# Coursera - IBM - Classification
# Mobile Phone Price Classification

December 29, 2025

## 1 Overview

This project uses the following dataset with mobile phone parameters: Mobile Price Classification - Kaggle

The goal is to predict the price class of the phone. The original price class is in 4 categories, but to keep the analysis focused, I will reinterpret the dataset to only include 2 classes: ["Low", "High"].

## 2 EDA

The data has the following properties:

- 2000 entries
- 20 feature columns (2 float, 18 int)
- 1 target column (price_range: int)
- All categories are balanced initially

The original data has 4 categories, I will reinterpret the data to have 2 categories. Here I will take the low-cost category as a single category, and mix the 3 other, higher cost categories to create an artificially unbalanced dataset.

| | mean | std | min | 50% | max |
|---|---|---|---|---|---|
| battery_power | 1238.51850 | 439.418206 | 501.0 | 1226.0 | 1998.0 |
| blue | 0.49500 | 0.500100 | 0.0 | 0.0 | 1.0 |
| clock_speed | 1.52225 | 0.816004 | 0.5 | 1.5 | 3.0 |
| dual_sim | 0.50950 | 0.500035 | 0.0 | 1.0 | 1.0 |
| fc | 4.30950 | 4.341444 | 0.0 | 3.0 | 19.0 |
| four_g | 0.52150 | 0.499662 | 0.0 | 1.0 | 1.0 |
| int_memory | 32.04650 | 18.145715 | 2.0 | 32.0 | 64.0 |
| m_dep | 0.50175 | 0.288416 | 0.1 | 0.5 | 1.0 |
| mobile_wt | 140.24900 | 35.399655 | 80.0 | 141.0 | 200.0 |
| n_cores | 4.52050 | 2.287837 | 1.0 | 4.0 | 8.0 |
| pc | 9.91650 | 6.064315 | 0.0 | 10.0 | 20.0 |
| px_height | 645.10800 | 443.780811 | 0.0 | 564.0 | 1960.0 |
| px_width | 1251.51550 | 432.199447 | 500.0 | 1247.0 | 1998.0 |
| ram | 2124.21300 | 1084.732044 | 256.0 | 2146.5 | 3998.0 |
| sc_h | 12.30650 | 4.213245 | 5.0 | 12.0 | 19.0 |
| sc_w | 5.76700 | 4.356398 | 0.0 | 5.0 | 18.0 |
| talk_time | 11.01100 | 5.463955 | 2.0 | 11.0 | 20.0 |
| three_g | 0.76150 | 0.426273 | 0.0 | 1.0 | 1.0 |
| touch_screen | 0.50300 | 0.500116 | 0.0 | 1.0 | 1.0 |
| wifi | 0.50700 | 0.500076 | 0.0 | 1.0 | 1.0 |
| price_range | 1.50000 | 1.118314 | 0.0 | 1.5 | 3.0 |

Figure 1: There is quite a wide range of values. For example, pixel sizes range in the thousands, while categorical variables (sometimes binary) hover around 0-1. Scaling will likely be necessary.

The classes are initially balanced (500 samples in each category), in the preprocessing step the classes will be reinterpreted.

# 3 Preprocessing

In this step the data was split into test-train sets. Usually the scaling is also done here, but that will be taken care of as part of the pipeline in GridSearch.

The data is also already split up to be resampled using SMOTE and undersampling of the majority class.

# 4 Model Training

## 4.1 Naive Linear Model

The first step was to fit a basic, unparametrized Logistic Regressio model. This model already provided good out-of-the-box performance:

```
{'type': 'Original',
 'accuracy': 0.942,
 'recall': 0.976,
 'auc': 0.908,
 'precision': 0.9481865284974094,
 'fscore': 0.9749001126933716}
```

Figure 2: Basic Linear Regression already provides high accuracy, precision, recall and F1.

From this basic model we can get some insight into which features might be important

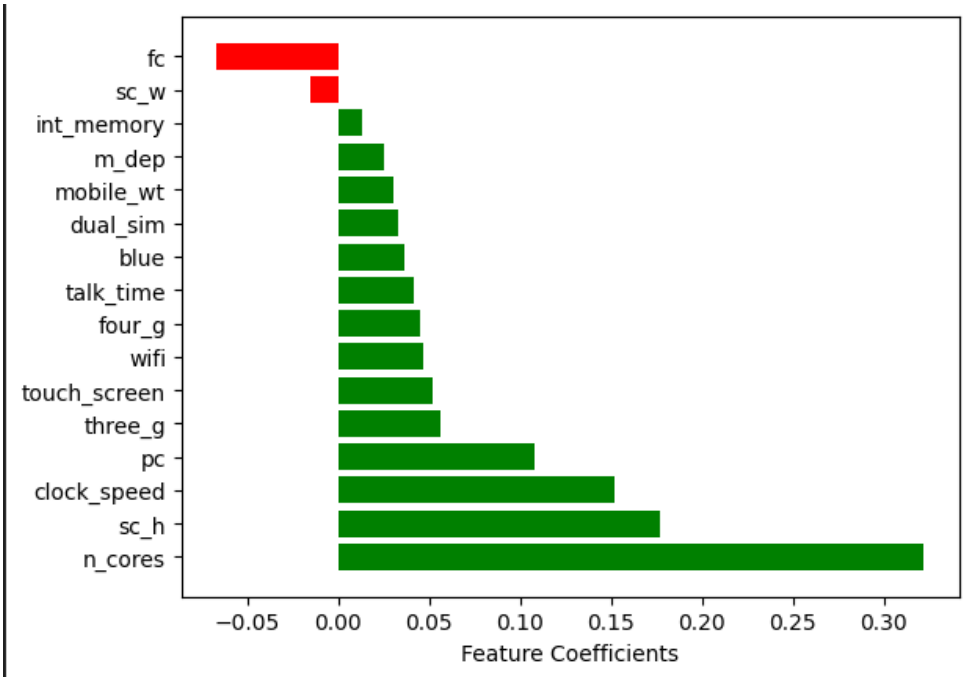


Figure 3: According to the linear model, the most important feature is the amount of cores, the screen height and the clock speed.

## 4.2 Gridsearch Training

Gridsearch was used to find the best model, and select the best parameters, the image below shows which parameters were used for this process.

```
pipe = Pipeline([('scaler', MinMaxScaler()), ('model', LogisticRegression())])

params = [
    {
        "scaler": [MinMaxScaler(), StandardScaler(), "passthrough"],
        "model": [LogisticRegression(max_iter=1000)],
        "model__C": [0.1, 1, 10],
        "model__penalty": ["l1", "l2"],
    },
    {
        "scaler": [MinMaxScaler(), StandardScaler(), "passthrough"],
        "model": [KNeighborsClassifier()],
        "model__n_neighbors": [3, 5, 7],
    },
    {
        "scaler": [MinMaxScaler(), StandardScaler(), "passthrough"],
        "model": [RandomForestClassifier()],
        "model__n_estimators": [2*n+1 for n in range(20)],
        "model__criterion": ["gini", "entropy"],
        "model__max_depth": [2*n+1 for n in range(10) ],
        "model__bootstrap": [True, False],
        "model__max_features": ["auto", "sqrt", "log2"],
    },
    {
        "scaler": [MinMaxScaler(), StandardScaler(), "passthrough"],
        "model": [SVC()],
        "model__kernel": ['linear', 'rbf', 'poly', 'sigmoid'],
        "model__C": [0.1, 1, 10, 100],
    }
]
```

Figure 4: Parameters used for gridsearch.

Eventually, the best model was an SVC with a polynomial kernel, no feature scaling and a C value of 100. The score of this model was 99% on both train and test datasets. With other performance metrics seen in the figure below:

```
'accuracy': 0.992,
'recall': 0.992,
'auc': 0.992,
'precision': 0.9973190348525469,
'fscore': 0.9922035289290111}
```

Figure 5: A score of .99 on all metrics.

Finally, comparing the confusion matrices of the naive linear model with teh optimized SVC, we can see that the misclassification is much better for both False Negatives and False Positives. In the SVC case, there are only 4 samples misclassified.

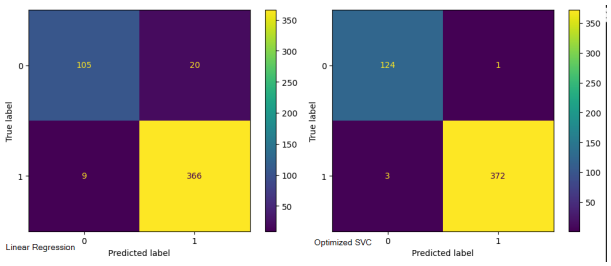


Figure 6: Left: LinearRegression, Right: SVC

# 5 Rebuilding the best model and testing rebalancing methods

Now we can reuse the best model and see if rebalancing the classes has some more influence on performance. By using SMOTE and Undersampling, together with adding weights to the class loss function we can rebalance the ratio fo the classes. The result of this analysis is shown in the figure below, where we can see that there is only a minimal effect. Since we usually care more about precision, we can conclude that there is no significant improvement here. When looking at recall, the class weight method even decreases performance.

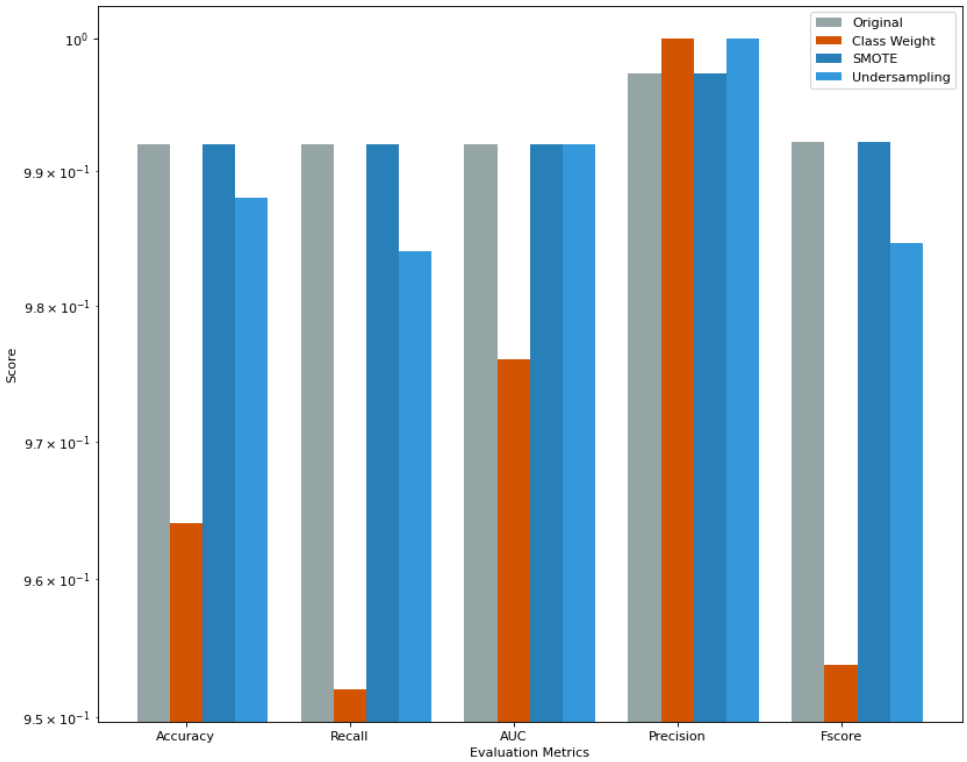


Figure 7: There is no great effect when using rebalancing (mind the logarithmic y-axis)

# 6 Interpretation

For interpretation we can look to the eli5 library. Specifically the ‘explain_weights‘ function is very useful in determining which features are important. Unfortunately, the function only works with SVC models with a ‘linear‘ kernel/ Luckily the performance is very similar to the polynomial kernel and the interpretability is assumed to transfer.

Here, with scaling the data to ensure that the coefficients are similarly weighted, we can see that the actual most important parameters in the model are:

- Amount of RAM
- Battery Power
- Pixel width and height

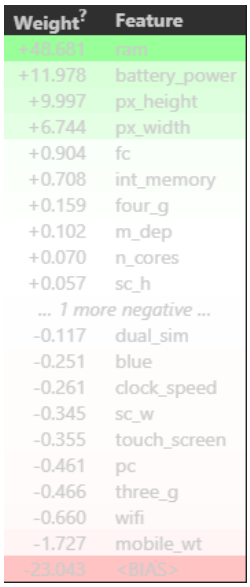


Figure 8: The most important features are: RAM, battery and pixels

# 7 Conclusion

It makes sense that the price is greatly dependent on RAM memory and pixel size. Generally flagship phones have larger screens and more RAM available. Also, similar models with RAM upgrades are generally more expensive. Most other parameters seem to have very minor effects on the outcome, with the largest negative contribution to price (weight), only contributing a factor of -1.7.

It is slightly surprising that features like dual_sim, bluetooth and 3g/4g have little effect on price. The original dataset contains a similar ratio of either, and I would expect these features to have impact on price from a consumer perspective. Another surprise is the low effect of camera number of megapixels, both for the front-facing camera (fc) as for the primary camera (pc). The primary camera even having a negative effect on price, something the simple regression model from earlier also predicted.

## 7.1 Next Steps

At this point the classifier performs well enough that there are no clear next steps. A little more pre-processing might help, since some of the input features (like pixel_height, screen_width, clock_speed, front_camera_megapixels) are skewed. Correcting this might improve model performance slightly. Another option is to use all 4 classes from the source data and see if feature importance changes meaningfully.