Manuel Ortiz Cachá (k20114995)

7CCSMDM1 Data Mining

Coursework 1

Due: Thursday 18 February 2021 (23:59 UK time)

PART 1: CLASSIFICATION

We will start by downloading the respective dataset (adult data set from the *UCI Machine Learning Repository*) and printing some general information about it. As the problem formulation says, we will drop the *fnlwgt* column.

As we can see from the *.info()* method, this dataframe has almost 50 thousand rows and 14 columns; and some of these columns have missing values: *workclass, occupation* and *native-country*. So, it is a multivariate data set (it has more than one attribute) with one row per adult, describing in some manner their background. The aim is to predict whether a person makes over 50K a year or not. We have stored this information in the array *y*, it contains the data from the *class* column.

From the *.describe()* method we can see that of the total 14 columns, 5 of them are numerical type (*age* (continuous attribute from 0 to 4 containing the age range of the person)*, education-num* (continuous attribute containing a number from 1 to 16 indicating the adult's education), *capitalgain, capitalloss* (two continuous attributes from 0 to 4) and *hoursperweek* (a continuous attribute from 0 to 4 indicating the range of hours of work per week). There are 9 more categorical attributes: *workclass, education, marital-status, occupation, relationship, race, sex*, *native-country* and *class.* These categorical columns do not need description as they are quite self-explanatory. As the *education* column is the categorical representation of *education-num*, we will drop this column as it says the same as the other; and models, in general, work much better with numerical values. We will also drop the *class* column because we already have this information stored in the array *y*.

Finally, we stay with 12 columns, 5 numerical and 7 categorical.

1. **Create a table in the report with information about the adult data set.**

We already saw the number of examples from the *.shape* method. The number of missing values can be calculated with the built-in method *.isna().sum().sum()* (we first sum missing values per column, and then sum all of them). The fraction is a simple division. To calculate the number of missing values per example (per row) we can use the above-mentioned built-in function but now passing *axis=1* as a parameter and summing the rows that have more than 0 missing values. Lastly, the fraction is a simple division.

number of instances	48,842
number of missing values	6,465
fraction of missing values over all attribute values	0.009454685
number of instances with missing values	3,620
fraction of instances with missing values over all instances	0.074116539

*NOTE: these numbers reflect the data set BEFORE dropping the columns of *education* and *class*.

We have also printed this information with a module name PrettyTable that creates tables in a more readable format.

2. Convert all 13 attributes into nominal using a *Scikit-learn LabelEncoder* (**OrdinalEncoder**).

We have dropped one attribute column (*education*) because it was already encoded, so in our case we will do it over the 12 attributes we have left.

Feature encoding is done because there are several models that do not work well with categorical attributes, or directly do not work (Artificial Neural Networks NEED numerical inputs). Additionally, models usually work better with numerical attributes because there could be some kind of bias in the words representing the attributes that could affect the model's performance (their length, their first letter…). These are a couple of good reasons for encoding our data set.

The exercise asks us to use *LabelEncoder*, but as this transformer is for label targets (1D arrays), we will instead use *OrdinalEncoder*, that also works with 2D arrays.

Yet it is true, that after encoding our attributes, another problem arises. The model could believe that the numbers representing the categorical attributes hide some kind of comparison, when, for example, for the *race* or the *native-country* this is not true. In other cases, like for example in *education*, it is true. This issue can be approached by using another type of encoder called *OneHotEncoder* which assigns 0 or 1 creating N columns for the N different categories of that attribute. So, instead of having another column with numbers from 0 to N-1, it creates N columns indicating whether that instance has (1) or not (0) that value or category.

In the next picture we can see a good example of it with N=7. At the left we can see how *OrdinalEncoder* created a new column with values from 0 to N-1=6. On the other hand, we can see how *OneHotEncoder* created N=7 columns with values that are either 0 or 1.

	Bridge_Types	Bridge_Types_Cat
0	Arch	0
1	Beam	1
2	Truss	6
3	Cantilever	3
4	Tied Arch	5
5	Suspension	4
6	Cable	2

BRIDGE-TYPE (TEXT)	BRIDGE-TYPE (Arch)	BRIDGE-TYPE (Beam)	BRIDGE-TYPE (Truss)	BRIDGE-TYPE (Cantilever)	BRIDGE-TYPE (Tied Arch)	BRIDGE-TYPE (Suspension)	BRIDGE-TYPE (Cable)
Arch	1	0	0	0	0	0	0
Beam	0	1	0	0	0	0	0
Truss	0	0	1	0	0	0	0
Cantilever	0	0	0	1	0	0	0
Tied Arch	0	0	0	0	1	0	0
Suspension	0	0	0	0	0	1	0
Cable	0	0	0	0	0	0	1

This can introduce other types of problems. If an attribute has a lot of possible categories (like for example, the *occupation* or the *native-country* columns) it will create numerous columns, increasing time execution. All this is very well explained in the following article: https://towardsdatascience.com/categorical-encoding-using-label-encoding-and-one-hot-encoder-911ef77fb5bd .

As the first model we have to do is without missing values, we are going to drop them before applying *OrdinalEncoder*. After the encoding, we have to print the set of possible discrete values for each attribute, in other words: the unique values of each attribute. This can be done with a

simple for: for each column we print its unique values. As before, we are going to print out these values helping ourselves with *PrettyTable*.

3. **Ignore any instance with missing value(s) and use Scikit-learn to build a decision tree for classifying an individual to one of the <= 50K and > 50K categories. Compute the error rate of the resulting tree.**

Before starting with the model, we have to split our data set in train and test to later evaluate the model performance on the test set. After dropping the rows with missing values with the *dropna()* method, we are left with 45.222 exemplars (this has been done just before the encoding, in the previous question). We will make a train/test split of 80%/20% and train our model with the training set. This training set will have the 80% of the initial 45k instances, resulting in a training data set of more than 36k instances. On the other hand, the test set will have near to 9k exemplars, enough to evaluate our classifier.

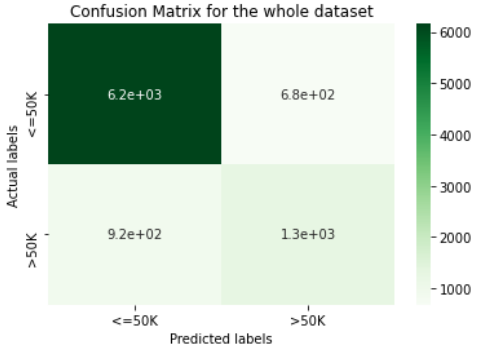
To train this Decision Tree Classifier, we will help ourselves with a built-in method of *Scikit-Learn*: *DecisionTreeClassifier*. After initialising the model, we train it with *.fit()*.

Later, to validate it, we will make use of a 10-times 10-fold cross validation, which is highly recommended because of its good results (there is theoretical evidence supporting its use). This method is used in the script with *RepeatedKFold()*, we have set the parameters of *n_splits=10* (10-fold) and *n_repeats=10* (10-times).

Now that we have our model trained with the scores saved in the *scores* variable, we can evaluate our model to see how well it did. We have obtained 100 different scores (10-times 10-fold, resulting in 100 outcomes), meaning that it is quite reliable. Our train set has an accuracy of **0.820 (0.006)**, and an error rate of **0.180 (0.006)**, which is nice. The second number in parentheses is the standard deviation, so in general the accuracies for each fold of the 100 folds is between 0.814-0.826. With the error rate the analysis is the same.

For the test set we have to predict the possible category using *.predict()* on the *X1_test* set that we reserved for this part. Those predictions will be stored in *y1_hat* and will be later used to evaluate the accuracy with the actual *y1_test* instances. Finally, we have an accuracy of **0.823**, which makes good sense observing the train accuracy. Therefore, our error rate for the test set is **0.177**.

Further on, we can use a confusion matrix to better evaluate our model. I have used some code found in Stack Overflow to print out the confusion matrix in a readable manner (https://stackoverflow.com/questions/19233771/sklearn-plot-confusion-matrix-with-labels). The results are the following:



As we can see, our model performs really good when predicting the class <=50K, and not so good when predicting the other class. One of the reasons of this difference could be the number of instances of one class and of the other. If we sum the actual instances of <=50K from the train set, we have near to 27,000 exemplars. On the other hand, we only have 9,000 instances from the actual >50K class. This could lead to a better training of the first class, as it has more chances to practice, and not such a thorough training of the >50K class. This is better reflected when calculating the precision, recall and f1-score. From our programme:

Precision score for the whole dataset= TP/(TP+FP) =

<=50K: 0.870

>50K: 0.653

Precision score for the whole dataset= TP/(TP+FP) =

<=50K: 0.900

>50K: 0.583

f1 score for the whole dataset= 2*(precision*recall)/(precision+recall) =

<=50K: 0.885

>50K: 0.616

We can observe that the model performs much better on the <=50K class.

4. Investigate two basic approaches of handling missing values.

The first thing this question of the Coursework asks us is to construct another data set. As we named the data set of the first Decision Tree X1, we will name this new data set X2 to locate it easily.

This data set (X2) should include all instances with missing values and the same number of random instances without missing values. This means that X2 will have a total of 3,620*2=7,240 data points. This can be done easily with the built-in method of *pandas .sample()*. This method will select randomly the number of instances or the fraction of instances you need.

What we will do first will be to merge *df* and *y* again so that we have the true values with the corresponding instances (*df*). Now, we will create another dataframe where we will put the instances with missing values (*X21*), this data set has a shape of 3,620 rows and 13 columns (12 attributes and the target).

Now, as we have our data points with missing values in *X21*, we can drop all of them from *df* to stay with a dataframe without missing values (*df_without_nans*). To complete the data set we need for this exercise, we need 3,620 random instances without missing values. We can select them with *.sample()*, and we will store them in *X22*.

*****NOTE: beyond this point, every time someone runs the script, there is some unavoidable randomness on the model training of D'_1 and D'_2. Each time, the *.sample()* method will pick 3,620 different random instances, changing how it is trained. The differences should be small, but they will exist. I will put the values obtained from the first time I am running this script.

Our last step is to concatenate *X21* and *X22* to end up with *X2*, named as *D'* in the question formulation. The *.sample(frac=1)* is to shuffle this dataframe, so the missing values are not on top, and the other values without missing values below. Now that we have our training set ready, we can drop the targets and store them in *y2*.

The next thing the question asks us is to create two new dataframes from the resulting *D'* (*X2*) by replacing the missing values with two different values. We are fortunate to have all the missing values in categorical columns, so we do not have to worry about the numerical attributes.

The first one will be named *D1* (D'_1) and it will have all the missing values replaced with *'missing'*. This is done to treat equally the missing values as any other type of category. This is done easily with the built-in method *.fillna()* from *pandas*, where you only have to pass as parameter the value you want to use to replace them, in our case we will pass *'missing'* as the parameter.

The second data set will be named *D2* (D'_2), and in this case it will replace the missing values for the most popular value of that column. This can be handled easily too with the built-in method *.fillna()*, but now we have to pass a dictionary with the most popular values of each column. This is done with the *mode()* of each column.

Now that we already have our data sets ready to train and test the new models, we will repeat the same process done before by following 4 simple steps:

1. Encode our new training instances with *OrdinalEncoder*.
2. Train our models with the respective data sets (*D1* and *D2*) and their targets, which we stored before in *y2*. We also computed the training error rate, which is the following for each training set:

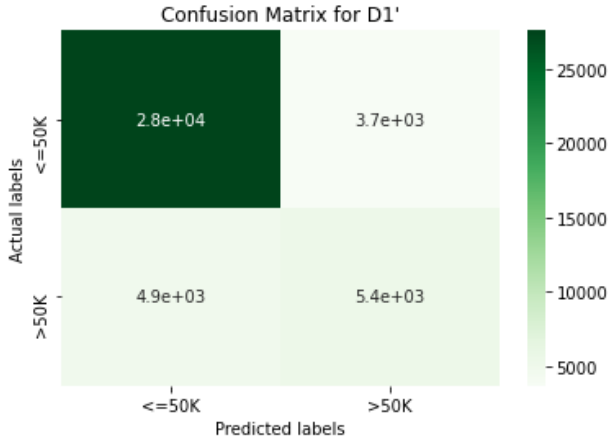
   Train Error Rate for D1': 0.158 (0.012). The standard deviation is in parentheses.

   Train Error Rate for D2': 0.159 (0.011). The standard deviation is in parentheses.

3. Modify the initial complete data set (*D*) so it works for each model. Before that, we will drop all instances that were used for the training. Thereby, the model evaluation is done with a test set (*testdf*) that does not contain any instances from the training set. We have to create two new test sets. The first one, *testD1*, with *'missing'* instead of the missing values. The second one, *testD2*, with the most popular value of that specific column, passing the dictionary we used previously for *D2*. After replacing the missing values with its respective new value, we encode it again just like in the first step, with *OrdinalEncoder*.
4. Evaluate our models. This is done by predicting new instances for the modified complete data sets (*testD1* and *testD2*). After obtaining the predicted values (*yd1_hat* for the D'_1 and *yd2_hat* for D'_2) we can proceed with the evaluation. We will call the function we created before (*confusion()*) that prints some evaluation metrics and the confusion matrices. The result is the following:

   a. D'_1

The **error rate** we obtained **for the model** (*dtree1*) **trained with the first data set** is **0. 206**. The confusion matrix is the following:



Here are other metrics from this model (printed from the program):

Test Error rate for D1': 0.206

Precision score for D1'= TP/(TP+FP) =

<=50K: 0.849

>50K: 0.596

Recall score for D1'= TP/(TP+FP) =

<=50K: 0.882

>50K: 0.527

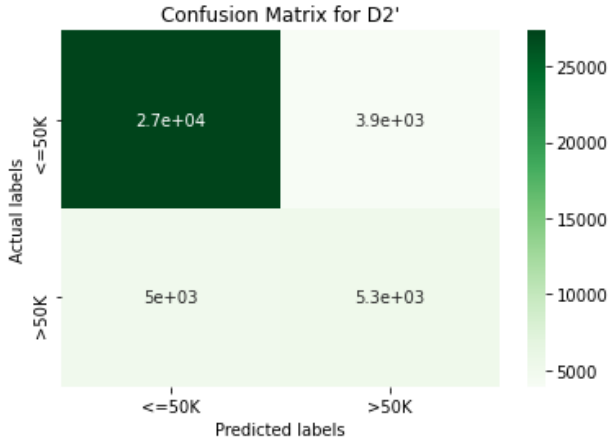
f1 score for D1'= 2*(precision*recall)/(precision+recall) =

<=50K: 0.865

>50K: 0.559

b. D'_2

The **error rate** we obtained **for the model** (*dtree2*) **trained with the second data set** is **0.215**. The confusion matrix is the following:



Here are the other metrics from this model (printed from the program):

Test Error rate for D2': 0.215

Precision score for D2'= TP/(TP+FP) =

<=50K: 0.845

>50K: 0.576

Recall score for D2'= TP/(TP+FP) =

<=50K: 0.875

>50K: 0.514

f1 score for D2'= 2*(precision*recall)/(precision+recall) =

<=50K: 0.860

>50K: 0.543

BRIEF COMMENT ON THE OBTAINED RESULTS:

As we can see there are no major differences on the performance of both models. This could be comprehensive since they did not have much data to train on, maybe if they had been more numerous, the differences would have been more significant.

The model trained with D'_1 seems to work better for class *<=50K*. There could also be some improvement in the first model (compared to the second one) for the instances of class *>50K*, which is harder to detect as we have seen with all the models (the precision and the recall is much better for the first class than for the second).

I actually do not know why this could happen. However, my guess is that when we substituted the missing values for *'missing'*, the impact on the model is lower, since with the other substitution method we made a guess that their actual value would be the most popular. When you think it closely, you cannot know if an adult is from United States instead of from Colombia just because it is more *typical*, to put an example.

In fact, if we take a closer look, this data set was surely obtained from a census made in 1994 in USA (I suppose it because it was made by a company called Silicon Graphics, which sounds a lot like California). And it does not make much sense that in the census of their country they do not know where an American citizen was born. It would be more logical that they did not know if that adult was from Europe, South America or Asia instead of from the United States.

So, if we assume that we made more errors by guessing that the missing values were actually the most typical ones, the first model has had less impact in *falsifying* the data. Therefore, it is more realistic than the second method, so it will make less errors.

All these suppositions are made without knowing how much impact each attribute has had on the final outcome, of course.

PART 2: CLUSTERING

We will start by downloading the respective dataset (Wholesale customers data data set from the *UCI Machine Learning Repository*) and printing some general information about it (*.info()* and *.describe()*), just like before.

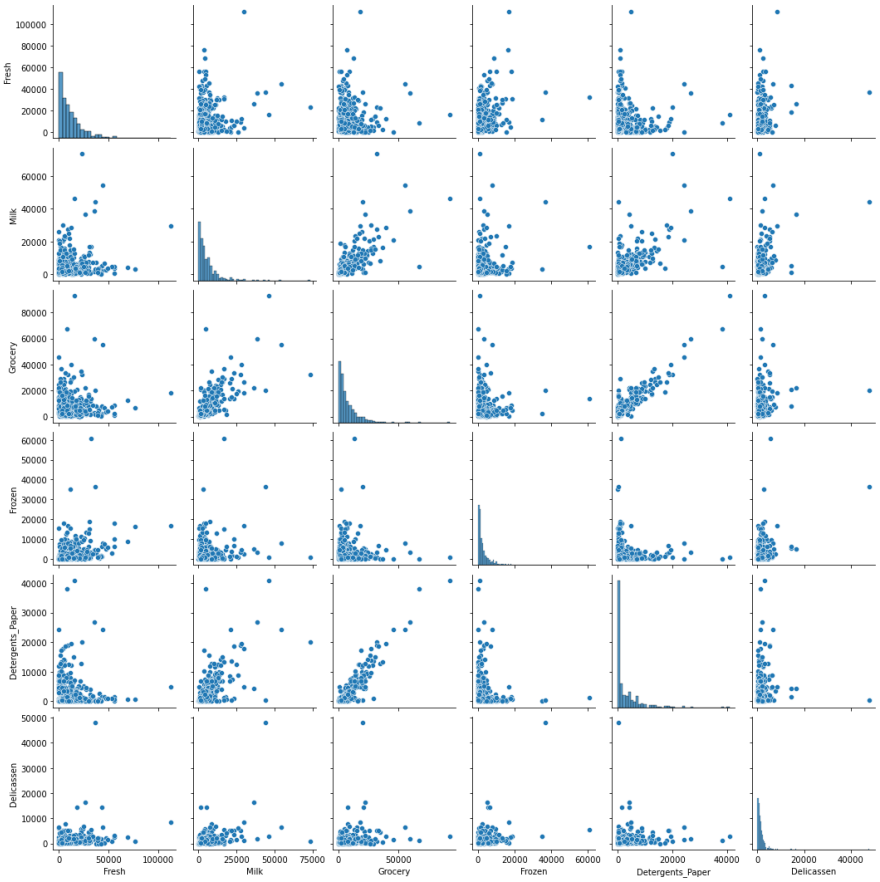
From both these methods we quickly note that it is an *easier* data set than the one from part 1. Easier in the sense that this one only has numerical values (*int64*), and there is not one missing value. Therefore, we have a couple of problems less to worry about.

This data set contains 440 different values of customers general information. By reading in the *Machine Learning Repository*, we find out that 6 of the total 8 columns are annual spending of particular products (*Fresh, Milk, Grocery, Frozen, Detergents_Paper* and *Delicassen*, which actually refers to delicatessen).

The other two columns are more descriptive, *Region* represents the customer's region, which is already encoded but it could be Lisbon, Oporto or Other (I do not know which number represents each). The last column, *Channel*, represents the customers channel, and it can take two possible values: Horeca (Hotels/Restaurants/Cafés) or Retail. I do not know which one represents 1 or 2 neither. However, the problem formulation tells us to drop these columns so we will do so with the *.drop()* method.

As I said before, all this information about the data set can be found in its source (https://archive.ics.uci.edu/ml/datasets/wholesale+customers). The aim of this exercise is to make a clustering of the customers to detect any groups of similar customers.

Before starting, I am going to plot all the pairs of attributes to have a first sense of how it looks:



So, it looks like without knowing how many clusters there are, it is difficult to make an estimation of *k*. However, we can tell that there are some attributes really correlated like *Grocery* and *Detergents_Paper* and other ones that do not seem to have any correlation at all, like *Frozen* and *Delicassen*.

1. **Create a table in the report with information about the Wholesale customers data dataframe.**

We have already seen all this information with the *.describe()* method, and by searching a little, you can also find it in the *UCI Machine Learning Repository*. Anyway, I will print it out with my program.

All these characteristics of each attribute can be easily obtained with *NumPy* built-in methods. For the mean you can use *np.mean()* and for the range you can directly use *min()* and *max()*.

This can be easily calculated with a for loop. The results are displayed on the console with *PrettyTable* and I will leave them in this following table too:

ATTRIBUTE	RANGE [min, max]	MEAN
Fresh	[3, 112151]	12000.2977273
Milk	[55, 73498]	5796.265909091
Grocery	[3, 92780]	7951.277272727
Frozen	[25, 60869]	3071.931818182
Detergents_Paper	[3, 40827]	2881.493181818
Delicassen (Delicatessen)	[3, 47943]	1524.870454545

2. **Run k-means with k=3 and construct a scatterplot for each pair of attributes using Pyplot**

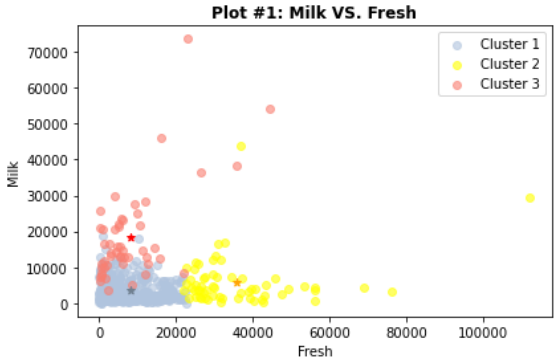
The first good news is that we do not have to process the data in any manner, we can directly use it on the training of the algorithm. First of all, we have to run k-means by passing the parameter *k*. This parameter indicates the number of clusters the algorithm has to find. In other words, it is the number of centroids the algorithm has to generate.

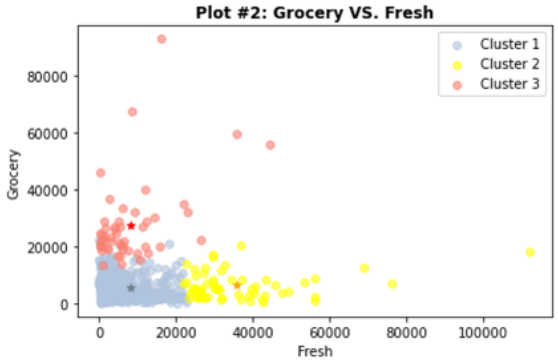
To do so, we will make use of *Scikit-Learn*'s built-in algorithm *KMeans*. The parameter *k* is passed by through *n_clusters*. Once we have trained this model (*kMeans3*) with our data, we can extract two useful values: *kMeans3.targets_* (the cluster the data points belong to) and *kMeans3.cluster_centers_* (the centroids of our clusters). We will store these values to make the pairplots later.

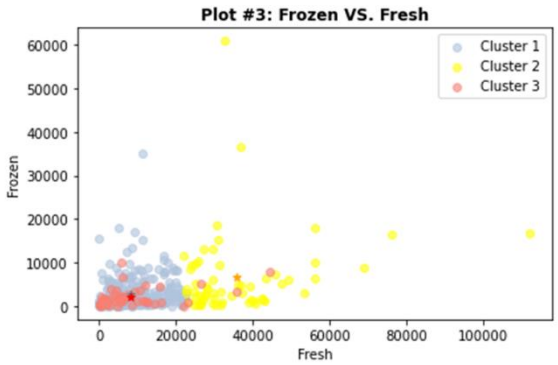
As it says in the problem formulation, with N different attributes, we should plot the triangular number of N-1 in total. This is $T_{6-1} = 5 + 4 + 3 + 2 + 1 = 15$. We are going to use a couple of for loops that loop through the columns to print a scatter plot of each pair of attributes. We are also going to choose the colour of the data points depending on the cluster they belong to.

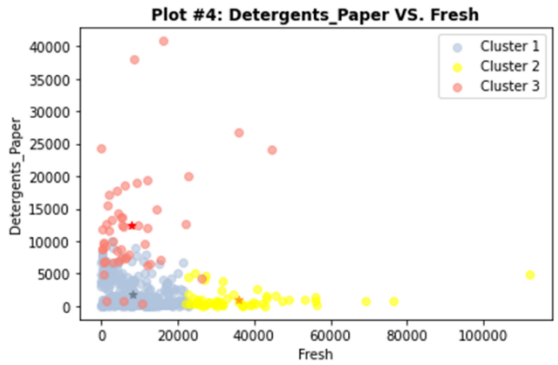
We have used three different colours for each cluster: red, yellow and grey. We have also plotted the centroids of each cluster in a similar colour so we can see where they are easily. On the next pages we are going to render the plots first, and after that we will write a few words about them.

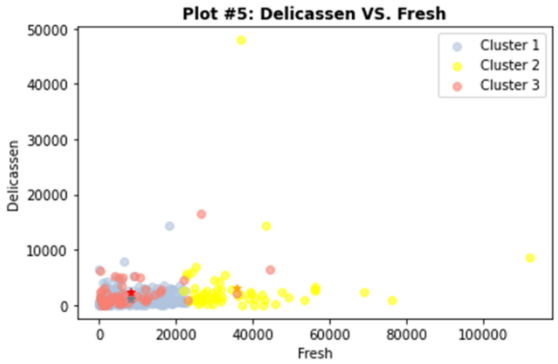
Manuel Ortiz Cachá (k20114995)

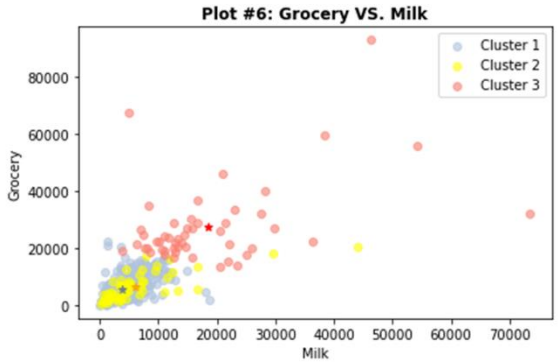


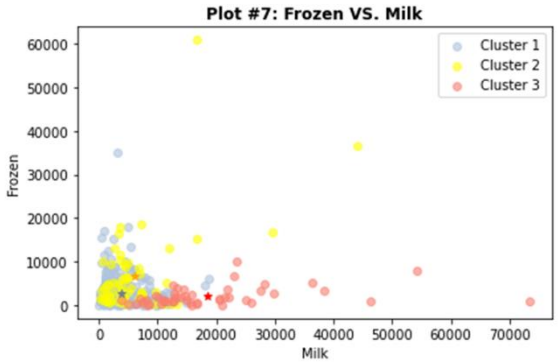


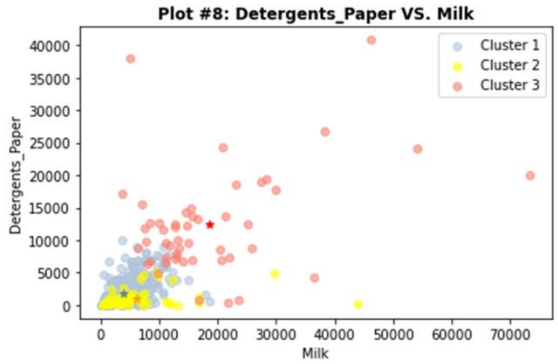


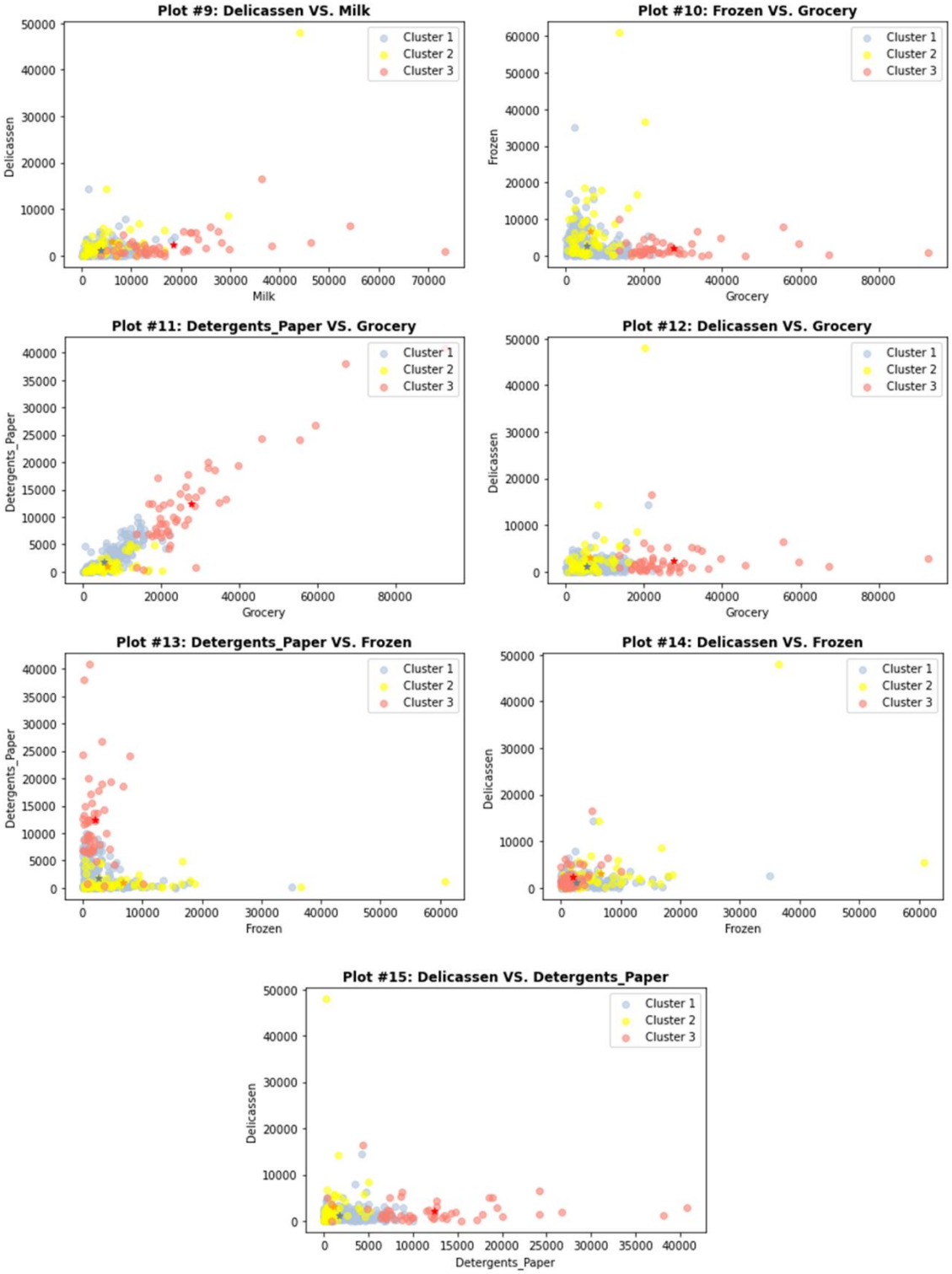












We could divide the plots depending on the amount of information they give us. There are some plots that show really clearly where the clusters are and how the grouping is done. This can be observed in the plots #1, 2 and 4. These 3 plots share a common x-axis: *Fresh*, which could mean that this attribute is a good indicator for making a good enough clustering.

There is another group of plots that have a cluster division sufficiently clear, but for two clusters. It seems like the clusters 1 (grey) and 2 (yellow) overlap for a large number of pairs of attributes. This happens for the plots #6, 7, 8, 9, 10, 11, 12, 13 and 15.

Something similar happens for the clusters 1 (grey) and 3 (red). This can be observed in the plots #3 and 5. Both centroids for these two groups of plots are quite near, having their data points quite overlapped.

There is another plot that does not give us almost any information. This is the plot #14, with the attributes *Delicassen* (Delicatessen) and *Frozen* there is no clear division. This can make sense because a priori it seems like these two attributes do not have much relation. In fact, we used these two attributes to put an example of attributes with little correlation.

We could end up the analysis with this following table that classifies the different groups of plots/clusterings we have:

AMOUNT OF INFORMATION		**PLOTS #**
A LOT OF INFORMATION: 3 distinct clusters		1, 2, 4
NOT SO MUCH INFORMATION: 2 distinct clusters	(1&2) and 3	6, 7, 8, 9, 10, 11, 12, 13, 15
	(1&3) and 2	3, 5
ALMOST ANY INFORMATION: clusters hardly distinguishable		14

3. Run k-means with k={3, 5, 10} and complete a table with the BC (Between Cluster Score), WC (Within Cluster Score) and the ratio BC/WC.

We will start with the model of k-means (*kMeans3*) with *k=3* as we already have it implemented and trained. The two metrics we need are *BC* and *WC*. First, I am going to briefly explain what each one means:

1. *Between Cluster Score (BC):* it is the measure of cluster separation, how far are the clusters between them. We want this measure to be as high as possible because that will mean that our clusters are well differentiated. The formula is the following:

$$\sum_{k=1}^{K-1} \sum_{k'=k+1}^{K} d(\boldsymbol{c}_k, \boldsymbol{c}_{k'})^2$$

2. *Within Cluster Score (WC)*: it is the measure of cluster compactness, how close are the data points of each cluster. We aim to have the smallest WC score as possible because that will indicate that our clusters are really tight, meaning that the data points are all close to their respective centroid. The formula is the following:

$$\sum_{k=1}^{K} \sum_{i \in \boldsymbol{c}_k} d(\boldsymbol{x}_i, \boldsymbol{c}_k)^2$$

Both these formulas can be implemented easily in Python, each summation is a for loop; and for the distance we will use the Euclidean distance as it is the most typical one for numerical attributes. We can make use of *NumPy* to calculate the distance with *np.linalg.norm(a-b)*, which is the same as $d(\boldsymbol{a}, \boldsymbol{b})$ when calculating the Euclidean distance.

After running the models and calculating these metrics, we print them out. We will also put them in this following table:

	k=3	***k=5***	***k=10***
BC	3,110,621,948.46884	25,621,025,526.67816	198,778,026,716.51245
WC	80,342,166,920.94078	52,928,148,942.57613	31,211,819,104.95596
BC/WC	0.038717	0.484072	6.368678

BRIEF COMMENT ON THE OBTAINED RESULTS:

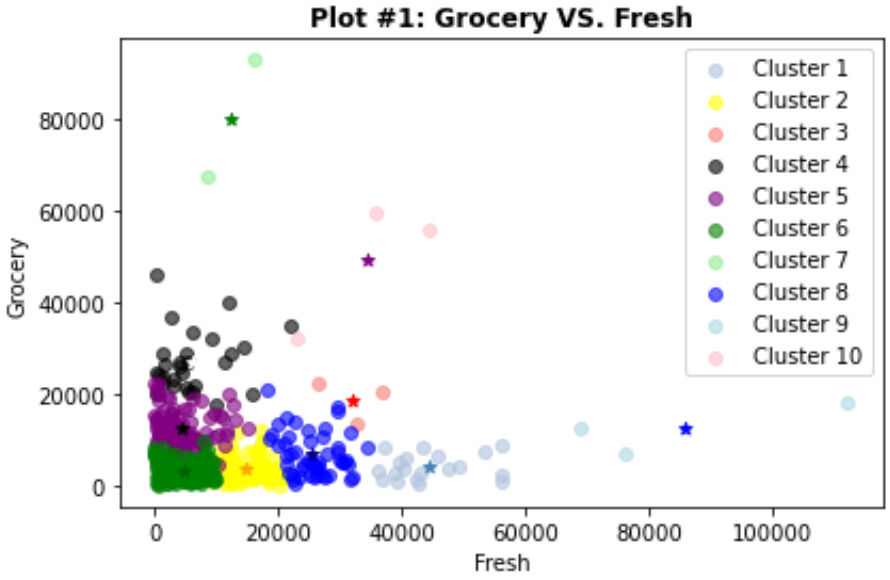
As we can see, the number of clusters (*k*) is directly proportional with the *BC* and inversely proportional with *WC*, resulting also in a direct proportion with the *Ratio BC/WC* as the numerator increases and the denominator decreases.

This makes perfect sense because as the *k* increases, there are more centroids. These centroids will almost surely group lesser points, but these points will be closer between them, so the *WC* will decrease as almost every point will be closer to a centroid. At the end, the *WC* is a sum of every data point distance to its centroid, so the number of summands is the same, but now each point will be closer to a centroid, because there are several more centroids.

In the case of the *BC*, it is also more or less intuitive why it increases. The main reason is the number of summands. If a data set has *k* clusters, the *BC* will have as much summands as the triangular number of *k-1* (explained before). So, for *k=3* there are 3 summands, for *k=5* it increases to 10 (more than three times more), and finally, for *k=10* the number ascends to 45 (more than 4 times more when the number of clusters only duplicates). Therefore, although our clusters are now closer, they are much more numerous, meaning that they are much more summands in our equation.

It is also logic to think that as the number of clusters grow, there will be more centroids that overlap. So, technically, with *k=3* if $\boldsymbol{c}_1$ and $\boldsymbol{c}_2$ overlap, we will have to sum only two times the distance from $\boldsymbol{c}_3$ to the other clusters, as this distance will be similar. However, if we set *k* to 10, we may have 5 centroids that are very near or overlap, but now we will have to make that sum 5 times for the rest of the centroids.

As it is difficult to imagine how 10 clusters look like, I am going to show one plot comparing two differentiable attributes and their different clusters:



Selecting a good value for *k* is the key step of all this process. This can be done with previous knowledge or by doing a Grid Search to find the optimal number of clusters. Beforehand, it is difficult to tell which one is better. However, if we guide ourselves with what we said before (we want a high *BC Score* and a low *WC Score*), our best model would be with *k=10*. But it is important to keep in mind that these values increase and decrease respectively with the number of centroids.