# CITS3401 – Project 2 Report

Student Name: Thomas Cleary
Student Number: 21704985

# Aim

The aim of this project is to predict whether the price of a mobile phone is high or not.

# Task 1 – Data Processing

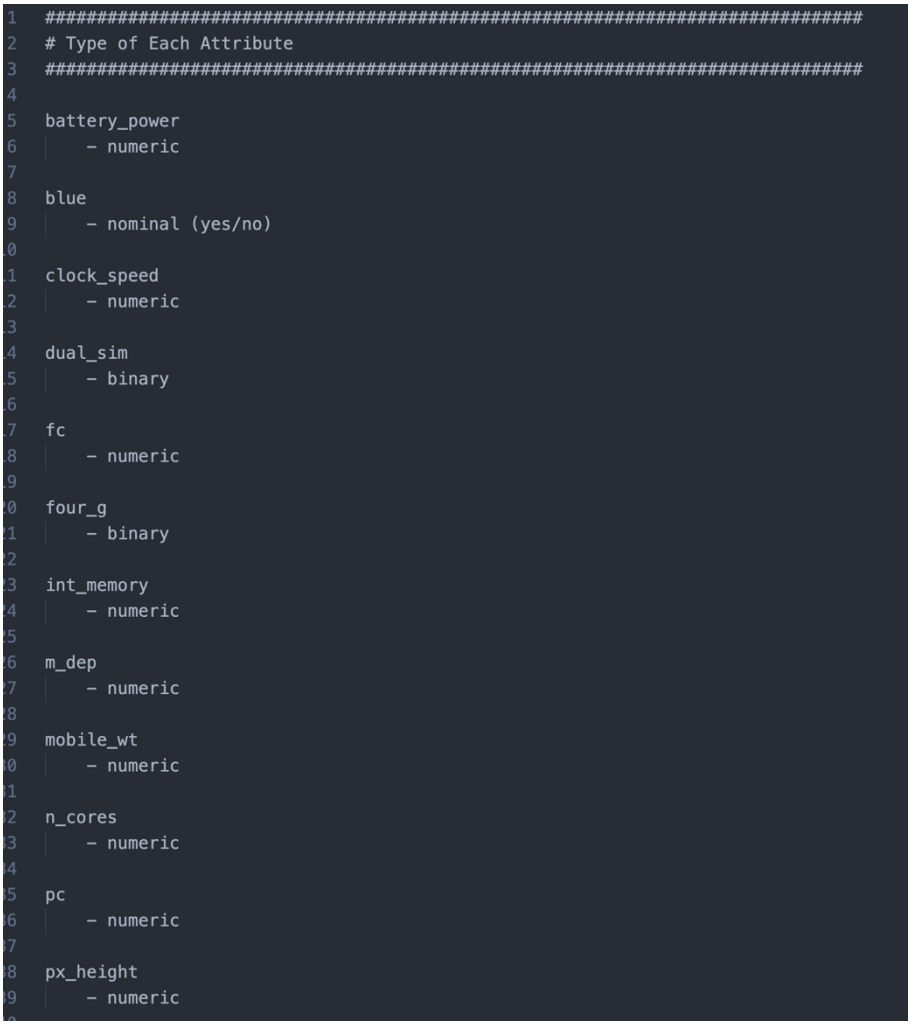
To process the provided data file *mobile_price.csv* the following steps were taken:

1. Understanding the type of data for each column in the data file
2. Checking for inconsistent data within each column/attribute
3. Converting inconsistent data within each row to conform to what we expect the value to be
4. Checking this converted data no longer contains any inconsistencies
5. Removing any unwanted columns/attributes from the data

These steps, bar the first one, were performed with Python.

## Understanding the Data

Before performing any processing, I identified the data type of each column in the provided data file as can be seen in the screenshot below. The full list can be found in the file *task1/task1b_discretise_attributes/task1b_attribute_types.txt.*



```
#############################################################################
# Type of Each Attribute
#############################################################################

battery_power
    - numeric

blue
    - nominal (yes/no)

clock_speed
    - numeric

dual_sim
    - binary

fc
    - numeric

four_g
    - binary

int_memory
    - numeric

m_dep
    - numeric

mobile_wt
    - numeric

n_cores
    - numeric

pc
    - numeric

px_height
    - numeric
```

# Fixing Inconsistencies

To check and fix inconsistencies in the 2000 line file *mobile_data.csv* I automated the process with a python script located at *task1/task1d_data_cleaning/check_column_consistency.py*.

This script loops through every data row in the provided data file and then loops through each column in this row.

If it finds a data point with a value it is not expecting, it will add the unexpected value to a list in a dictionary where the key is the name of the column this data is in (it will not be added if this specific value has already been found in another row).

A text file is then created that has a section for each column we found an inconsistency in. This section lists all the unexpected values we found in that column. The screenshot below is a portion of the text file that was created when checking the provided data file (*task1/task1d_data_cleaning/inconsistencies.txt)*.

As can be seen for the *blue* column there are several variations of “yes” and “no” found in several rows.

```
1  ##################################
2  Column Name = blue
3  has
4  No
5  NO
6  Yes
7  Has
8  Not
9  YES
10 not
11 ##################################
12 Column Name = clock_speed
13 1
14 2
15 3
16 ##################################
17 Column Name = dual_sim
18 YES
19 Yes
20 NO
21 has
22 Not
23 Has
24 not
25 No
```

The script *task1/task1d_data_cleaning/clean_mobile_data.py* changes the inconsistent values found previously into values that we expect.

For example, it converts any cased version of "yes" such as "YeS" or "yeS" into just "yes". Or any values that we expected to be float that are in fact an integer are converted to a float, such as 1 -> 1.0.

Even though I did not consider it to be an inconsistency at first, I did end up using this script to convert the values in the *price_category* column to be a value that is more descriptive, as seen below.

```
79  def clean_price_category(data):
80      if data == "1":
81          return "high"
82      else:
83          return "not_high"
84
```

This script produces a new csv file called *"cleaned_mobile_data.csv"*. This new csv file was run through the script that checks for inconsistencies and would produce a text file that had one line, "NO INCONSISTENCIES REMAINING", if there were no inconsistencies found.

The text file *"task1/task1d_data_cleaning/final_inconsistencies.txt"* is that file as seen below.

```
1  NO INCONSISTENCIES REMAINING
2
```

At this stage I was confident that the values for each column were consistent between each of the 2000 data rows.

## Removing Columns

While not necessary in this early stage I created a script, *task/task1c_attribute_selection/select_attributes.py,* that would allow me to specify a list of columns I wish to have removed from the csv file. This script would then create a new csv file with those columns removed.

I created a new csv file, *reduced_clean_mobile_data.csv* with this script that has the *id* column removed. I also used this script to change the names of the columns to something more descriptive as can be seen below.

```
def get_new_colnames():
    return [
        "id", "battery_power", "has_bluetooth",
        "clock_speed", "has_dual_sim", "front_camera_mp",
        "has_four_g", "internal_memory", "mobile_depth",
        "mobile_weight", "num_cores", "primary_camera_mp",
        "pixels_height", "pixels_width", "ram",
        "screen_height", "screen_width", "talk_time",
        "has_three_g", "has_touch_screen", "has_wifi",
        "price_category"
    ]
```

After finding attributes easier to remove with WEKA this script's only real purpose was to rename the columns.

# Task 2 – Association Rule Mining

To perform association rule mining on the cleaned data set the numeric attributes must be discretised. This was performed with WEKA's *discretize* filter. I chose to divide numeric attributes into three bins to separate them into something close to "low", "medium" and "high".

To mine more interesting rules, the attributes *has_three_g*, *talk_time*, *screen_width*, *pixel_width* were removed from the data set using WEKA.

I removed these variables as each of them were correlated with another attribute:

- has_three_g -> has_four_g
- talk_time -> battery_power
- screen_width -> screen_height
- pixel_width -> pixel_height

This data file can be found as *task2/rm_3g_talktim_scw_pxw.arff*.

## Parameters

With the truncated data I used WEKA's default Apriori-type algorithm with a minimum confidence of 0.7.

I found anything higher than 0.7 resulted in not finding any/many rules associated with *price_category=high* on the right-hand side of each rule.

The output of this can be found in the file *task2/70confidenceRules.txt*

## Top 5 Rules – Price Category = High

The top five rules when *price_category=high* is on the right hand side are:

1. battery_power='(1499-inf)' ram='(2750.666667-inf)' ==> price_category=high conf:(0.91)

   Plain English:
   - If a phone has more than 1499ma of battery power and has more than 2750MB of RAM then the price of the mobile phone will be high.

2. has_four_g=yes ram='(2750.666667-inf)' ==> price_category=high conf:(0.76)

   Plain English:
   - If a phone has 4G capability and more than 2750MB of RAM then the price of the mobile phone will be high.

3. ram='(2750.666667-inf)' has_touch_screen=yes ==> price_category=high conf:(0.74)

   Plain English:
   - If a phone has more than 2750MB of RAM and also has a touch screen the price of the phone will be high.

4. 541. has_bluetooth=yes ram='(2750.666667-inf)' ==> price_category=high conf:(0.74)

   Plain English:
   - If a phone has Bluetooth capability and more than 2750MB of RAM then the price of the phone will be high.

5. mobile_depth='(-inf-0.4]' ram='(2750.666667-inf)' ==> price_category=high conf:(0.74)

   Plain English:
   - If a phone has a depth no more than 0.4cm and has at least 2750MB of RAM then the price of the phone will be high.

## Top 5 Rules – Price Category = Not_High

The top five rules when *price_category=high* is on the right hand side are:

1. ram='(-inf-1503.333333]' ==> price_category=not_high conf:(1)

   Plain English
   - If a phone has less than 1503MB of RAM then it will not be a high priced phone

2. front_camera_mp='(-inf-6.333333]' ram='(-inf-1503.333333]' ==> price_category=not_high conf:(1)

   Plain English
   - If a phone has a front camera with less than ~6MP and less than 1503MB of RAM then the phone will not be highly priced.

3. pixels_height='(-inf-653.333333]' vram='(-inf-1503.333333]' ==> price_category=not_high conf:(1)

   Plain English
   - If the vertical resolution of the phone is less than 653 pixels and if it has less than 1503MB of RAM then it will not be a highly priced phone.

4. has_bluetooth=no ram='(-inf-1503.333333]' ==> price_category=not_high conf:(1)

   Plain English
   - If the phone does not have Bluetooth and has less than 1503MB of RAM then it will not be high priced phone.

5. ram='(-inf-1503.333333]' has_touch_screen=yes ==> price_category=not_high conf:(1)

   Plain English
   - If the phone has less than 1503MB and a touch screen it will not be a highly priced phone.

## Recommendations

My recommendation for a company wishing to design a highly priced mobile phone is to include at least 2750MB of RAM. Each of the top five rules involving highly priced phones have the condition that the phone have at least this amount of ram.

As well as this, from the rules above, the phone should have at least 1500ma of battery capacity as well as 4G and Bluetooth capabilities.

Finally the depth of the mobile phone device should not exceed 0.4cm.

# Task 3 – Training Classifiers

Using WEKA two classifiers were trained with *price_category* as the target variable.

The two machine learning algorithms used were J48 and SMO to produce a Decision Tree and Support Vector Machine respectively.

The data used was *task3/not_discretised.csv* which is the cleaned data from Task 1, however not discretised like in Task 2.

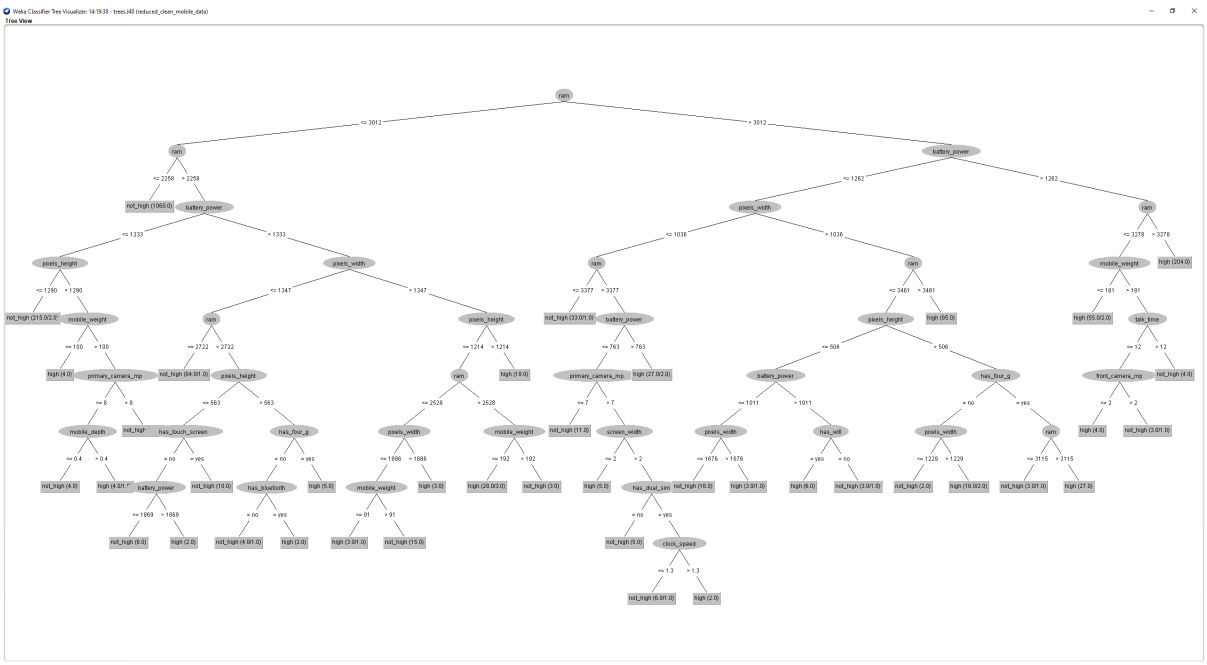
All attributes besides *id* were used when training the classifiers.

## Decision Tree

The entire output for the decision can be found in *task3/j48decisionTree/buffer.txt,* however the summary is shown below.

```
Time taken to build model: 0.08 seconds

=== Stratified cross-validation ===
=== Summary ===

Correctly Classified Instances        1878               93.9    %
Incorrectly Classified Instances       122                6.1    %
Kappa statistic                          0.8384
Mean absolute error                      0.0637
Root mean squared error                  0.2387
Relative absolute error                 16.9725 %
Root relative squared error             55.1295 %
Total Number of Instances             2000

=== Detailed Accuracy By Class ===

               TP Rate  FP Rate  Precision  Recall   F-Measure  MCC      ROC Area  PRC Area  Class
               0.956    0.112    0.962      0.956    0.959      0.838    0.944     0.969     not_high
               0.888    0.044    0.871      0.888    0.879      0.838    0.944     0.848     high
Weighted Avg.  0.939    0.095    0.939      0.939    0.939      0.838    0.944     0.939

=== Confusion Matrix ===

    a    b   <-- classified as
 1434   66 |    a = not_high
   56  444 |    b = high
```

A full resolution image of the tree structure can be found in *task3/j48decisionTree/decision_tree.png* however a scaled version is shown below.




The accuracy of this classifier is (444+1434)/2000 ~ 93.9%, meaning that 93.9% of instances have been correctly classified as either *high* or *not_high* (the error rate is then 1 – accuracy = 6.1%).

The sensitivity of the classifier is 444/500 ~ 88.8% as 444 instances out of 500 were correctly classified as *high.*

The specificity of the classifier is 1434/1500 ~ 95.6% as 1434 instances out of 1500 were correctly classified as *not_high.*

The precision of the classifier is 444/(444+66) ~ 87.1%.

The recall of the classifier is 444/(444+56) ~ 88.8%

Thus, from these metrics we can see that the classifier is slightly better at predicting if a phone's price category is *not_high* compared to *high* as specificity is greater than sensitivity.

The accuracy suggests that for every 9 in 10 phones the classifier will predict its price category correctly.

# Support Vector Machine

The entire output for the SMO output can be found in *task3/svm/buffer.txt* however the summary is shown below.

```
Time taken to build model: 0.16 seconds

=== Stratified cross-validation ===
=== Summary ===

Correctly Classified Instances        1960               98      %
Incorrectly Classified Instances        40                2      %
Kappa statistic                          0.9469
Mean absolute error                      0.02
Root mean squared error                  0.1414
Relative absolute error                  5.3314 %
Root relative squared error             32.6599 %
Total Number of Instances             2000

=== Detailed Accuracy By Class ===

               TP Rate  FP Rate  Precision  Recall   F-Measure  MCC      ROC Area  PRC Area  Class
               0.984    0.032    0.989      0.984    0.987      0.947    0.976     0.985     not_high
               0.968    0.016    0.953      0.968    0.960      0.947    0.976     0.930     high
Weighted Avg.  0.980    0.028    0.980      0.980    0.980      0.947    0.976     0.972

=== Confusion Matrix ===

    a    b   <-- classified as
 1476   24 |    a = not_high
   16  484 |    b = high
```

The accuracy of this classifier is (484+1476)/2000 = 98%, meaning *that* 98% of instances have been correctly classified as either *high* or *not_high* (the error rate is then 1 – accuracy = 2%).

The sensitivity of the classifier is 484/500 ~ 96.8% as 484 instances out of 500 were correctly classified as *high.*

The specificity of the classifier is 1476/1500 ~ 98.4% as 1476 instances out of 1500 were correctly classified as *not_high.*

The precision of the classifier is 484/(484+24) ~ 95.7%.

The recall of the classifier is 484/(484+16) ~ 96.8%

Thus, from these metrics we can see that the classifier is slightly better at predicting if a phone's price category is *not_high* compared to *high* as specificity is greater than sensitivity. This was also the case with the Decision Tree classifier.

As the metrics for this classifier are indicate more accurate predictive results compared to the Decision Tree it seems the support vector machine is a more apt classifier for prediciting whether a phone's price category will be *high* or *not_high.*

# Task 4 – Clustering

Before running a clustering algorithm, the data was transformed so that all numeric variables were converted to nominal using WEKA's *NumericToNominal* filter.

All attributes in the data set were included when running the clustering algorithm.

The SimpleKMeans Algorithm was then used to cluster the data.

## SimpleKMeans Results

The full result buffer can be found at *task4/kmeans_classes_to_clusters.txt*. The summary can be seen below.

```
Time taken to build model (full training data) : 0.01 seconds

=== Model and evaluation on training set ===

Clustered Instances

0      1202 ( 60%)
1       798 ( 40%)


Class attribute: price_category
Classes to Clusters:

   0   1  <-- assigned to cluster
 917 583 | not_high
 285 215 | high

Cluster 0 <-- not_high
Cluster 1 <-- high

Incorrectly clustered instances : 868.0  43.4    %
```

As can be seen from the summary, the algorithm has correctly classified 56.6% instances. This is only slightly better than 50% which would imply a weak clustering result. Improvements that could be made to increase the number of correctly clustered instances include:

- Reducing the number of attributes in the data set
- Converting some attributes to a large set of nominal binary values

The final cluster centroids can be seen below.

```
46
47  Final cluster centroids:
48                                          Cluster#
49  Attribute            Full Data              0          1
50                        (2000.0)     (1202.0)    (798.0)
51  ================================================================
52  battery_power              618          618          772
53  has_bluetooth               no          yes           no
54  clock_speed                0.5          0.5          0.5
55  has_dual_sim               yes           no          yes
56  front_camera_mp              0            0            1
57  has_four_g                 yes          yes          yes
58  internal_memory             27           27           14
59  mobile_depth               0.1          0.1          0.5
60  mobile_weight              182          131          105
61  num_cores                    4            8            6
62  primary_camera_mp           10            1           20
63  pixels_height              347           88          179
64  pixels_width               874          874         1429
65  ram                       1229         1464         1277
66  screen_height               17           12           17
67  screen_width                 1            1            3
68  talk_time                    7           15            4
69  has_three_g                yes          yes          yes
70  has_touch_screen           yes          yes           no
71  has_wifi                   yes           no          yes
72
73
```

From the summary above we can see that of all high-priced mobile phones, there were 70 more assigned to cluster 0 than cluster 1. This appears to coincide with the rules found with association rule mining in that cluster 0 has the following (while cluster 1 does not):

- More RAM
- Bluetooth
- Less depth

However, we would have expected cluster 0 to have a higher value for battery_power and be the only cluster with 4G. However, this is not the case in this somewhat inaccurate model.

# Task 5 – Data Reduction

To perform numerosity reduction I have used stratified random sampling to reduce the amount of data in the sample.

As well as this I have used WEKA's wavelet filter to perform attribute reduction.

## Numerosity Reduction

To perform numerosity reduction I have used python to simulate stratified sampling on the provided data set. The script *task5/sampling/stratified_sampling.py* performs this reduction.

The data was divided into two sub populations, namely *price_category=not_high* and *price_category=high.* 10% of each population was randomly sampled to be our reduced data set. The python code shown below performs this sampling.

```
8   SAMPLE_SIZE_PERCENTAGE = 0.1 # 10%
9
10
11  def reduce_data(data_rows):
12      """ reduce data rows to 10% of original size by random sampling """
13
14      population_size = len(data_rows)
15      sampled_rows = []
16
17      while (len(sampled_rows) / population_size) < SAMPLE_SIZE_PERCENTAGE:
18          random_index = random.randrange(0, population_size)
19
20          sampled_rows.append(data_rows[random_index])
21
22      return sampled_rows
23
```

The data from the reduced populations was written to the file *task5/sampling/reduced_clean_mobile_data.csv*

# Attribute Reduction

To perform attribute reduction WEKA's *wavelet* filter was applied to the numerosity reduced data. This data file is named *task5/wavelet/reduced_wavelet.arff*. It has both numerosity and attribute reduction applied to it.

# Decision Tree on Reduced Data

The J48 Decision Tree output for the reduced data is provided in the file *task5/wavelet/j48buffer.txt* and its summary is shown below.

```
=== Stratified cross-validation ===
=== Summary ===

Correctly Classified Instances         157               78.5    %
Incorrectly Classified Instances        43               21.5    %
Kappa statistic                          0.415
Mean absolute error                      0.2215
Root mean squared error                  0.4458
Relative absolute error                 58.8619 %
Root relative squared error            102.9625 %
Total Number of Instances              200

=== Detailed Accuracy By Class ===

               TP Rate  FP Rate  Precision  Recall   F-Measure  MCC      ROC Area  PRC Area  Class
               0.540    0.133    0.574      0.540    0.557      0.415    0.700     0.493     high
               0.867    0.460    0.850      0.867    0.858      0.415    0.700     0.832     not_high
Weighted Avg.  0.785    0.378    0.781      0.785    0.783      0.415    0.700     0.748

=== Confusion Matrix ===

   a   b   <-- classified as
  27  23 |   a = high
  20 130 |   b = not_high
```

The accuracy of this classifier is (130+27)/200 ~ 78.5%, meaning that 78.5% of instances have been correctly classified as either *high* or *not_high* (the error rate is then 1 – accuracy = 21.5%).

The sensitivity of the classifier is 27/50 ~ 54% as 27 instances out of 50 were correctly classified as *high.*

The specificity of the classifier is 130/150 ~ 86.6% as 130 instances out of 150 were correctly classified as *not_high.*

The precision of the classifier is 27/(27+20) ~ 57.4%.

The recall of the classifier is 27/(27+23) ~ 54%

Comparing this classifier to the same classifier that was trained on the full data set the reduced data set has hindered this classifier's ability to predict price_category correctly. The accuracy is roughly 15% lower, and the precision is around 30% lower. Suggesting that the classifier trained on reduced data is much worse at predicting if a phones price category is *high.*

However, the specificity is only 0.5% lower, suggesting that the ability to predict *price_category=not_high* is just about as good as the original classifier.

## SVM on Reduced Data

The SMO Support Vector Machine output for the reduced data is provided in the file *task5/wavelet/svmbuffer.txt* and it's summary is shown below.

```
=== Stratified cross-validation ===
=== Summary ===

Correctly Classified Instances         190               95      %
Incorrectly Classified Instances        10                5      %
Kappa statistic                          0.863
Mean absolute error                      0.05
Root mean squared error                  0.2236
Relative absolute error                 13.2847 %
Root relative squared error             51.6387 %
Total Number of Instances              200

=== Detailed Accuracy By Class ===

               TP Rate  FP Rate  Precision  Recall   F-Measure  MCC      ROC Area  PRC Area  Class
               0.860    0.020    0.935      0.860    0.896      0.864    0.920     0.839     high
               0.980    0.140    0.955      0.980    0.967      0.864    0.920     0.950     not_high
Weighted Avg.  0.950    0.110    0.950      0.950    0.949      0.864    0.920     0.923

=== Confusion Matrix ===

   a   b   <-- classified as
  43   7 |   a = high
   3 147 |   b = not_high
```

The accuracy of this classifier is (43+147)/200 = 95%, meaning that 95% of instances have been correctly classified as either *high* or *not_high* (the error rate is then 1 – accuracy = 5%).

The sensitivity of the classifier is 43/50 = 86% as 43 instances out of 50 were correctly classified as *high.*

The specificity of the classifier is 147/200 ~ 73.5% as 147 instances out of 200 were correctly classified as *not_high.*

The precision of the classifier is 43/(43+3) ~ 93.5%.

The recall of the classifier is 43/(43+7) = 86%

Comparing this classifier to the same classifier that was trained on the full data set the reduced data set has only slightly hindered this classifiers ability to predict whether a phone's price category will be high.

The accuracy has only fallen 3% and the precision has only fallen 2.2%. However, the sensitivity and specificity have fallen 10.8% and 24.9% respectively.

Suggesting that the predictive accuracy of this classifier has reduced with a reduced data set yet not as harshly as the Decision Tree. This classifier still retained high accuracy and precision despite being trained on 10% of the original data.

Once again suggesting that SVM is the superior classifier in terms of predicting whether a phones price category is high or not.

# Task 6 – Attribute Selection

Task 6 has asked to select the top 10 most important attributes from the given data set based on which are most relevant to predicting whether a phone is highly priced or not.

## Top 10 Attributes – Manual Selection

The top 10 most important attributes from my own understanding of predicting whether a phone is highly priced or not are (in order of importance):

1. ram
2. internal_memory
3. battery_power
4. pixels_height
5. pixels_width
6. clock_speed
7. num_cores
8. primary_camera_mp
9. mobile_depth
10. has_four_g

I have chosen these attributes based on my understanding of the cost-value ratio of the different components of a mobile phone.

RAM is inherently an expensive form of memory and as you increase the amount the price will increase more so than if you increased the internal memory.

The higher a screen's resolution becomes, the more expensive it is to make, and requires more battery power to keep running. It also requires more CPU power to render the screen. Hence why I have chosen attributes 2-7.

# Top 10 Attributes – Information Gain

Using WEKA's *GainRatioAttributeEval* as the attribute evaluator the top 10 most important attributes based on information gain are:

1. ram
2. battery_power
3. pixels_width
4. pixels_height
5. has_four_g
6. has_bluetooth
7. has_dual_sim
8. has_wifi
9. has_three_g
10. has_touch_screen

(the WEKA output can be found in *task6/top10_att_infogain.txt*)

# Which is Better?

The attributes selected by Information Gain are theoretically better. Information Gain determines which attributes in the data set are most useful for discriminating between classes in the data set, i.e. *price_category=high/not_high*.

Information Gain splits the data on the list of attributes above, in order, to lower the overall entropy of the data set.

Information Gain gives the optimal list of attributes based on the data provided to it, whereas my list of attributes is based on knowledge and experience outside of the data set.

Knowing that humans are not very good at intuitively seeing statistical patterns in data I would say that the list provided by Information Gain is a more accurate representation of the top 10 most important attributes.