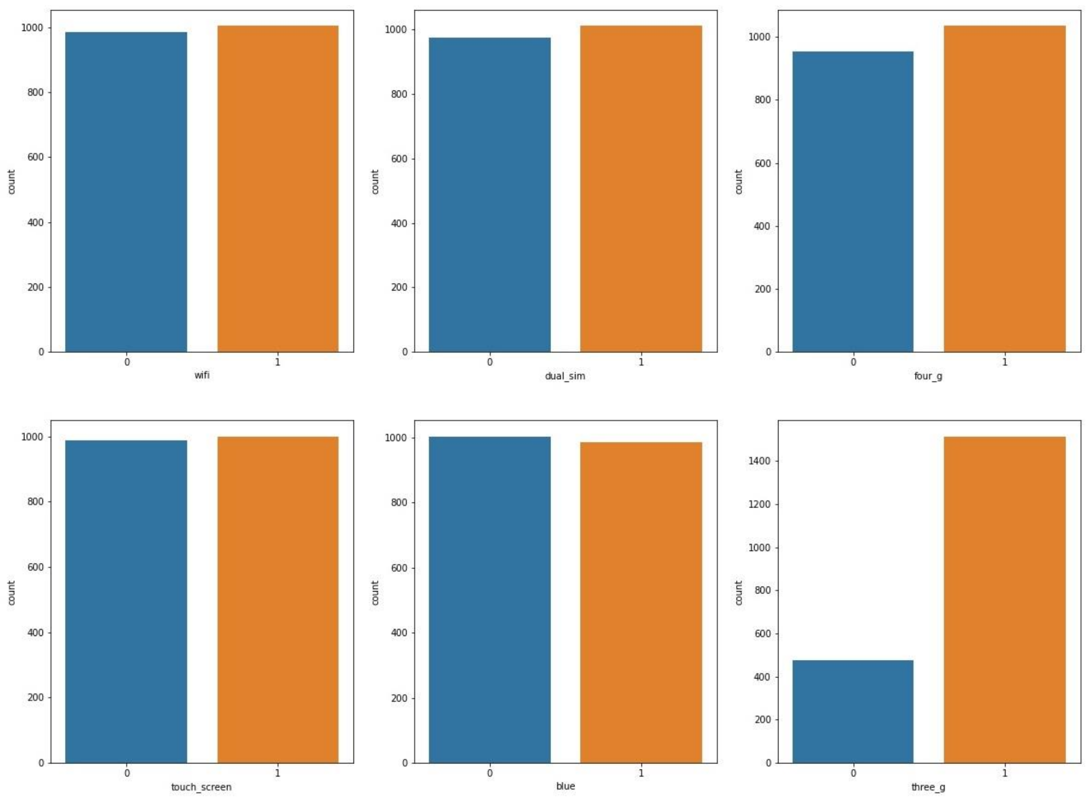


Figure 4. Distribution of categorical features

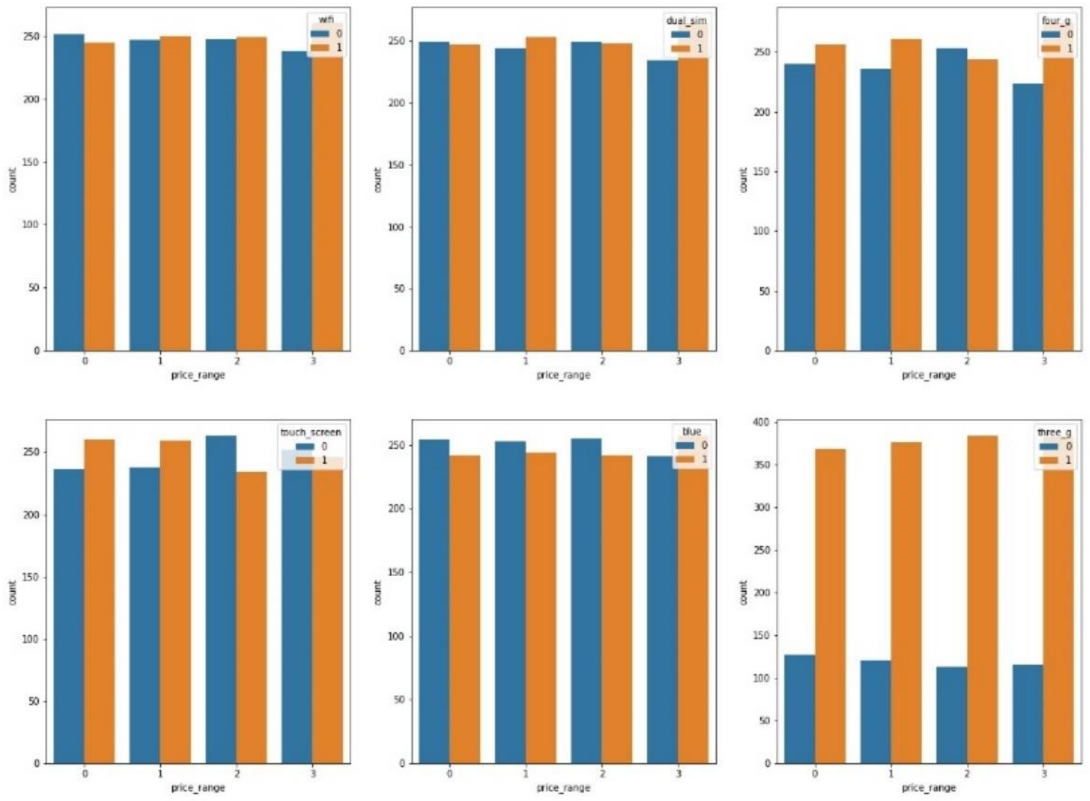


Figure 5. Distribution of categorical features based on price range

3.4. Relationship between ram and battery power

Figure 6 shows relationship between ram and battery power, as it can be seen by increasing the amount of ram battery power increases, too. Also, price range of the mobile with much ram and high battery power belongs to class 3 of price range, expensive mobile.

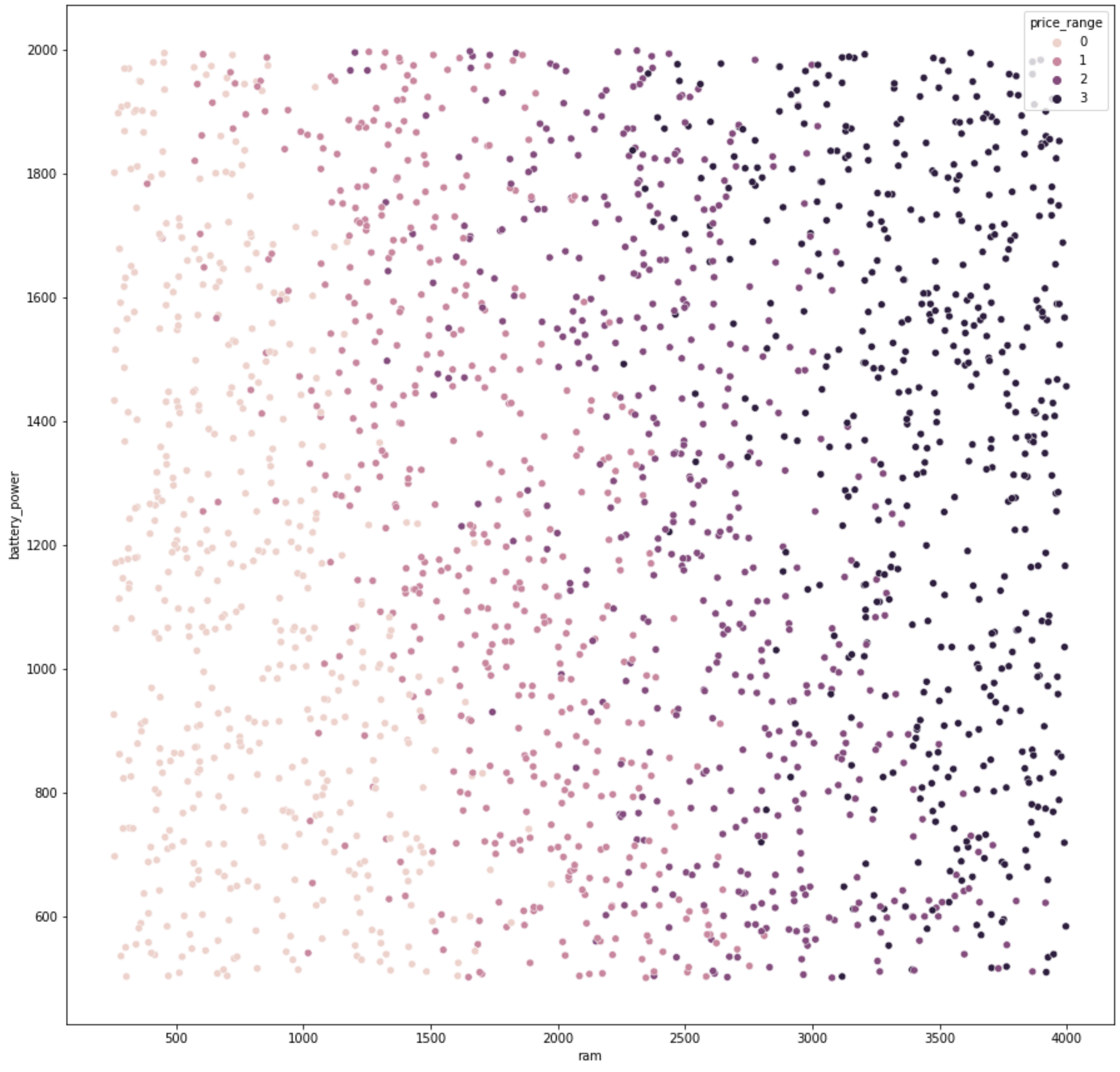


Figure 6. Scatter plot between ram and battery power

3.5. Relationship between ram and price range

Below plot shows that mobiles with higher price have much more amount of ram. As you can see expensive mobiles belong to higher range of ram.

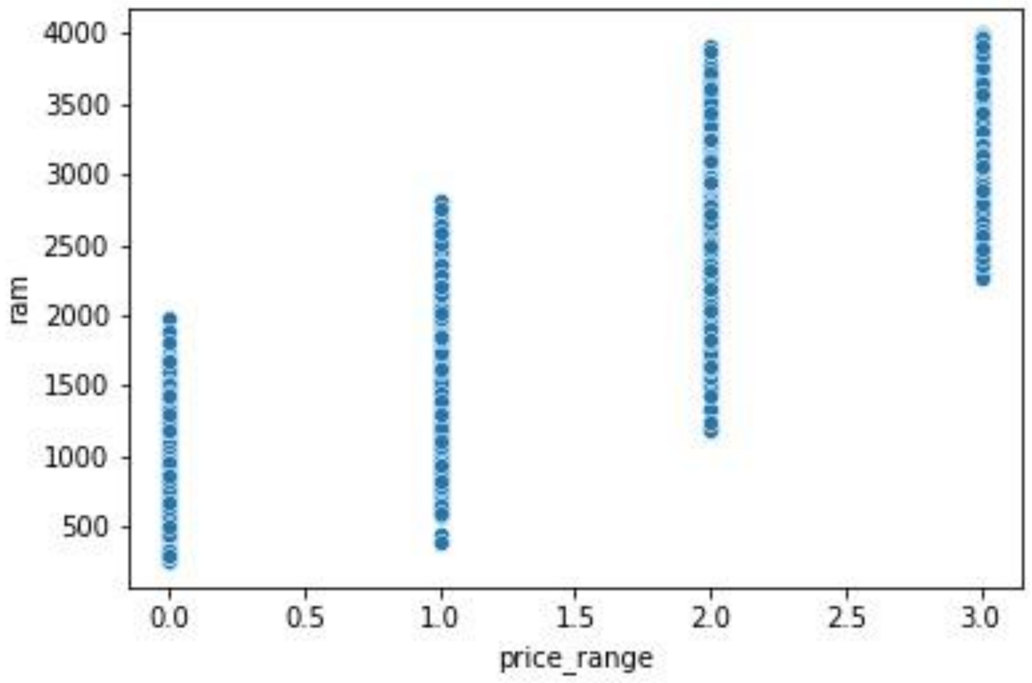


Figure 7. Scatter plot between ram and price range

3.6. Relationship between battery power and price range

As figure 8 shows, by going the price of mobile from class 0 to 1, power of battery increases, also class 1 and 2 have almost the same battery power, but like first part, by going the price from class 2 to 3 battery power increases.

3.7. Relationship between height and width and price range

According to figure 9, increasing height and width resolution leading to higher price. Also, mobiles which belong to higher class have higher range of resolution.

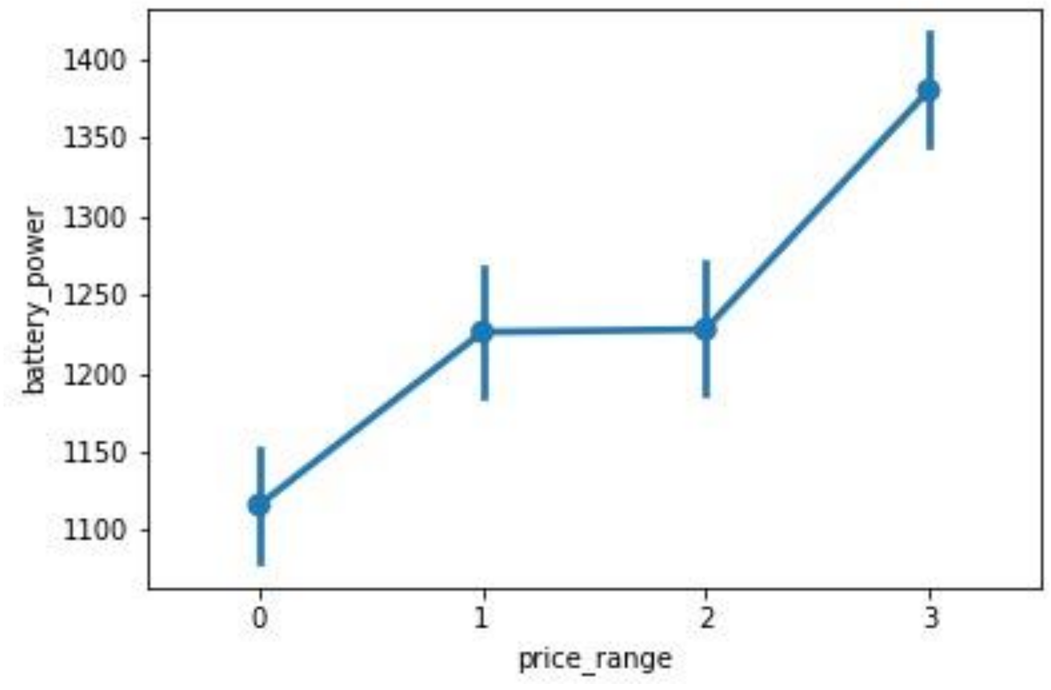


Figure 8. Point plot between price range and battery power

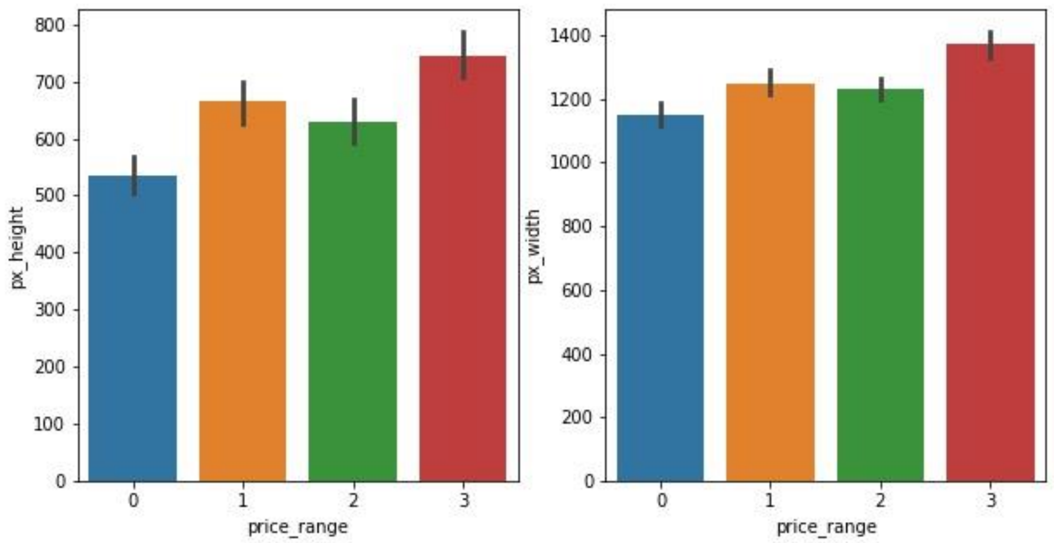
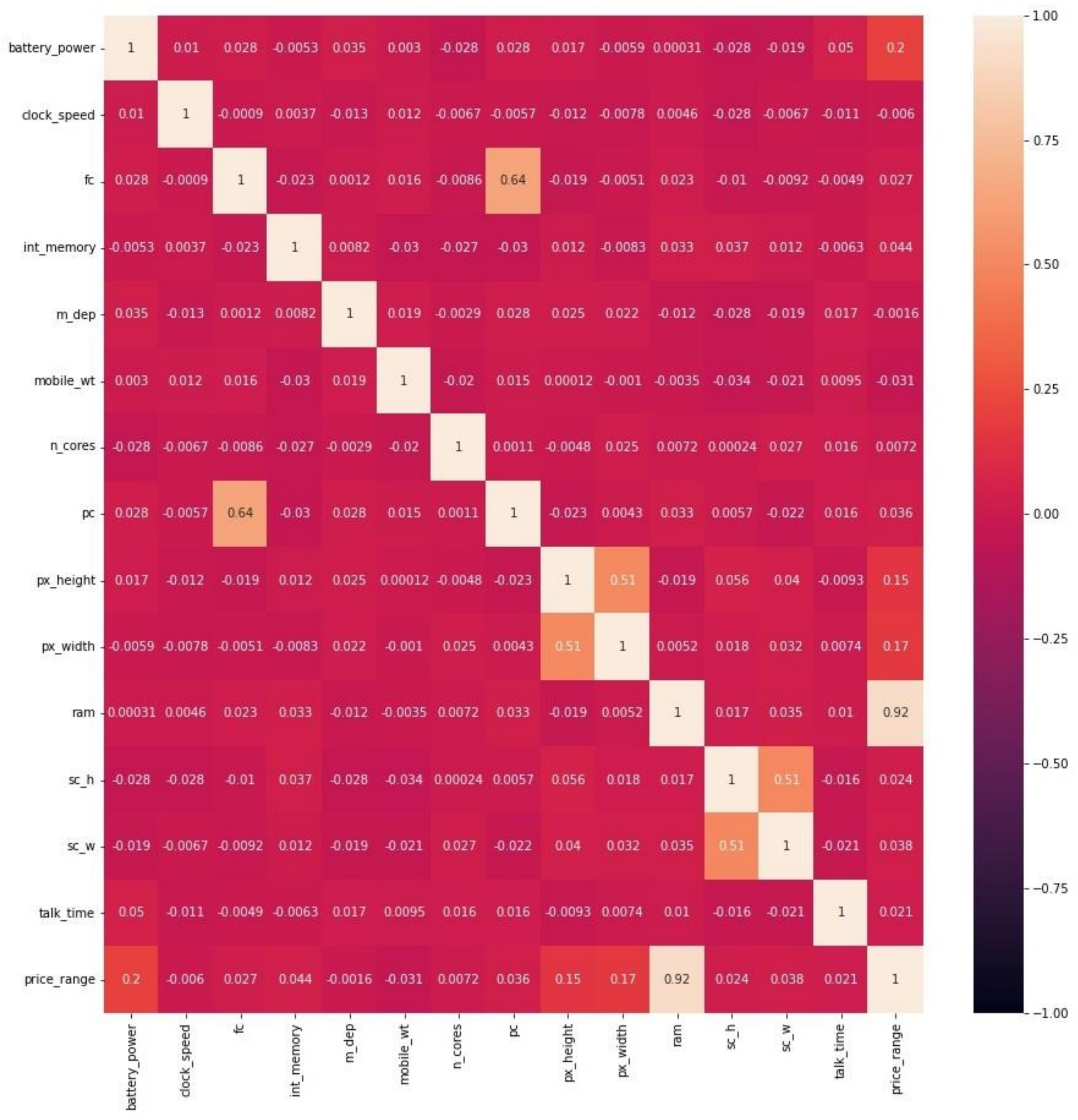


Figure 9. Bar plot of height and width resolution based on price range

3.8. Correlation matrix

As it can be seen from correlation matrix there are some strong correlation between some numerical features, Specially with price range. Ram is strongly correlated with price range. As a result, applying a linear model might be led to good result.



	battery_power	clock_speed	fc	int_memory	m_dep	mobile_wt	n_cores	pc	px_height	px_width	ram	sc_h	sc_w	talk_time	price_range
battery_power	1	0.01	0.028	-0.0053	0.035	0.003	-0.028	0.028	0.017	-0.0059	0.00031	-0.028	-0.019	0.05	0.2
clock_speed	0.01	1	-0.0009	0.0037	-0.013	0.012	-0.0067	-0.0057	-0.012	-0.0078	0.0046	-0.028	-0.0067	-0.011	-0.006
fc	0.028	-0.0009	1	-0.023	0.0012	0.016	-0.0086	0.64	-0.019	-0.0051	0.023	-0.01	-0.0092	-0.0049	0.027
int_memory	-0.0053	0.0037	-0.023	1	0.0082	-0.03	-0.027	-0.03	0.012	-0.0083	0.033	0.037	0.012	-0.0063	0.044
m_dep	0.035	-0.013	0.0012	0.0082	1	0.019	-0.0029	0.028	0.025	0.022	-0.012	-0.028	-0.019	0.017	-0.0016
mobile_wt	0.003	0.012	0.016	-0.03	0.019	1	-0.02	0.015	0.00012	-0.001	-0.0035	-0.034	-0.021	0.0095	-0.031
n_cores	-0.028	-0.0067	-0.0086	-0.027	-0.0029	-0.02	1	0.0011	-0.0048	0.025	0.0072	0.00024	0.027	0.016	0.0072
pc	0.028	-0.0057	0.64	-0.03	0.028	0.015	0.0011	1	-0.023	0.0043	0.033	0.0057	-0.022	0.016	0.036
px_height	0.017	-0.012	-0.019	0.012	0.025	0.00012	-0.0048	-0.023	1	0.51	-0.019	0.056	0.04	-0.0093	0.15
px_width	-0.0059	-0.0078	-0.0051	-0.0083	0.022	-0.001	0.025	0.0043	0.51	1	0.0052	0.018	0.032	0.0074	0.17
ram	0.00031	0.0046	0.023	0.033	-0.012	-0.0035	0.0072	0.033	-0.019	0.0052	1	0.017	0.035	0.01	0.92
sc_h	-0.028	-0.028	-0.01	0.037	-0.028	-0.034	0.00024	0.0057	0.056	0.018	0.017	1	0.51	-0.016	0.024
sc_w	-0.019	-0.0067	-0.0092	0.012	-0.019	-0.021	0.027	-0.022	0.04	0.032	0.035	0.51	1	-0.021	0.038
talk_time	0.05	-0.011	-0.0049	-0.0063	0.017	0.0095	0.016	0.016	-0.0093	0.0074	0.01	-0.016	-0.021	1	0.021
price_range	0.2	-0.006	0.027	0.044	-0.0016	-0.031	0.0072	0.036	0.15	0.17	0.92	0.024	0.038	0.021	1

Figure 10. Correlation matrix

4. Hypothesis tests

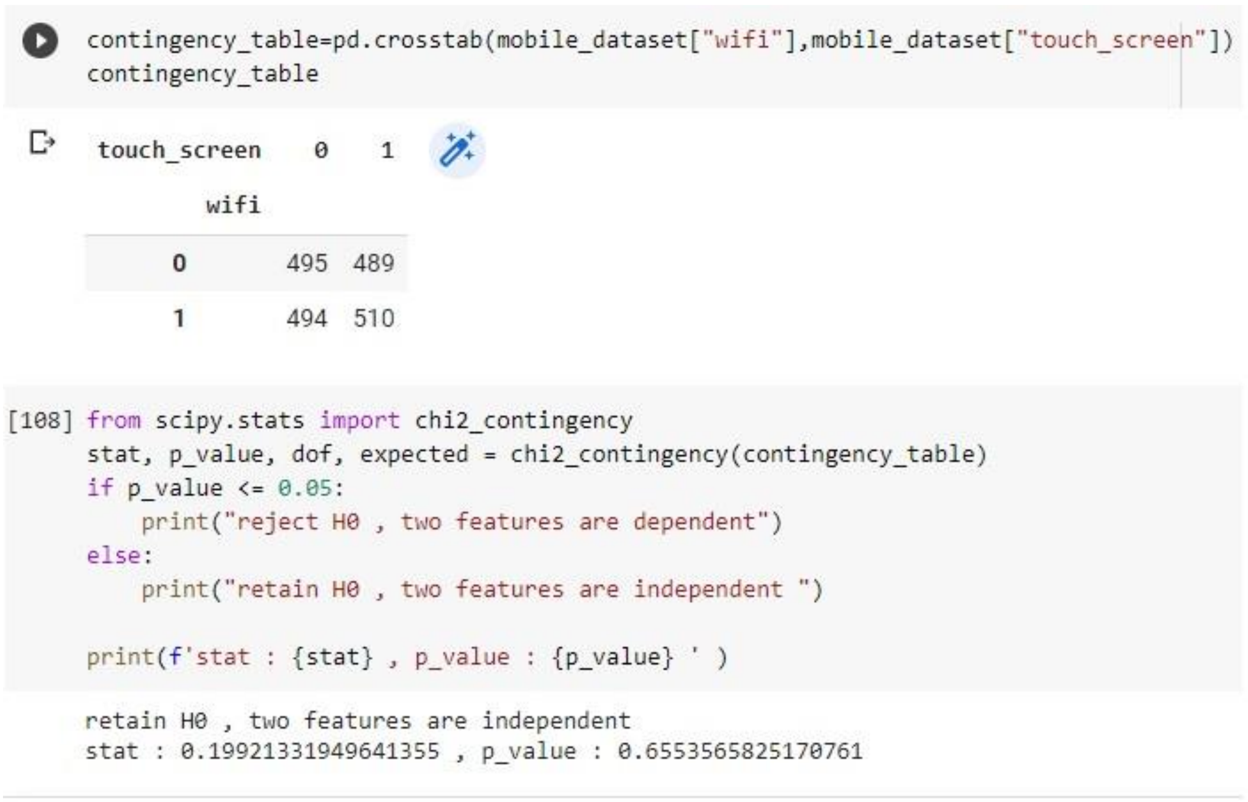
4.1. Relation between two categorical features wifi and touch screen

We use Chi-Squared Test to test whether two categorical variables are related or independent. For applying Chi-Squared test, first we have to calculate contingency table.

H0: the two features are independent

H1: the two features are dependent

Then we calculate p-value and statistic for comparison, after that if p-value is less than or equal to significance level (0.05 here), H0 will be rejected otherwise H0 will be retained.



```
contingency_table=pd.crosstab(mobile_dataset["wifi"],mobile_dataset["touch_screen"])
contingency_table
```

touch_screen	0	1
wifi		
0	495	489
1	494	510

```
[108] from scipy.stats import chi2_contingency
      stat, p_value, dof, expected = chi2_contingency(contingency_table)
      if p_value <= 0.05:
          print("reject H0 , two features are dependent")
      else:
          print("retain H0 , two features are independent ")

      print(f'stat : {stat} , p_value : {p_value} ' )
```

```
retain H0 , two features are independent
stat : 0.19921331949641355 , p_value : 0.6553565825170761
```

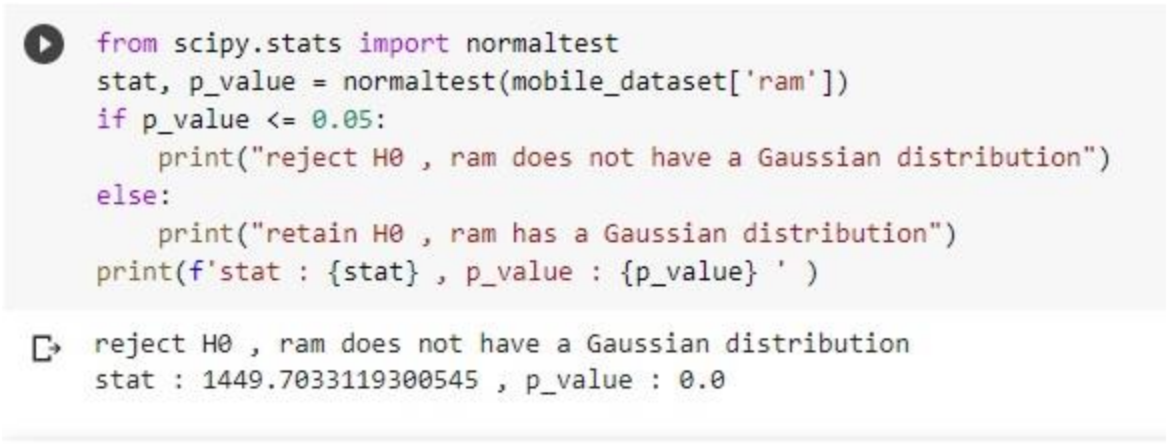
Figure 11. Chi-Squared Test between wifi and touch screen

As above figure shows, H0 retains and two features are independent.

4.2. Ram feature has Gaussian distribution or not

We use D'Agostino's K^2 Test to test whether a sample has Gaussian distribution or not. Like previous section we have to calculate p-value and statistic. Then we calculate p-value and statistic for comparison, after that if p-value is less than or equal to significance level (0.05 here), H0 will be rejected otherwise H0 will be retain.

H0: ram has a Gaussian distribution.
H1: ram does not have a Gaussian distribution.



```python
from scipy.stats import normaltest
stat, p_value = normaltest(mobile_dataset['ram'])
if p_value <= 0.05:
    print("reject H0 , ram does not have a Gaussian distribution")
else:
    print("retain H0 , ram has a Gaussian distribution")
print(f'stat : {stat} , p_value : {p_value} ' )
```

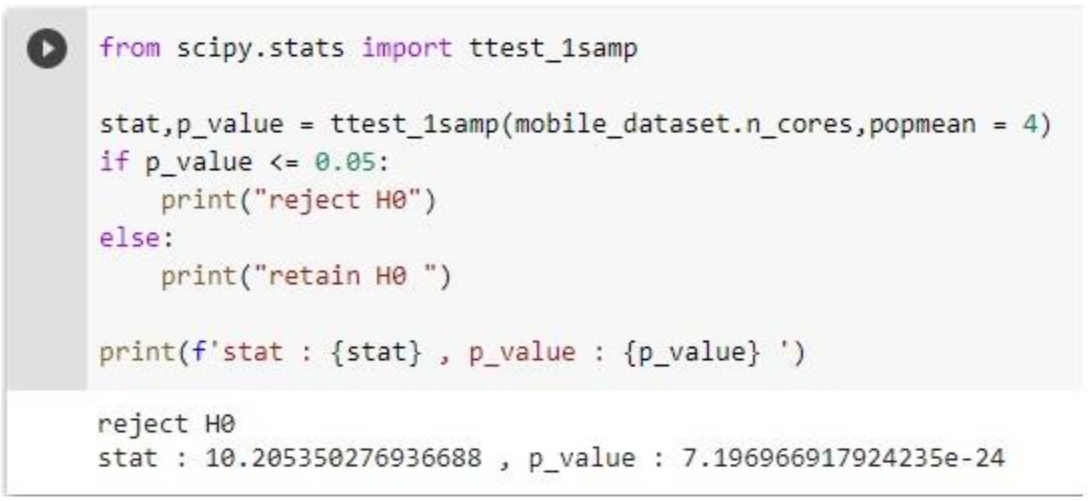
```
reject H0 , ram does not have a Gaussian distribution
stat : 1449.7033119300545 , p_value : 0.0
```

Figure 12. D'Agostino's K^2 Test for ram

As you can see, H0 rejected and ram does not have a Gaussian distribution.

4.3. Phones have average 4 cpu cores or not

We use One sample T-Test to test the mean of a sample is equal to mean of population. We already hypothesized the mean and compare it to the sample mean. Like previous sections we need to calculate p-value and t-statistic. If p-value is less than or equal to significance level (0.05 here), H0 will be rejected otherwise H0 will be retain.
H0: mean of n_cores is 4
H1: mean of n_cores is not 4


```
from scipy.stats import ttest_1samp

stat,p_value = ttest_1samp(mobile_dataset.n_cores,popmean = 4)
if p_value <= 0.05:
    print("reject H0")
else:
    print("retain H0 ")

print(f'stat : {stat} , p_value : {p_value} ')
```

```
reject H0
stat : 10.205350276936688 , p_value : 7.196966917924235e-24
```

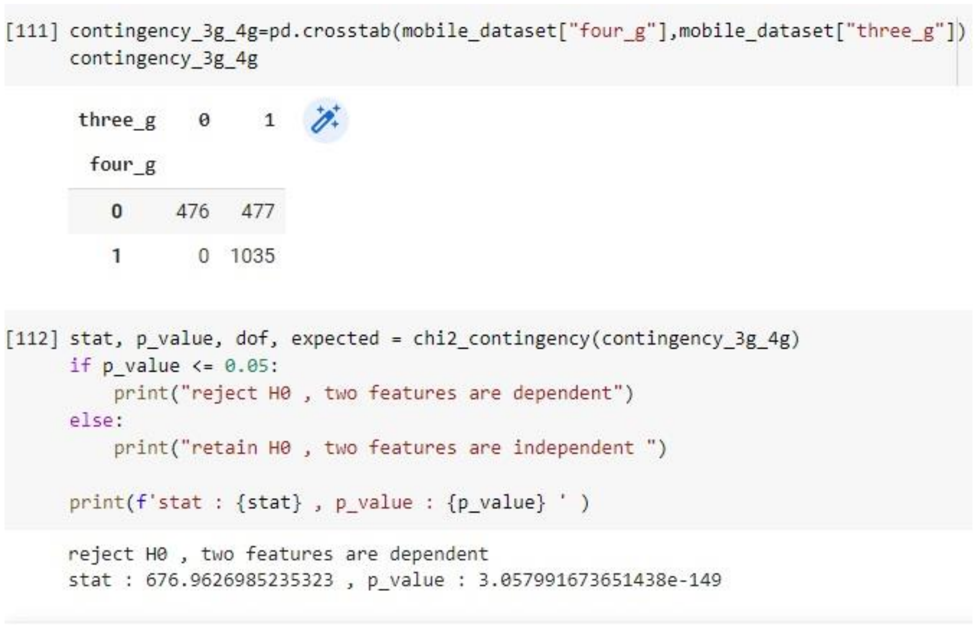
Figure 13. One sample T-Test for n cores

As above figure shows, H0 rejected and mean of n_cores isn't 4.

4.4. Relation between two categorical features 3G and 4G

Like 4.1, we use Chi-Squared Test here, too.

H0: the two features are independent
H1: the two features are dependent


```
[111] contingency_3g_4g=pd.crosstab(mobile_dataset["four_g"],mobile_dataset["three_g"])
      contingency_3g_4g
```

three_g	0	1
four_g		
0	476	477
1	0	1035

```
[112] stat, p_value, dof, expected = chi2_contingency(contingency_3g_4g)
      if p_value <= 0.05:
          print("reject H0 , two features are dependent")
      else:
          print("retain H0 , two features are independent ")

      print(f'stat : {stat} , p_value : {p_value} ' )
```

```
reject H0 , two features are dependent
stat : 676.9626985235323 , p_value : 3.057991673651438e-149
```

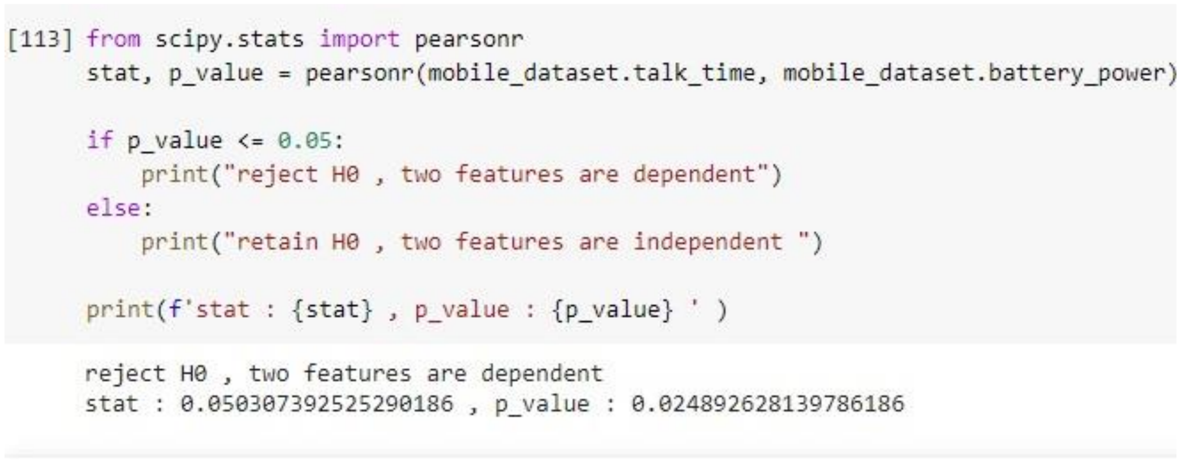
Figure 14. Chi-Squared Test for 3G and 4G

As you can see H0 rejected and two features are dependent.

4.5 Are talk time and battery power related?

We use Pearson's Correlation Coefficient to test whether two sample have a linear relationship or not. Like other section, we need to calculate p-value and statistic and then comparing p-value to significance level (0.05) to reject or retain H0.

H0: the two features are independent.
H1: the two features are dependent.



```
[113] from scipy.stats import pearsonr
      stat, p_value = pearsonr(mobile_dataset.talk_time, mobile_dataset.battery_power)

      if p_value <= 0.05:
          print("reject H0 , two features are dependent")
      else:
          print("retain H0 , two features are independent ")

      print(f'stat : {stat} , p_value : {p_value} ' )

      reject H0 , two features are dependent
      stat : 0.050307392525290186 , p_value : 0.024892628139786186
```

Figure 15. Pearson's Correlation Coefficient between talk time and battery power

As above figure shows H0 rejected and two features are dependent!!

5. Model Training

5.1. Train Test Split

We split our dataset into none-overlapping two datasets, train and test dataset, as test dataset include 20% of whole dataset and rest of it considers as train dataset. Then train our model on train set and report its performance on test set.

5.2. Pipeline

We define a pipeline which consist of two parts, preprocessor and logistic regression. By default we set preprocessor to StandardScaler. We use pipelines for ease of use and in future sections we will use it for hyperparameter tunning. We use 'saga' optimizer because it works with two kinds of penalty, l1 and l2.

5.3. Hyperparameter tuning and trying different scalers

We tune our hyperparameters using GridSearchCV which trying any combination of given hyperparameters and using cross validation to report test score for each set of hyperparameters. Also, we want to try different scalers and also train model without any scaler, for this purpose we can also use GridSearchCV. In preprocessor part, we will test StandardScaler, MinMaxScaler and none which means we don't use any scaler. For logistic regression, we will try following values for C and penalty:
C: 0.01, 0.1, 1, 10, 100
Penalty: l1, l2
We tune our hyperparameters using train set and when we find out best combination of them, we use them to predict our test set.
StandardScaler, subtracts feature mean from each feature and divides on standard deviation, after that sample mean is 0 and feature has unit variance, by doing so we center and scale each feature.
MinMaxScaler, subtracts feature min from each feature and divides on maximum of feature minus minimum of feature. By doing so, all features will be transformed into the range [0,1].
Penalty, defines the regularization term, l1 is norm l1 and l2 is norm l2. Regularization prevents model from overfitting. It causes coefficients to shrink toward zero.
C, is regularization coefficient, smaller value specify stronger regularization. In sklearn by default it is set to 1.

5.4. Is our model OVO or OVA?

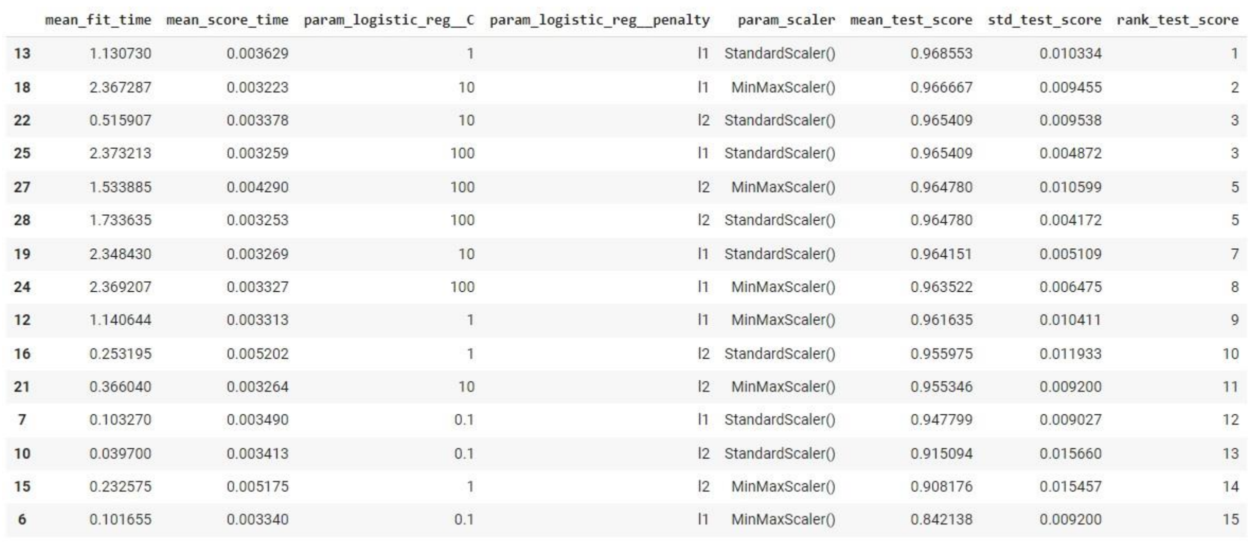
In logistic regression class in sklearn, mult_class argument specifies which strategy our model selects to deal with multi-class problem. By default is set to 'auto' which selects 'ovr' (OVA) for binary problems, or when solver is 'liblinear', and otherwise selects 'multinomial'. So, if we want to use OVO strategy we should use meta estimators.
OVA (or OVR) strategy: In this strategy multi-class dataset splitted into multiple binary class classification problem. A binary classifier is then trained on each dataset and prediction is made by the model is the most confident. In this approach we need one model for each class and this may be an issue for large datasets.

OVO strategy: Like ova, this strategy multi-class dataset splitted into multiple binary class classification problem. But it splits dataset into one dataset for each pair of classes. Then each binary classifier predicts one class label and model with the most predictions or votes is predicted by the strategy. As you know, in this model we should train more model than previous one, but this approach suggested for svm and kernel-based algorithms.

Multinomial: An alternative approach for multi-class problem is modify the logistic regression problem to support prediction of multiple class labels directly. Specifically, to predict the probability that an input example belongs to each known class label. Changing logistic regression to supports multinomial probability requires a change in loss function, from log loss to cross-entropy and a change in output from single value probability to one value for each class.

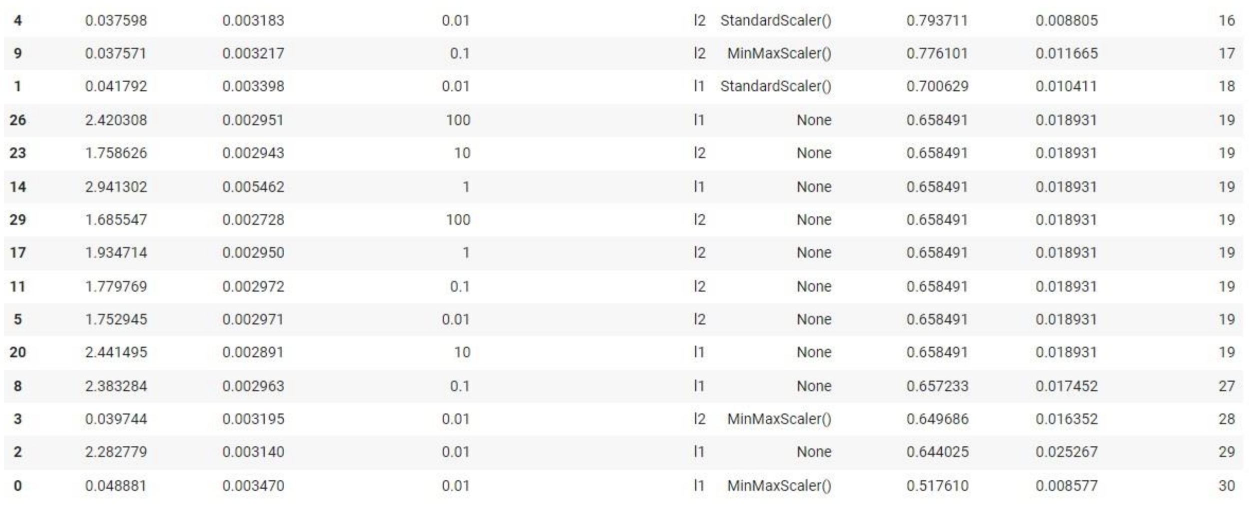
5.5. Hyperparameter tuning results

After running GridSearchCV for hyperparameter tuning, we got the following results:



	mean_fit_time	mean_score_time	param_logistic_reg__C	param_logistic_reg__penalty	param_scaler	mean_test_score	std_test_score	rank_test_score
13	1.130730	0.003629	1	l1	StandardScaler()	0.968553	0.010334	1
18	2.367287	0.003223	10	l1	MinMaxScaler()	0.966667	0.009455	2
22	0.515907	0.003378	10	l2	StandardScaler()	0.965409	0.009538	3
25	2.373213	0.003259	100	l1	StandardScaler()	0.965409	0.004872	3
27	1.533885	0.004290	100	l2	MinMaxScaler()	0.964780	0.010599	5
28	1.733635	0.003253	100	l2	StandardScaler()	0.964780	0.004172	5
19	2.348430	0.003269	10	l1	StandardScaler()	0.964151	0.005109	7
24	2.369207	0.003327	100	l1	MinMaxScaler()	0.963522	0.006475	8
12	1.140644	0.003313	1	l1	MinMaxScaler()	0.961635	0.010411	9
16	0.253195	0.005202	1	l2	StandardScaler()	0.955975	0.011933	10
21	0.366040	0.003264	10	l2	MinMaxScaler()	0.955346	0.009200	11
7	0.103270	0.003490	0.1	l1	StandardScaler()	0.947799	0.009027	12
10	0.039700	0.003413	0.1	l2	StandardScaler()	0.915094	0.015660	13
15	0.232575	0.005175	1	l2	MinMaxScaler()	0.908176	0.015457	14
6	0.101655	0.003340	0.1	l1	MinMaxScaler()	0.842138	0.009200	15

Figure 16. Result of hyperparameter tuning part-1



4	0.037598	0.003183	0.01	l2	StandardScaler()	0.793711	0.008805	16
9	0.037571	0.003217	0.1	l2	MinMaxScaler()	0.776101	0.011665	17
1	0.041792	0.003398	0.01	l1	StandardScaler()	0.700629	0.010411	18
26	2.420308	0.002951	100	l1	None	0.658491	0.018931	19
23	1.758626	0.002943	10	l2	None	0.658491	0.018931	19
14	2.941302	0.005462	1	l1	None	0.658491	0.018931	19
29	1.685547	0.002728	100	l2	None	0.658491	0.018931	19
17	1.934714	0.002950	1	l2	None	0.658491	0.018931	19
11	1.779769	0.002972	0.1	l2	None	0.658491	0.018931	19
5	1.752945	0.002971	0.01	l2	None	0.658491	0.018931	19
20	2.441495	0.002891	10	l1	None	0.658491	0.018931	19
8	2.383284	0.002963	0.1	l1	None	0.657233	0.017452	27
3	0.039744	0.003195	0.01	l2	MinMaxScaler()	0.649686	0.016352	28
2	2.282779	0.003140	0.01	l1	None	0.644025	0.025267	29
0	0.048881	0.003470	0.01	l1	MinMaxScaler()	0.517610	0.008577	30

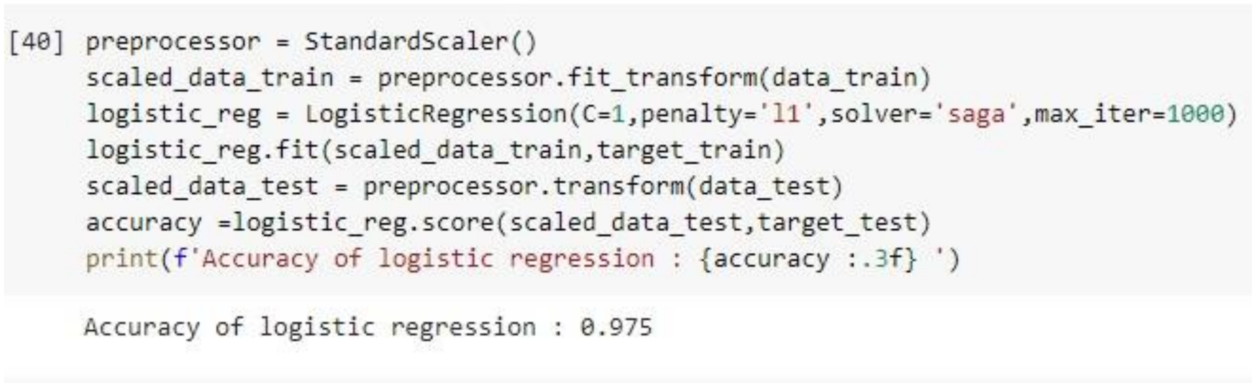
Figure 17. Result of hyperparameter tuning part-2

As it can be seen best combination of hyperparameters are: l1 penalty, C =1 and using StandardScaler. Of course, as the table shows we got close results for other combinations. This table also shows using scaler can improve model performance a lot, without using scaler maximum accuracy that we have got is 65.84%, but by using scaler we got accuracy up to 96%!!

5.6. Accuracy on test set

We use the hyperparameters that we got from last section, and train a model, then test it on test dataset.

As following figure shows, we got accuracy of 97.5% on test set which is not too bad!!



```
[40] preprocessor = StandardScaler()
     scaled_data_train = preprocessor.fit_transform(data_train)
     logistic_reg = LogisticRegression(C=1,penalty='l1',solver='saga',max_iter=1000)
     logistic_reg.fit(scaled_data_train,target_train)
     scaled_data_test = preprocessor.transform(data_test)
     accuracy =logistic_reg.score(scaled_data_test,target_test)
     print(f'Accuracy of logistic regression : {accuracy :.3f} ')
```

```
Accuracy of logistic regression : 0.975
```

Figure 18. Model performance on test set

5.7. Interpretation of results by confusion matrix

Confusion matrix is a summary of prediction results on a classification problem. The number of correct and incorrect predictions are summarized with count values and broken down by each class.

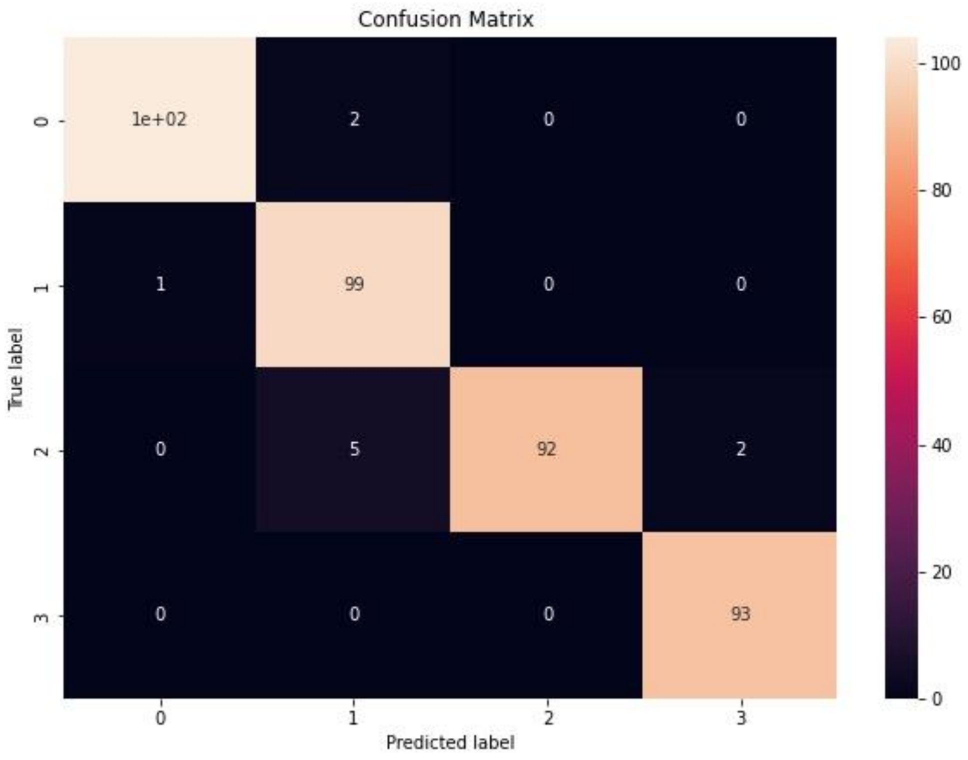


Figure 19. Confusion matrix

Numbers on diagonal of matrix shows number of correct predictions for each class, as you can see performance of our model on each class is almost the same due to balanced dataset we have, we got almost the same result on each class.

We can also see on figure 20, normalized confusion matrix, if we divide each row element by sum of the entire row. the final normalized matrix will show us the percentage out of all true labels for a particular class, what was the % prediction of each class made by our model for that specific true label.

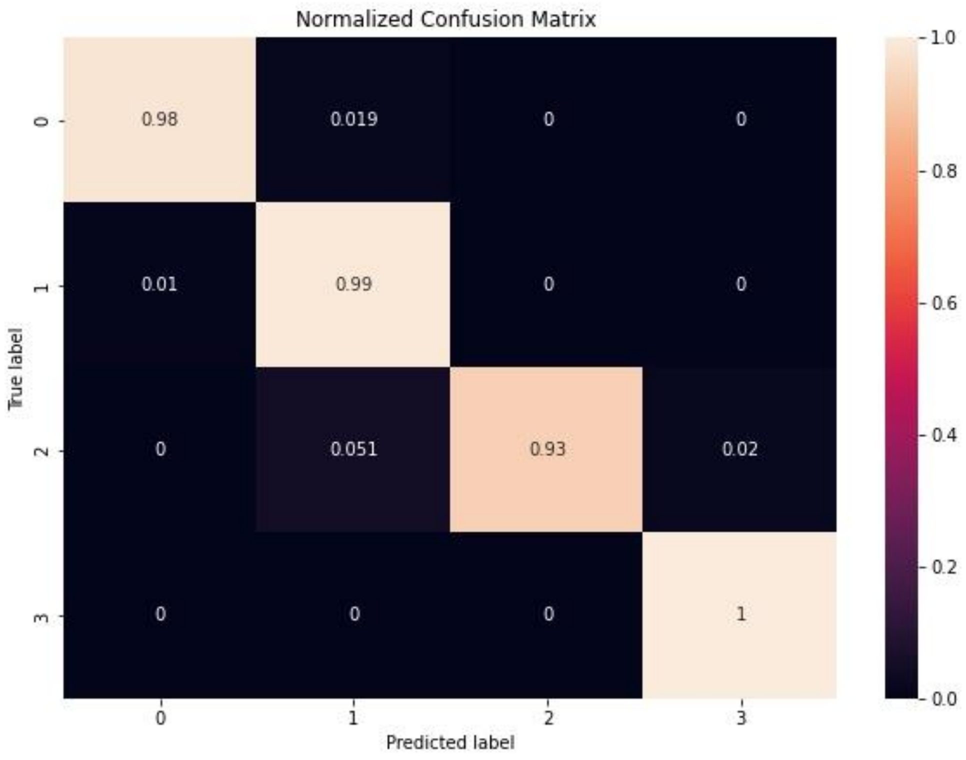


Figure 20. Normalized confusion matrix

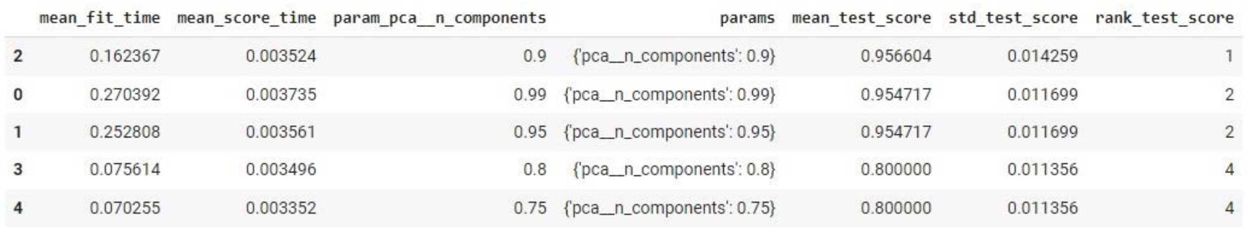
6. PCA

6.1. What is PCA?

Principal Component Analysis, or PCA, is a dimensionality-reduction method that is often used to reduce the dimensionality of large data sets, by transforming a large set of variables into a smaller one that still contains most of the information in the large set.

6.2. Applying PCA on dataset

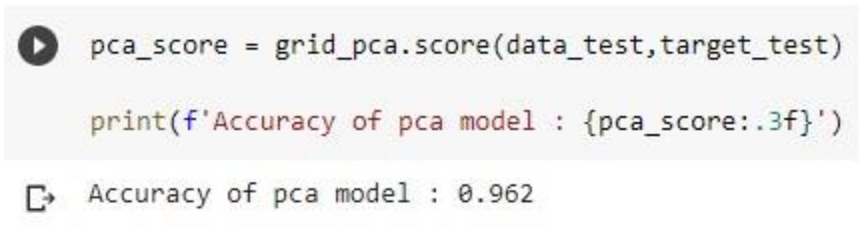
We apply PCA on the dataset with different portions of variance, 0.99, 0.95, 0.9, 0.8 and 0.75. Then train model on the dataset and report the results. The following figure shows the results:



	mean_fit_time	mean_score_time	param_pca__n_components	params	mean_test_score	std_test_score	rank_test_score
2	0.162367	0.003524	0.9	{'pca__n_components': 0.9}	0.956604	0.014259	1
0	0.270392	0.003735	0.99	{'pca__n_components': 0.99}	0.954717	0.011699	2
1	0.252808	0.003561	0.95	{'pca__n_components': 0.95}	0.954717	0.011699	2
3	0.075614	0.003496	0.8	{'pca__n_components': 0.8}	0.800000	0.011356	4
4	0.070255	0.003352	0.75	{'pca__n_components': 0.75}	0.800000	0.011356	4

Figure 21. Result of model which trained on PCA reduced dataset

As you can see we got nearly close results to previous model, when we don’t use PCA, but by reducing portion of variance, accuracy model



```
pca_score = grid_pca.score(data_test,target_test)

print(f'Accuracy of pca model : {pca_score:.3f}')
```

```
Accuracy of pca model : 0.962
```

dropping down. Another notable result that we can get from this table is time of model fitting is lower than when don’t use PCA, it is approximately 10X faster with nearly same accuracy. When we have large dataset with lots of features, we can apply PCA to reduce dimension of dataset to train model faster. Also, we test our model with best mean test score of above table on test set.

Figure 22. Accuracy of PCA model on test set

Performance on test set is also close to previous model, without PCA, 96.2%. Previous model had accuracy of 97.5% on test set.

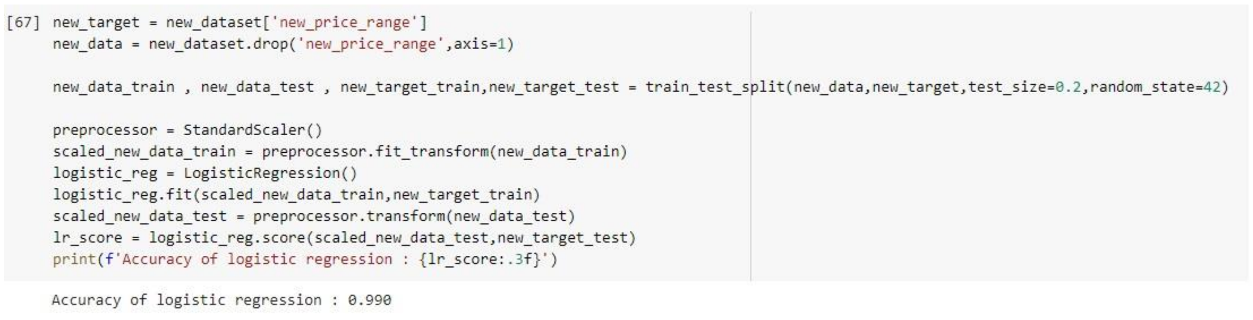
7. Creating new dataset by labelling

7.1. Creating new dataset

We change the label of records in classes 1,2 and 3 to 4 and keep label of class 0. By doing so, we make an imbalanced dataset. Then we train our model on the new dataset.

7.2. Results of the model on new dataset

We trained our model on the new dataset and the below figure shows the accuracy of model in test set of the imbalanced dataset.



```
[67] new_target = new_dataset['new_price_range']
     new_data = new_dataset.drop('new_price_range',axis=1)

     new_data_train , new_data_test , new_target_train,new_target_test = train_test_split(new_data,new_target,test_size=0.2,random_state=42)

     preprocessor = StandardScaler()
     scaled_new_data_train = preprocessor.fit_transform(new_data_train)
     logistic_reg = LogisticRegression()
     logistic_reg.fit(scaled_new_data_train,new_target_train)
     scaled_new_data_test = preprocessor.transform(new_data_test)
     lr_score = logistic_reg.score(scaled_new_data_test,new_target_test)
     print(f'Accuracy of logistic regression : {lr_score:.3f}')

     Accuracy of logistic regression : 0.990
```

Figure 23. Accuracy of model on test set of imbalanced dataset

We got 99% accuracy!! But as mentioned earlier it isn’t appropriate metric when dataset is imbalanced. So, we use appropriate metrics and upsampling method which discussed earlier.

We also look at normalized confusion matrix and classification report of our model which consists of recall, precision and f1-score.

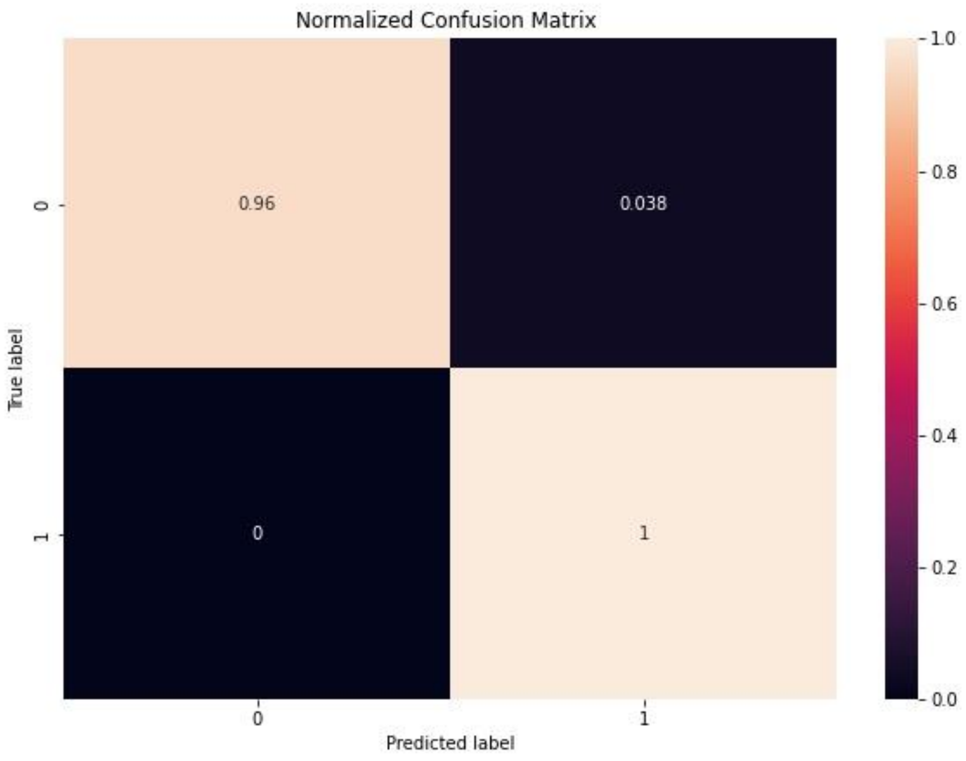
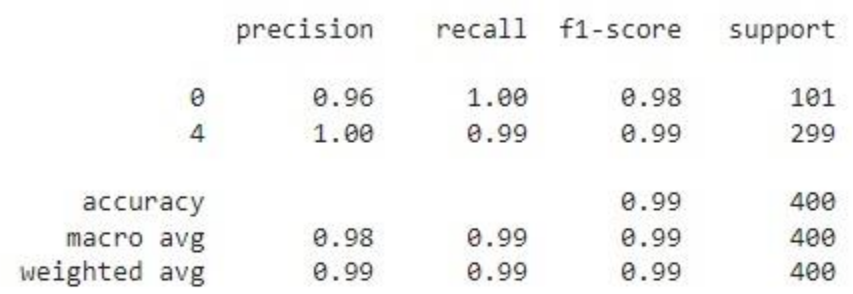


Figure 24. Normalized confusion matrix



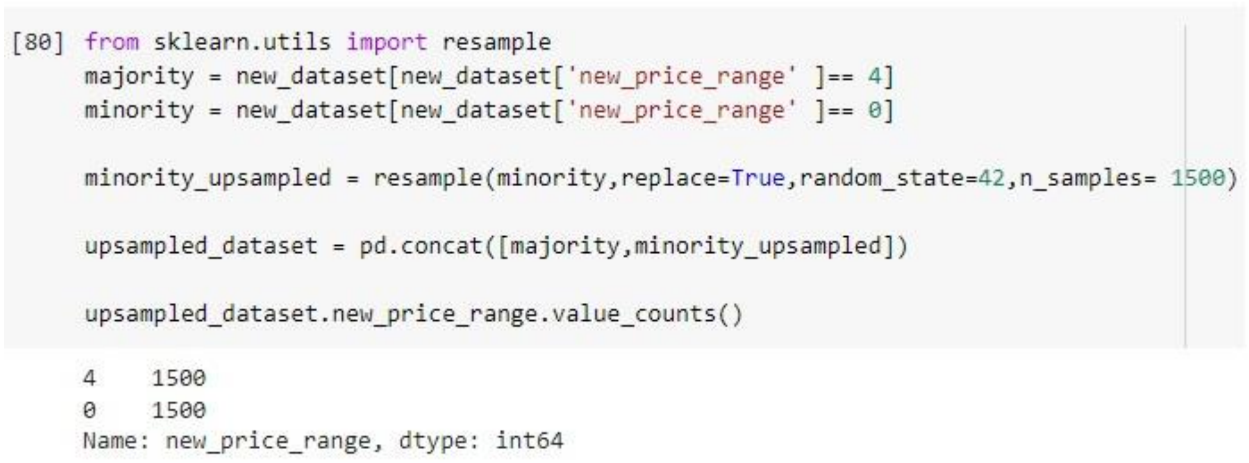
```
              precision    recall  f1-score   support

           0       0.96      1.00      0.98       101
           4       1.00      0.99      0.99       299

    accuracy                           0.99       400
   macro avg       0.98      0.99      0.99       400
weighted avg       0.99      0.99      0.99       400
```

Figure 25. Classification report

Figure 24 shows that accuracy of our model in both classes are approximately same, but 0 class has a lower accuracy.
According to figure 25, Classification report, our model has also good f1-score in both classes, this report shows despite of imbalanced dataset our model trains well.

7.3. Upsampling minority class

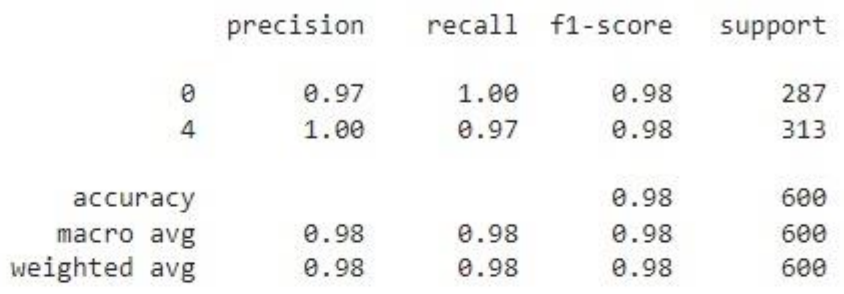
We upsample the minority class of our dataset to have the same size with majority class. We upsample 0 class to increase size of it to 1500. Then, train the model and see results.



```
[80] from sklearn.utils import resample
     majority = new_dataset[new_dataset['new_price_range' ]== 4]
     minority = new_dataset[new_dataset['new_price_range' ]== 0]

     minority_upsampled = resample(minority,replace=True,random_state=42,n_samples= 1500)

     upsampled_dataset = pd.concat([majority,minority_upsampled])

     upsampled_dataset.new_price_range.value_counts()

4    1500
0    1500
Name: new_price_range, dtype: int64
```

Figure 26. Upsampling minority class

Also, we look at classification report of model. As below figure shows this model has good f1-score in both classes, too.



```
              precision    recall  f1-score   support

           0       0.97      1.00      0.98       287
           4       1.00      0.97      0.98       313

    accuracy                           0.98       600
   macro avg       0.98      0.98      0.98       600
weighted avg       0.98      0.98      0.98       600
```

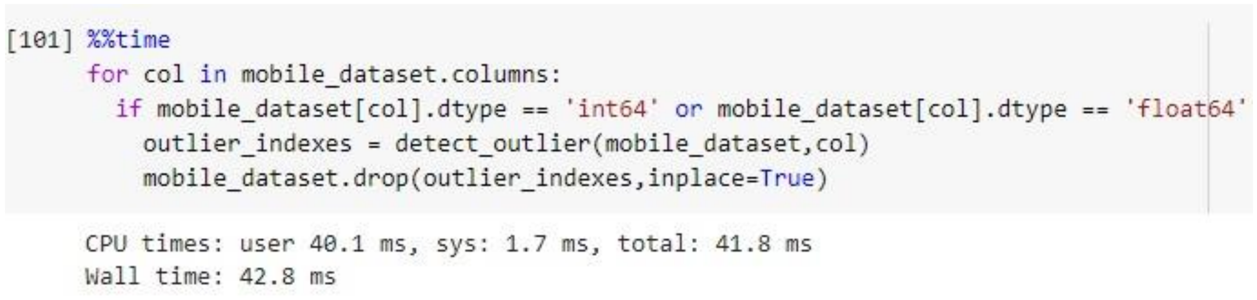
Figure 27. Classification report for upsample dataset

8. EXTRA

We use dask for data preprocessing and model training and compare its runtime.

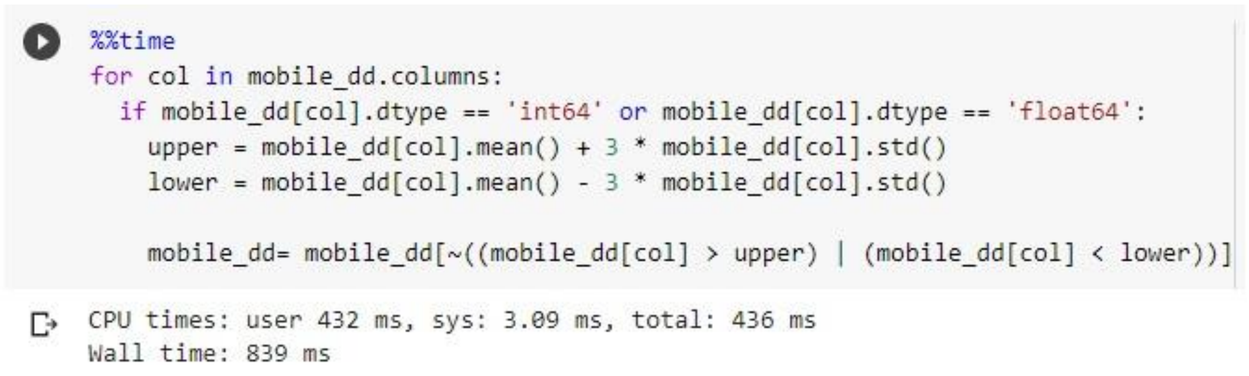
8.1. Data preprocessing using dask

We have simple preprocessing step here, just deleting outliers. In the following figures you can see runtimes of them:



```
[101] %%time
      for col in mobile_dataset.columns:
        if mobile_dataset[col].dtype == 'int64' or mobile_dataset[col].dtype == 'float64'
          outlier_indexes = detect_outlier(mobile_dataset,col)
          mobile_dataset.drop(outlier_indexes,inplace=True)

      CPU times: user 40.1 ms, sys: 1.7 ms, total: 41.8 ms
      Wall time: 42.8 ms
```

Figure 28. Normal preprocessing



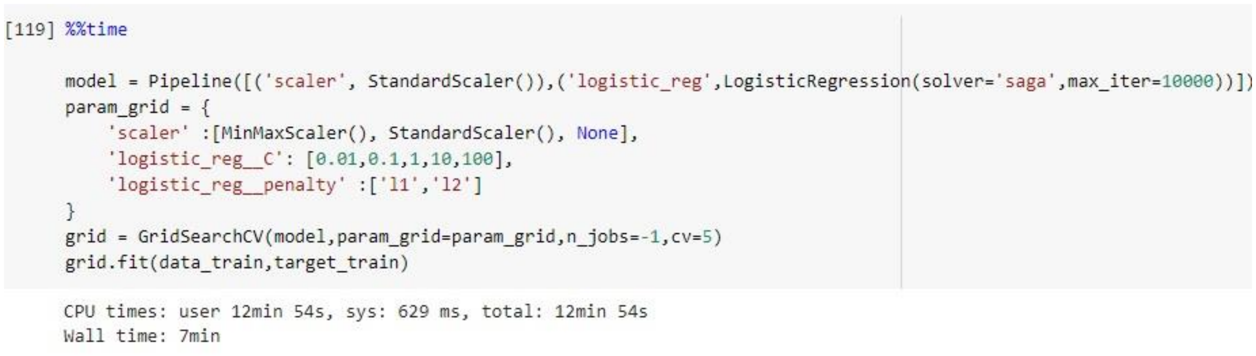
```
%%time
for col in mobile_dd.columns:
  if mobile_dd[col].dtype == 'int64' or mobile_dd[col].dtype == 'float64':
    upper = mobile_dd[col].mean() + 3 * mobile_dd[col].std()
    lower = mobile_dd[col].mean() - 3 * mobile_dd[col].std()

    mobile_dd= mobile_dd[~((mobile_dd[col] > upper) | (mobile_dd[col] < lower))]

CPU times: user 432 ms, sys: 3.09 ms, total: 436 ms
Wall time: 839 ms
```

Figure 29. Preprocessing using dask

Surprisingly, we got longer wall time when we used dask, it seems due to size of dataset isn't too large and isn't appropriate using multiprocessing for that.

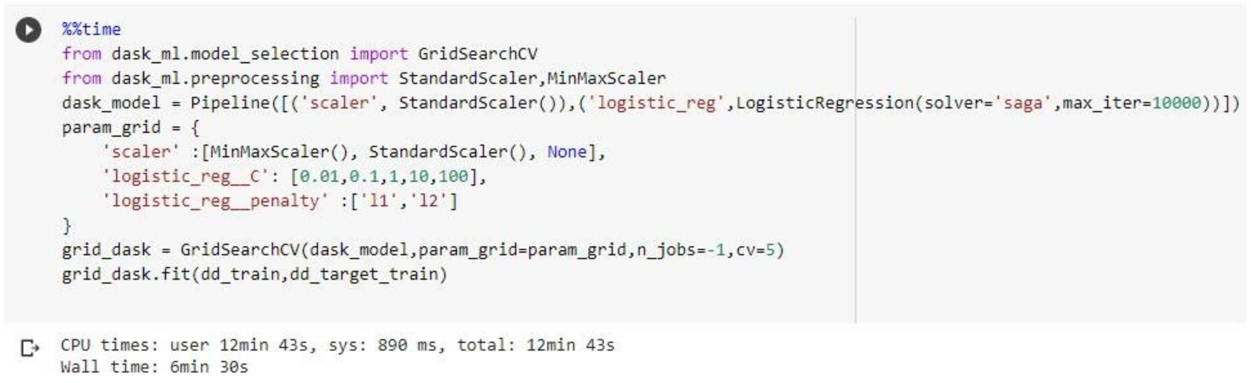
8.2. Model training using dask

We compare runtime of GridSearchCV in dask with sklearn. In following figures you can see runtimes of them:



```
[119] %%time

      model = Pipeline([('scaler', StandardScaler()),('logistic_reg',LogisticRegression(solver='saga',max_iter=10000))])
      param_grid = {
          'scaler' :[MinMaxScaler(), StandardScaler(), None],
          'logistic_reg__C': [0.01,0.1,1,10,100],
          'logistic_reg__penalty' :['l1','l2']
      }
      grid = GridSearchCV(model,param_grid=param_grid,n_jobs=-1,cv=5)
      grid.fit(data_train,target_train)

      CPU times: user 12min 54s, sys: 629 ms, total: 12min 54s
      Wall time: 7min
```

Figure 30. Hyperparameter tuning by sklearn



```
%%time
from dask_ml.model_selection import GridSearchCV
from dask_ml.preprocessing import StandardScaler,MinMaxScaler
dask_model = Pipeline([('scaler', StandardScaler()),('logistic_reg',LogisticRegression(solver='saga',max_iter=10000))])
param_grid = {
    'scaler' :[MinMaxScaler(), StandardScaler(), None],
    'logistic_reg__C': [0.01,0.1,1,10,100],
    'logistic_reg__penalty' :['l1','l2']
}
grid_dask = GridSearchCV(dask_model,param_grid=param_grid,n_jobs=-1,cv=5)
grid_dask.fit(dd_train,dd_target_train)

CPU times: user 12min 43s, sys: 890 ms, total: 12min 43s
Wall time: 6min 30s
```

Figure 31. Hyperparameter tuning by dask

As you can see, when we used dask, we got 30s less wall time for tuning, It's not a huge difference, but not bad!!

9. References

1. 17 Statistical Hypothesis Tests in Python [https://machinelearningmastery.com/statistical-hypothesis-tests-in-python-cheat-sheet]
2. 5 Techniques to Handle Imbalanced Data For a Classification Problem [https://www.analyticsvidhya.com/blog/2021/06/5-techniques-to-handle-imbalanced-data-for-a-classification-problem/]
3. One-vs-Rest and One-vs-One for Multi-Class Classification [https://machinelearningmastery.com/one-vs-rest-and-one-vs-one-for-multi-class-classification]
4. What is a Confusion Matrix in Machine Learning [https://machinelearningmastery.com/confusion-matrix-machine-learning]