Alberto Martin Belloso albm@itu.dk
Aleksander Kähler akae@itu.dk
Rasmus Emil Odgaard raod@itu.dk

Data Science in Games – Assignment 1

Apriori

Implemented by: Alberto Martín Belloso
Data chosen: shopping.json

Description of Apriori

Soft explanation of how they work in theory

Apriori is an algorithm that can be used to find general trends or patterns in a database. For this particular example, we want to find association rules in a database of shopping lists. For example, people who buy bread also tend to buy cheese.

The first step is to find the frequent itemsets. An itemset is a set of one or more items. In order to do that, we define a minimum threshold for an itemset being frequent. This threshold is called minimum support. So we will only take in consideration itemsets that appear at least a certain percentage of the times in the database. At first, we identify the frequent individual itemsets, and then we extend to larger and larger itemsets as long as those itemsets meet with the minimum support.

Once we have all frequent itemsets, we create association rules with them. To do so, we will use those itemsets larger or equal to two items. This is because an association rule requires at least two items. With those itemsets, we create all possible association rules. For example, if our itemset is {A, B} we will create {A} -> {B} and {B} -> {A}. If the itemset is {A, B, C} we will create:

{A} -> {B, C}   {A, B} -> {C}   {A, C} -> {B}   {B} -> {A, C}   {B, C} -> {A}   {C} -> {A,B}

and so on.

Finally, for each of the association rules we will compute different metrics in order to evaluate the importance of this association rules. The implemented metrics are:

- **Support**
  As mentioned before, this is an indication of how frequently the itemset appears in the dataset.

$$supp(A) = \frac{\#obervations\ with\ A}{Total\ \#transactions}$$

- **Confidence**
  How often a rule has been found to be true. If we have the rule {A} -> {B}, the confidence of this rule is the proportion of the transactions that contains A which also contains B.

$$conf(A \Rightarrow B) = \frac{supp(A \cup B)}{supp(A)}$$

- **Lift**
  The lift of a rule is the ratio of the observed support to that expected if X and Y were independent.

$$lift(A \Rightarrow B) = \frac{supp(A \cup B)}{supp(A)supp(B)}$$

Lift=1 implies that the probability of occurrence of the antecedent and that of the consequent are independent of each other. When two events are independent of each other, no rule can be drawn involving those two events.

Lift>1, lets us know the degree to which those two occurrences are dependent on one another, and makes those rules potentially useful for predicting the consequent in future data sets.

Lift<1, lets us know the items are substitute to each other. This means that presence of one item has negative effect on presence of other item and vice versa.

- **Conviction**
  Conviction is a measure of how much better than chance an association is.

$$conv(A \Rightarrow B) = \frac{1 - supp(B)}{1 - conf(A \Rightarrow B)}$$

For example, if the conviction of {A} -> {B} is 1.3 it shows that the rule would be incorrect 30% more often if the association between A and B was purely random chance.

- **Kulczynski**
  Average of two conditional probabilities: the probability of itemset B given itemset A, and the probability of itemset A given itemset B.

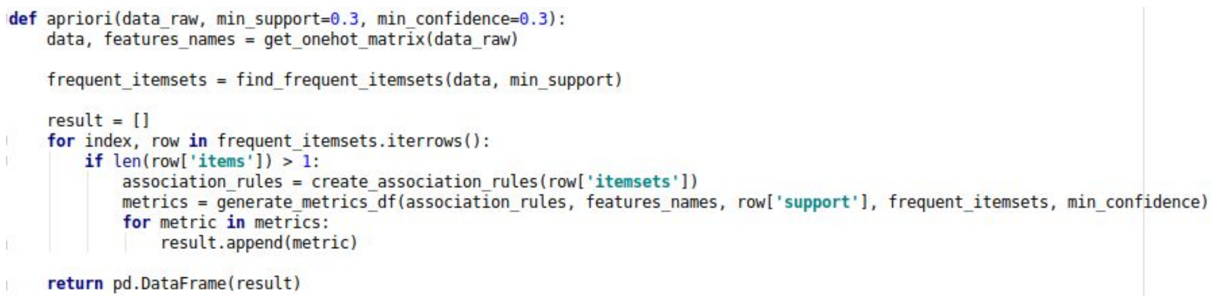
$$kulczynski = 0.5 * (conf(A \Rightarrow B) + conf(B \Rightarrow A))$$

If close to 0 or 1 then it is an interesting rule, negatively or positively correlated respectively, but a rule with kulczynski close to 0.5 might be interesting as well, depending on the ratio imbalance.

- **Ratio Imbalance**
  Measures the imbalance of two itemsets in rule implications, where 0 is perfectly balanced and 1 is very skewed.

$$IR(A, B) = \frac{|supp(A) - supp(B)|}{supp(A) + supp(B) - supp(A \cup B)}$$

Kulczynski and ratio imbalance are used together in order to find interesting rules.



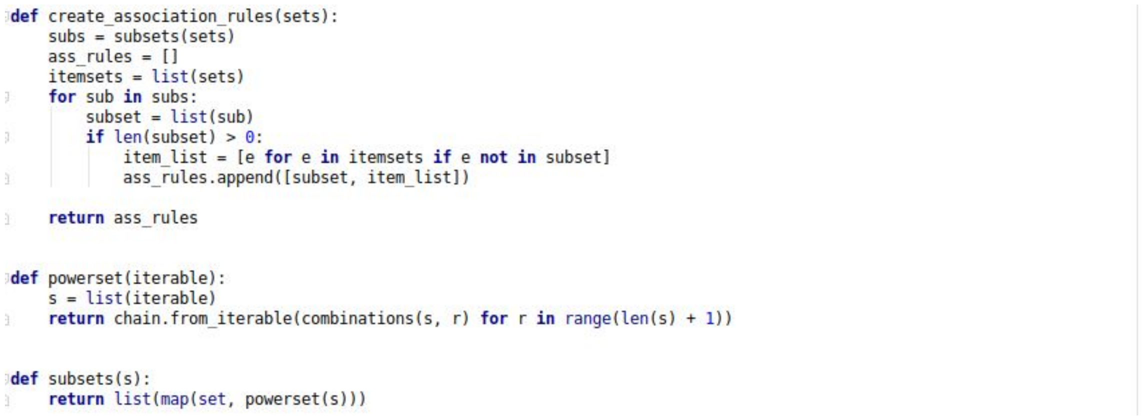
```
def apriori(data_raw, min_support=0.3, min_confidence=0.3):
    data, features_names = get_onehot_matrix(data_raw)

    frequent_itemsets = find_frequent_itemsets(data, min_support)

    result = []
    for index, row in frequent_itemsets.iterrows():
        if len(row['items']) > 1:
            association_rules = create_association_rules(row['itemsets'])
            metrics = generate_metrics_df(association_rules, features_names, row['support'], frequent_itemsets, min_confidence)
            for metric in metrics:
                result.append(metric)

    return pd.DataFrame(result)
```

This is the apriori general function. The main methods used will be described below.

The first step is to transform the data into a one-hot matrix. The data is presented as a json file with a list of all transactions, but we want to transform it into a one-hot matrix for convenience.

Next step is to find the frequent itemsets. To do so, we first compute the support of each individual itemset and then using the method *candidate_generation()* we also compute the support for all possible itemsets within our data, and save only the ones that meet with the defined minimum support.

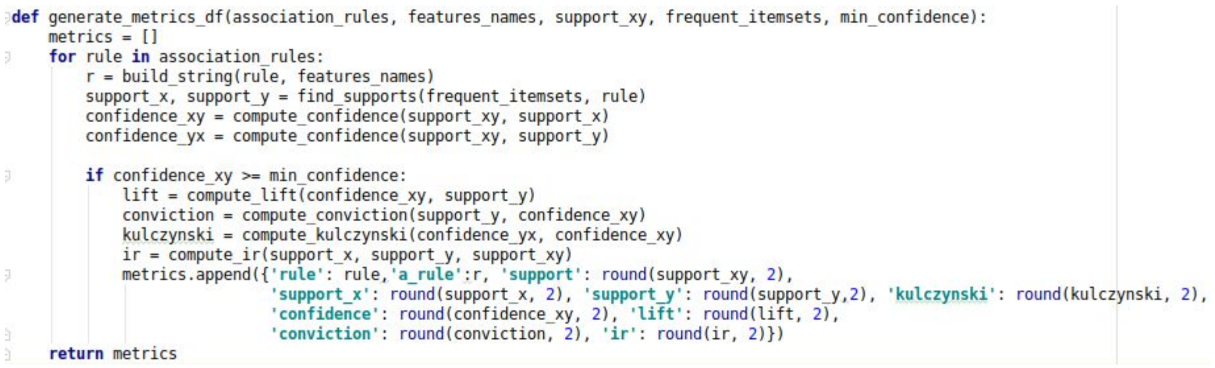
As mentioned before, once we have all the frequent itemsets, we iterate over them and if the itemset has more than one item, we create all possible association rules as follows:



```
def create_association_rules(sets):
    subs = subsets(sets)
    ass_rules = []
    itemsets = list(sets)
    for sub in subs:
        subset = list(sub)
        if len(subset) > 0:
            item_list = [e for e in itemsets if e not in subset]
            ass_rules.append([subset, item_list])

    return ass_rules


def powerset(iterable):
    s = list(iterable)
    return chain.from_iterable(combinations(s, r) for r in range(len(s) + 1))


def subsets(s):
    return list(map(set, powerset(s)))
```

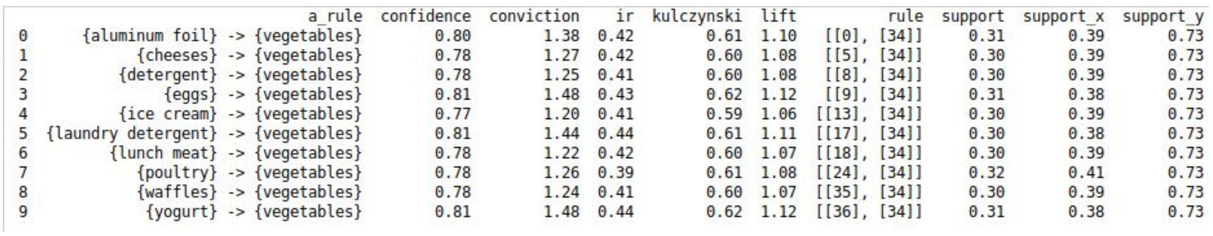
We first find all non-empty subsets of the given itemset (subs) and then for each non-empty subset s, we create the rule {s} -> {itemset - s}.

Finally, for each association rule of the given itemset, we compute all the different metrics mentioned before:



```
def generate_metrics_df(association_rules, features_names, support_xy, frequent_itemsets, min_confidence):
    metrics = []
    for rule in association_rules:
        r = build_string(rule, features_names)
        support_x, support_y = find_supports(frequent_itemsets, rule)
        confidence_xy = compute_confidence(support_xy, support_x)
        confidence_yx = compute_confidence(support_xy, support_y)

        if confidence_xy >= min_confidence:
            lift = compute_lift(confidence_xy, support_y)
            conviction = compute_conviction(support_y, confidence_xy)
            kulczynski = compute_kulczynski(confidence_yx, confidence_xy)
            ir = compute_ir(support_x, support_y, support_xy)
            metrics.append({'rule': rule,'a_rule':r, 'support': round(support_xy, 2),
                            'support_x': round(support_x, 2), 'support_y': round(support_y,2), 'kulczynski': round(kulczynski, 2),
                            'confidence': round(confidence_xy, 2), 'lift': round(lift, 2),
                            'conviction': round(conviction, 2), 'ir': round(ir, 2)})
    return metrics
```

So in the end we will have something like this:



```
                           a_rule  confidence  conviction    ir  kulczynski  lift        rule  support  support_x  support_y
0      {aluminum foil} -> {vegetables}       0.80        1.38  0.42        0.61  1.10   [[0], [34]]     0.31       0.39       0.73
1            {cheeses} -> {vegetables}       0.78        1.27  0.42        0.60  1.08   [[5], [34]]     0.30       0.39       0.73
2          {detergent} -> {vegetables}       0.78        1.25  0.41        0.60  1.08   [[8], [34]]     0.30       0.39       0.73
3               {eggs} -> {vegetables}       0.81        1.48  0.43        0.62  1.12   [[9], [34]]     0.31       0.38       0.73
4          {ice cream} -> {vegetables}       0.77        1.20  0.41        0.59  1.06  [[13], [34]]     0.30       0.39       0.73
5  {laundry detergent} -> {vegetables}       0.81        1.44  0.44        0.61  1.11  [[17], [34]]     0.30       0.38       0.73
6         {lunch meat} -> {vegetables}       0.78        1.22  0.42        0.60  1.07  [[18], [34]]     0.30       0.39       0.73
7            {poultry} -> {vegetables}       0.78        1.26  0.39        0.61  1.08  [[24], [34]]     0.32       0.41       0.73
8            {waffles} -> {vegetables}       0.78        1.24  0.41        0.60  1.07  [[35], [34]]     0.30       0.39       0.73
9             {yogurt} -> {vegetables}       0.81        1.48  0.44        0.62  1.12  [[36], [34]]     0.31       0.38       0.73
```

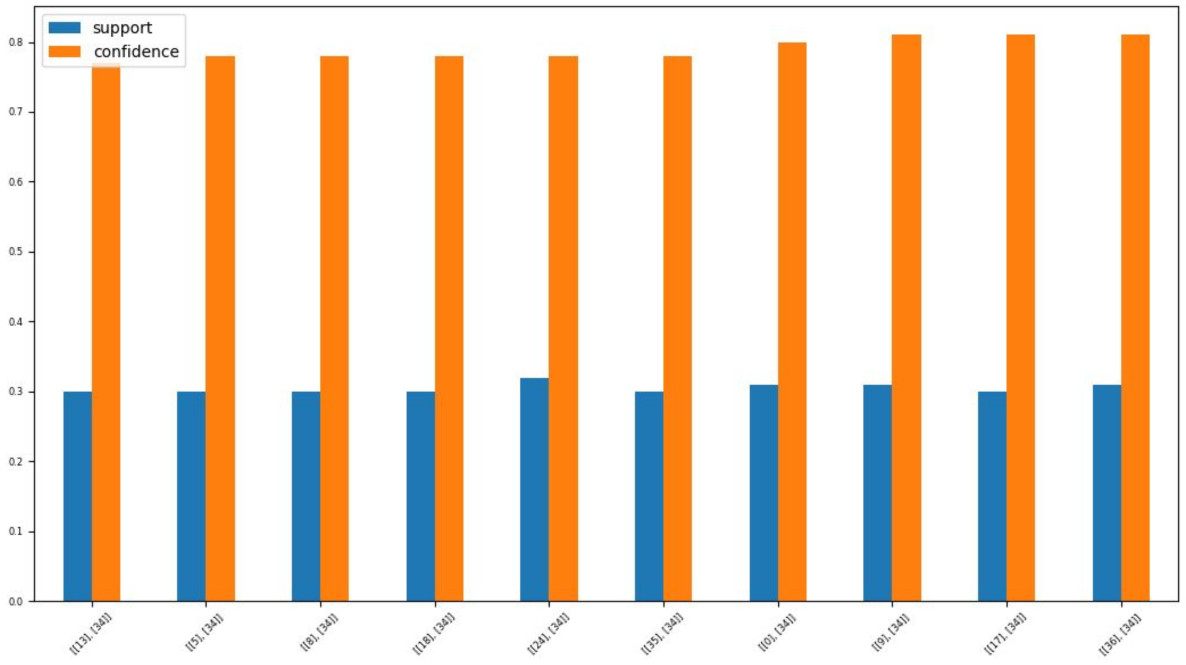
Evaluation of Apriori

This algorithm runs in a reasonable time for minimum support > 0.1. Otherwise it is very time consuming.

Alberto Martin Belloso albm@itu.dk
Aleksander Kähler akae@itu.dk
Rasmus Emil Odgaard raod@itu.dk

In order to evaluate the algorithm, we have mainly focused on Kulczynski and Imbalance Ratio metrics. This is because lift is not null invariant. This means that it varies depending on the number of null transactions. A null transaction is a transaction in which none of the items of an itemset are present.

So we are going to base our evaluation on Imbalance Ratio because it is null invariant.



```
                      a_rule  confidence  conviction    ir  kulczynski  lift          rule  support  support_x  support_y
4        {ice cream} -> {vegetables}        0.77        1.20  0.41        0.59  1.06  [[13], [34]]     0.30       0.39       0.73
1          {cheeses} -> {vegetables}        0.78        1.27  0.42        0.60  1.08   [[5], [34]]     0.30       0.39       0.73
2        {detergent} -> {vegetables}        0.78        1.25  0.41        0.60  1.08   [[8], [34]]     0.30       0.39       0.73
6       {lunch meat} -> {vegetables}        0.78        1.22  0.42        0.60  1.07  [[18], [34]]     0.30       0.39       0.73
7          {poultry} -> {vegetables}        0.78        1.26  0.39        0.61  1.08  [[24], [34]]     0.32       0.41       0.73
8          {waffles} -> {vegetables}        0.78        1.24  0.41        0.60  1.07  [[35], [34]]     0.30       0.39       0.73
0    {aluminum foil} -> {vegetables}        0.80        1.38  0.42        0.61  1.10   [[0], [34]]     0.31       0.39       0.73
3             {eggs} -> {vegetables}        0.81        1.48  0.43        0.62  1.12   [[9], [34]]     0.31       0.38       0.73
5 {laundry detergent} -> {vegetables}       0.81        1.44  0.44        0.61  1.11  [[17], [34]]     0.30       0.38       0.73
9           {yogurt} -> {vegetables}        0.81        1.48  0.44        0.62  1.12  [[36], [34]]     0.31       0.38       0.73
```

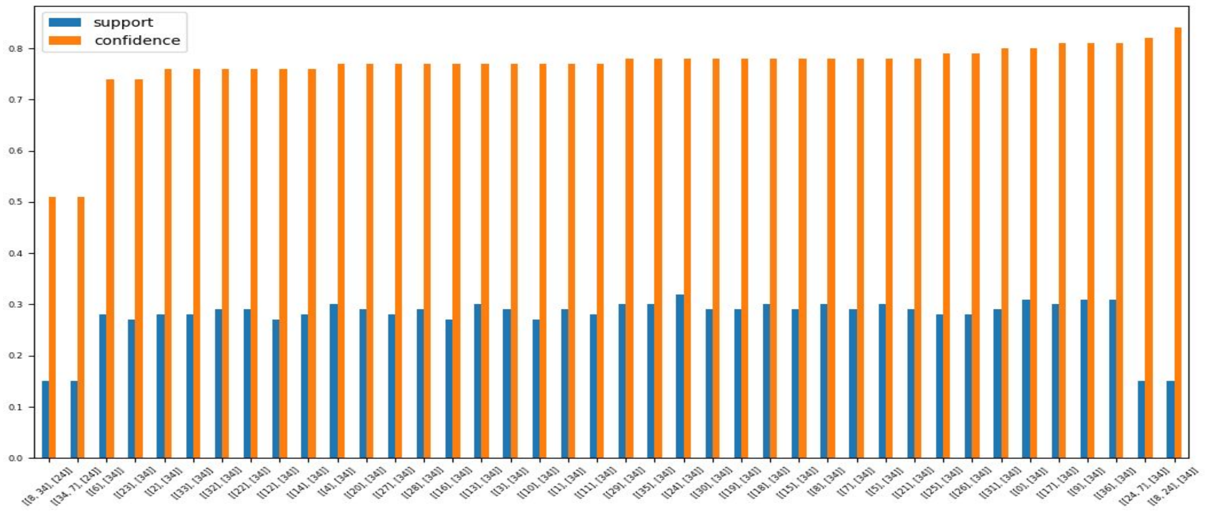
Execution 1: min_support=0.3 min_confidence=0.5 ordered by confidence

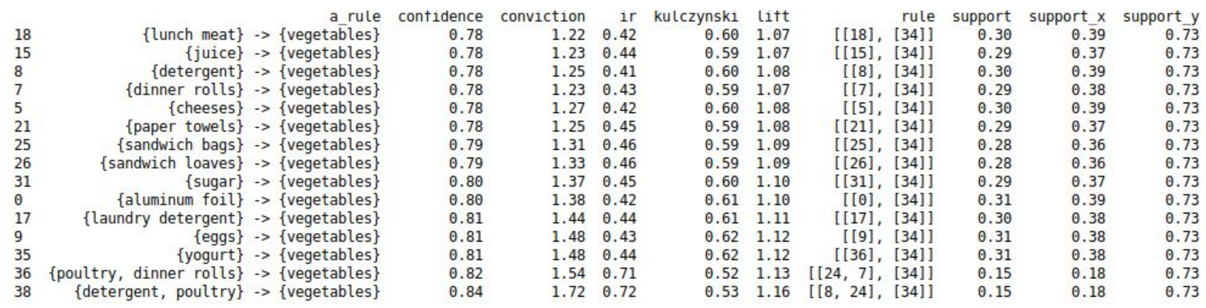
For this execution we can see that the 3 most interesting rules in terms of support-confidence are {eggs} -> {vegetables}, {laundry detergent}->{vegetables} and {yogurt}->{vegetables}

Also, the lift for these rules is >1 so in theory they are interesting association rules. But as mentioned before this can be wrong due to null transactions.

In this case, these rules are also interesting in terms of kulczynski-imbalance ratio because the kulczynski metric tells us that the association rules are positively correlated and their imbalance ratio is > 0. So we can say that these rules (and all the ones in the image) can be interesting.

Alberto Martin Belloso albm@itu.dk
Aleksander Kähler akae@itu.dk
Rasmus Emil Odgaard raod@itu.dk

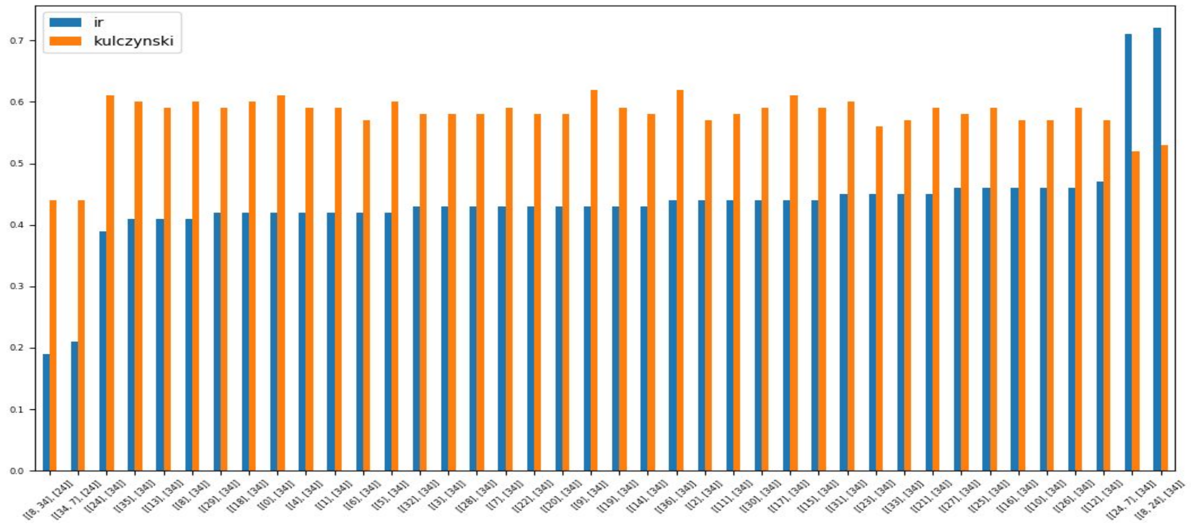




```
                           a_rule  confidence  conviction    ir  kulczynski  lift            rule  support  support_x  support_y
18          {lunch meat} -> {vegetables}        0.78        1.22  0.42        0.60  1.07    [[18], [34]]     0.30       0.39       0.73
15               {juice} -> {vegetables}        0.78        1.23  0.44        0.59  1.07    [[15], [34]]     0.29       0.37       0.73
8            {detergent} -> {vegetables}        0.78        1.25  0.41        0.60  1.08     [[8], [34]]     0.30       0.39       0.73
7         {dinner rolls} -> {vegetables}        0.78        1.23  0.43        0.59  1.07     [[7], [34]]     0.29       0.38       0.73
5              {cheeses} -> {vegetables}        0.78        1.27  0.42        0.60  1.08     [[5], [34]]     0.30       0.39       0.73
21        {paper towels} -> {vegetables}        0.78        1.25  0.45        0.59  1.08    [[21], [34]]     0.29       0.37       0.73
25       {sandwich bags} -> {vegetables}        0.79        1.31  0.46        0.59  1.09    [[25], [34]]     0.28       0.36       0.73
26     {sandwich loaves} -> {vegetables}        0.79        1.33  0.46        0.59  1.09    [[26], [34]]     0.28       0.36       0.73
31               {sugar} -> {vegetables}        0.80        1.37  0.45        0.60  1.10    [[31], [34]]     0.29       0.37       0.73
0        {aluminum foil} -> {vegetables}        0.80        1.38  0.42        0.61  1.10     [[0], [34]]     0.31       0.39       0.73
17   {laundry detergent} -> {vegetables}        0.81        1.44  0.44        0.61  1.11    [[17], [34]]     0.30       0.38       0.73
9                 {eggs} -> {vegetables}        0.81        1.48  0.43        0.62  1.12     [[9], [34]]     0.31       0.38       0.73
35              {yogurt} -> {vegetables}        0.81        1.48  0.44        0.62  1.12    [[36], [34]]     0.31       0.38       0.73
36  {poultry, dinner rolls} -> {vegetables}     0.82        1.54  0.71        0.52  1.13  [[24, 7], [34]]    0.15       0.18       0.73
38     {detergent, poultry} -> {vegetables}     0.84        1.72  0.72        0.53  1.16  [[8, 24], [34]]    0.15       0.18       0.73
```

Execution 2: min_support=0.15 min_confidence=0.5 ordered by confidence

Here, by reducing the minimum support, we can find other interesting rules with very high confidence, high imbalance ratio and positively correlated. So definitely we can say that all the rules that we can see in the image (and many more) are also interesting.



Finally, in this chart we can see the last execution but displaying the values of IR and Kul. As we can see, almost all the rules are positively correlated and have >0 IR, so they are interesting.

As a final comment, all rules we can see in this evaluation are {A} -> {vegetables}. This is because our confidence threshold is 0.5 and vegetables is the most common item in the database, but if we

Alberto Martin Belloso albm@itu.dk
Aleksander Kähler akae@itu.dk
Rasmus Emil Odgaard raod@itu.dk

reduce this threshold we can also find more interesting rules other than the ones involving vegetables.

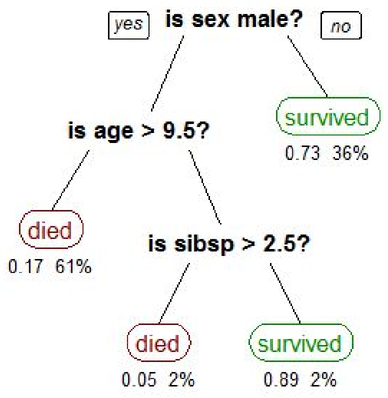
CART

Implemented by Aleksander Kähler
Data chosen: telecom_churn.csv

Description of CART

CART stands for Classification And Regression Tree, which is a type of decision tree typically used for solving classification problems in computer science and falls under the category of supervised learning. Via a decision tree it is possible to create a predictive model from observations about an item to conclusions about the item's target value. This is typically represented in branches and leaves respectively. In classification trees, these leaves and branches represent class labels and conjectures of features, which lead to the class labels.

One of the crucial things in creating a decision tree is to identify when to create a split by asking the right question. Most decision tree algorithms do this by examining the entire data set for possible split options, and calculate the most effective option. The split will then result in 2 nodes, from the parent. This can effectively be illustrated as a diagram as illustrated in figure below.



[1]

The same process is then repeated for each child node, where all possible options are evaluated in order to identify the best split based on the pre-set criterion.

Decision Trees, such as the CART, is typically created via a recursive algorithm where the algorithm starts at the root, and repeats splitting, until splitting no longer adds value to the predictions at each node. This process is called a *top-down induction decision tree* (TDIT) and is an example of a greedy algorithm. The data used to create the decision tree is typically in the form of:

$$(x, Y) = (x_1, x_2, x_3, ..., x_k, Y)$$

where *Y*, is the target variable which we are trying to classify. The vectors *x*, are the features used for this task.

Metrics

As mentioned before, it is important for the decision tree to know when to split based on the right question. For this, CART uses a method called Gini impurity to calculate how often a randomly chosen element from the data set would be incorrectly labeled if it was randomly labeled according

[1] Figure from https://en.wikipedia.org/wiki/Decision_tree_learning

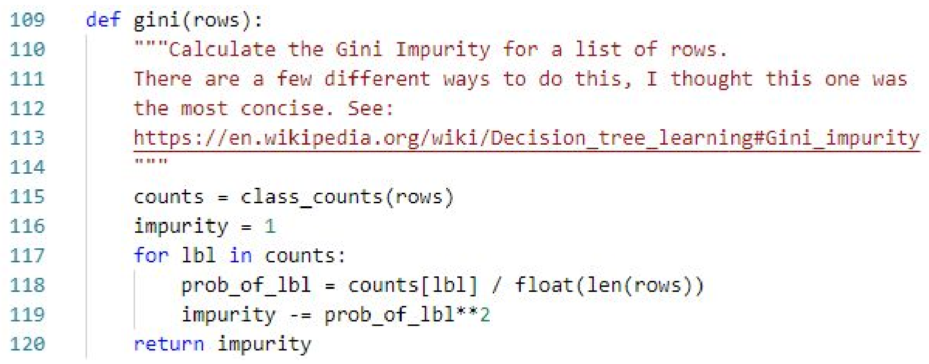
to the distribution of the labels in the data set. This can be computed by summing the probability p_i of an item, where i is the label of that item. This can be written as;

$$\sum_{k \neq i} p_k = 1 - p_i$$

This will reach its minimum, when all cases in the node falls into a single target category. To compute the Gini impurity for a set of a set of items in a data set, we can define *i* within a certain set of *J* classes, $i\{1, 2, 3, ..., J\}$ where p_i is part of the items labeled with the class i in the set. We can then compute the Gini impurity as;

$$I_G(p) = \sum_{i=1}^{J} p_i \sum_{k \neq i} p_k = \sum_{i=1}^{J} (1 - p_i) = \sum_{i=1}^{J} (p_i - p_i^2) = \sum_{i=1}^{J} p_i - \sum_{i=1}^{J} p_i^2 = 1 - \sum_{i=1}^{J} p_i^2 \ .$$

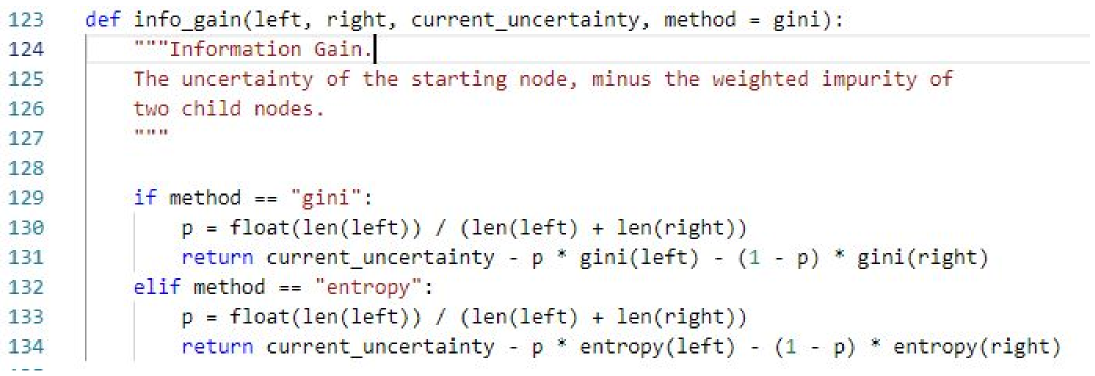
We compute this with python in the following code snippet;



```
109  def gini(rows):
110      """Calculate the Gini Impurity for a list of rows.
111      There are a few different ways to do this, I thought this one was
112      the most concise. See:
113      https://en.wikipedia.org/wiki/Decision_tree_learning#Gini_impurity
114      """
115      counts = class_counts(rows)
116      impurity = 1
117      for lbl in counts:
118          prob_of_lbl = counts[lbl] / float(len(rows))
119          impurity -= prob_of_lbl**2
120      return impurity
```

In line 115 and 118, the probability of any label within a certain set is calculated, whereafter in line 119 the Gini impurity is calculated by p_i^2 .

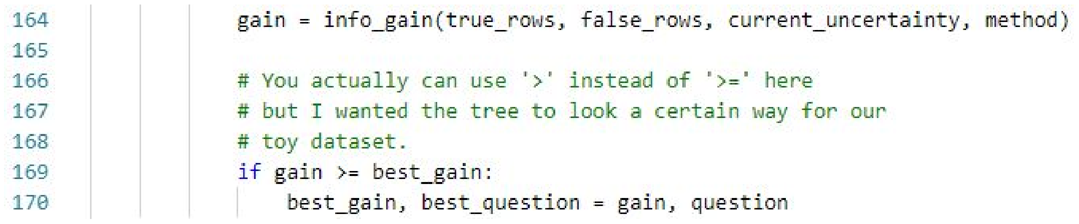
The Gini impurity is then used in order to calculate the information gain, which is used to decide which feature to split at each step of the tree.



```
123  def info_gain(left, right, current_uncertainty, method = gini):
124      """Information Gain.
125      The uncertainty of the starting node, minus the weighted impurity of
126      two child nodes.
127      """
128
129      if method == "gini":
130          p = float(len(left)) / (len(left) + len(right))
131          return current_uncertainty - p * gini(left) - (1 - p) * gini(right)
132      elif method == "entropy":
133          p = float(len(left)) / (len(left) + len(right))
134          return current_uncertainty - p * entropy(left) - (1 - p) * entropy(right)
```

The previous code snippet, we implement information gain in line 130-131 for Gini impurity and 133-134 for Entropy. The weighted impurity of the two child nodes is the value p, which is used in the next line.

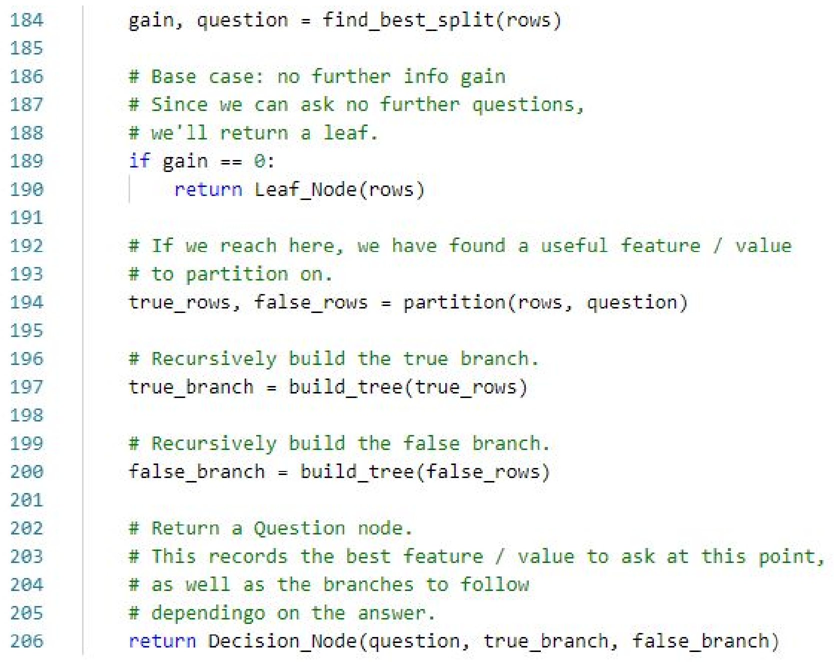
These two methods are called from the method find_best_split(rows), where the information gain is used to find the best gain, as illustrated in line 164 and 169-170 respectively.



```
            gain = info_gain(true_rows, false_rows, current_uncertainty, method)

            # You actually can use '>' instead of '>=' here
            # but I wanted the tree to look a certain way for our
            # toy dataset.
            if gain >= best_gain:
                best_gain, best_question = gain, question
```

As previously mentioned, decision trees are typically built through recursive methods, which is something that I have also used in this assignment. The method build_tree(rows), is the recursive method, which is repetitively called until the information gain is equal to zero, which means we have reached a leaf, or we reach a decision node, which is node where we want to ask a question regarding a feature.

This is implemented as shown in the code snippet below.



```
184    gain, question = find_best_split(rows)
185
186    # Base case: no further info gain
187    # Since we can ask no further questions,
188    # we'll return a leaf.
189    if gain == 0:
190        return Leaf_Node(rows)
191
192    # If we reach here, we have found a useful feature / value
193    # to partition on.
194    true_rows, false_rows = partition(rows, question)
195
196    # Recursively build the true branch.
197    true_branch = build_tree(true_rows)
198
199    # Recursively build the false branch.
200    false_branch = build_tree(false_rows)
201
202    # Return a Question node.
203    # This records the best feature / value to ask at this point,
204    # as well as the branches to follow
205    # dependingo on the answer.
206    return Decision_Node(question, true_branch, false_branch)
```

In line 189 we check for the information gain calculated in 184, and return an end node, if the gain is zero. If the gain is above zero, we continue to line 194 where the method partition is called, which checks for all the current rows, how they should be split depending on the current question.

The build_tree method is then recursively called again for true_rows and false_rows in line 197 and 200 repstively. Once the rows are iterated through and added to the tree, a Decision_Node is added with the corresponding question and references to the true_branch and false_branch children.

Evaluation of CART

As there are no specific method specified in the lecture slides on how to evaluate the CART algorithm, I've chosen to utilize a k-fold Cross Validation technique. The data used is telecom_churn, which contains data regarding customer churn rate and their previous telephone history suchs call minutes, call amount, etc.

With k-fold I divided the data set up into k=10 bins, which utilized the following result:

Method: entropy, k-Fold = 10, Execution time: 1952.32 sec.

k-Bin	**Accuracy**
1	0.90
2	0.91
3	0.91
4	0.91
5	0.93
6	0.90
7	0.92
8	0.92
9	0.94
10	0.89
Average accuracy:	**0.91 (+/- 0.02)**

For testing against the Gini impurity, I performed similar test.

Method: Gini, k-Fold = 10, Execution time: 3051.71 sec.

k-Bin	**Accuracy**
1	0.93
2	0.93
3	0.92
4	0.92
5	0.91
6	0.91
7	0.91
8	0.90
9	0.92
10	0.89
Average accuracy	**0.92 (+/- 0.01)**

There are a few differences between Gini and Entropy - Entropy performs slightly worse with a lower accuracy and a higher deviation. However, it is important to note that Entropy had a significantly lower execution time, due to the lower requirements of not having to compute the binary log as with Gini. Overall, the results could be improved through pruning and boosting, but for a minimum viable version of a CART algorithm, this result is acceptable.

k-means

Implemented by: Rasmus Emil Odgaard
Data chosen: fifa.csv

Theory

When performing unsupervised learning you are basically training your algorithm on data with no predefined labels. However in order to divide multiple data points into groups to detect potential tendencies, you will need to create clusters of data similar to each other. This is what a clustering algorithm does. There are multiple different ways of performing such clustering, depending on the goal of the data analysis. Common reasons for using clustering algorithms include getting an insight into the data set, however, it can also be used as a preprocessing step for identifying the labels before processing your data with a supervised learning approach

K-means

K-means is one such clustering algorithm. It works by representing the column features of the data (e.g. number of sales, month, age) as n-dimensional sets or objects, where n is the number of features. It is important however when working with numerical variables that the data as normalized, because otherwise the data could be skewed if a feature where to for instance contain values larger than the rest of the features.

The basics of the algorithm is calculating center of mass of potential clusters based on the distance between the different data points. This center of mass variable is named the centroid for the cluster. The computation is done by for each data object find the closest centroid by means of a distance formula. Often euclidean distance is used, but other distance measures such as Manhattan distance are used as well. When this clustering is done, the new position of the centroids are then defined by averaging all data objects classified to belong to its cluster. The process is then repeated until the centroids does not change their position and these clusters are the output of the algorithm. Below an example is written as pseudo-code.

Pseudo-code

```
1. Normalize data
2. Initiate random centroids
3. For each data object:
    a. Calculate distance to all centroids
    b. Associate to centroid with shortest distance
4. For each centroid:
    a. Calculate the average of all associated data objects
    b. If new centroids are equal to previous centroids
        i. Return
    c. Else
        i. Iterate step 3 and 4
```

Advantages and disadvantages

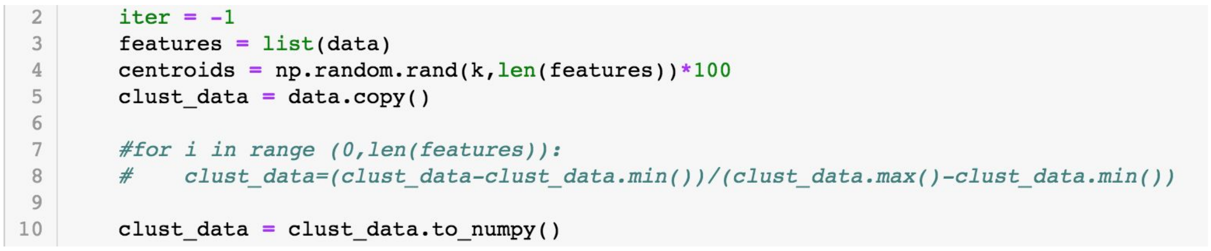
One major advantage of the k-means algorithm is that it is relatively efficient having a complexity of O(tkn), where n is the number of objects, k is the number of clusters, and t is how many iterations it takes to converge.

Disadvantages with k-means include k having to be defined before running the algorithm, which is impractical as you usually do not know the amount of clusters present in your data and as such will have to experiment. Furthermore, having the initial centroids be random can provide insufficient results such as finding a local optimum or placements of said centroids can be away from the bulk of the data objects resulting in empty clusters. The k-medoids algorithm accounts for the empty clusters by having the initial centroids be actual data points, however it is not as efficient. Another disadvantage of the k-means is that it is sensitive to outliers and noise.

Implementation

The implementation of the k-means algorithm was performed in the Jupyter python environment by the use of the NumPy and Pandas libraries for data handling, SciKit Learn (2011) for preprocessing and evaluation and Matplotlib and Seaborn for graphical representation. Testing was performed on the fifa.csv dataset. The implementation itself is based on the tutorial by Kinsley, H. (2016).

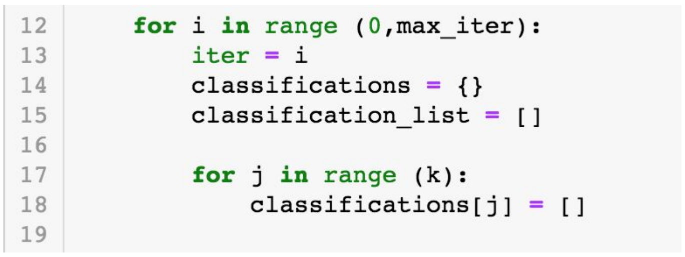
The first part of the implementation line 2-10 is basically preprocessing including copying the dataset to use when calculating the centroids and initializing the random centroids in a numpy array. The data is also normalized between 0 and 1 however since the fifa.csv data is normalized between 0 and 100 the normalization is outcommented and the centroids are instead multiplied by 100.



```
 2      iter = -1
 3      features = list(data)
 4      centroids = np.random.rand(k,len(features))*100
 5      clust_data = data.copy()
 6
 7      #for i in range (0,len(features)):
 8      #    clust_data=(clust_data-clust_data.min())/(clust_data.max()-clust_data.min())
 9
10      clust_data = clust_data.to_numpy()
```

Initialization of centroids and normalization

In line 12 the iterations of the algorithm starts running. The next 6 line is then spend resetting the dictionary of list containing the dataset subdivided by classification and a separate list for keeping the classifications in the correct order in regards to the feature sets. The dictionary makes sense here as the feature sets or data objects can be stored with the classification as key.



```
12      for i in range (0,max_iter):
13          iter = i
14          classifications = {}
15          classification_list = []
16
17          for j in range (k):
18              classifications[j] = []
19
```

Starting and resetting of the iterations

The distances from each feature set to each of the centroids are then calculated using the norm function of NumPy's linear algebra library. The smallest of these distances is then saved as the classification of said feature set, is appended into the list of classifications in order and the feature set itself is added to the classifications dictionary with the classification as key.

```
20         for index, featureset in enumerate(clust_data):
21             distances = [np.linalg.norm(featureset-centroids[c]) for c in range(k)]
22             classification = distances.index(min(distances))
23             classification_list.append(classification)
24             classifications[classification].append(featureset)
25
```

Finding the closest centroid for each of the feature sets

In the last section of the code the new centroids are calculated (line 30), however, for the algorithm to be stable a check for if the clusters are larger than zero is run to avoid dividing by zero in the case of a centroid that is not closest to any feature sets.

Another thing that happens in the code below is the comparison of the previous centroids with the current ones. If no changes have been made the clusters have reached their optimum and the algorithm terminates and returns the original data with a column added with the classifications of the feature sets and an array containing the position of the centroids. Have changes been made to the centroids however, the algorithm will run another iteration.

```
26         prev_centroids = np.copy(centroids)
27
28         for classification in classifications:
29             if (len(classifications[classification]) > 0):
30                 centroids[classification] = np.average(classifications[classification], axis=0)
31
32         if np.array_equal(prev_centroids, centroids):
33             data['Centroid'] = classification_list
34             break
35
36     print('Final iteration: ', iter)
37     data['Centroid'] = classification_list
38     return data, centroids
```

Calculation of new centroids and check for terminal condition

Evaluation of k-means

One method of evaluating the k-means algorithm is by the use of silhouette analysis. This is performed by checking how well the cluster is clustered with itself compared to the nearest cluster. Mathematically this is performed for every feature set *i* with the method:

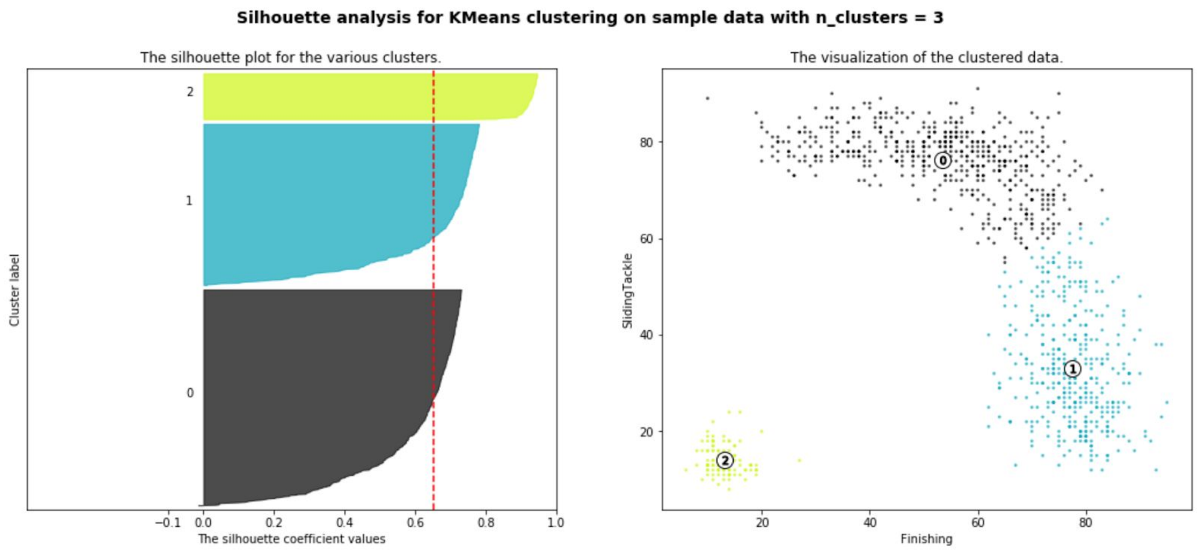
$$s(i) = \frac{b(i) - a(i)}{\max\{a(i), b(i)\}}$$

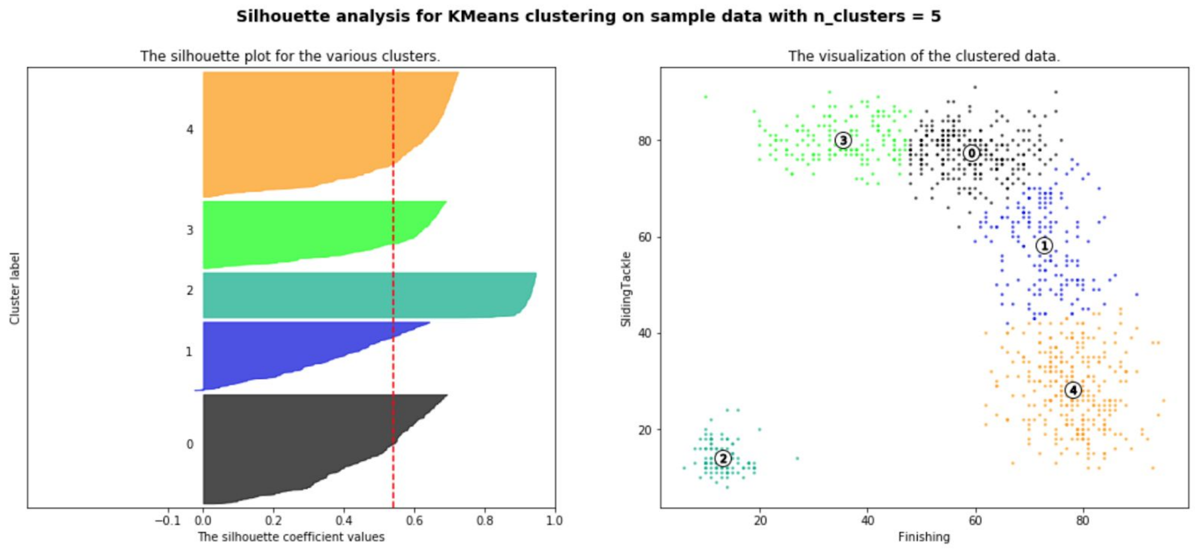
Where *s(i)* is the silhouette coefficient, *a(i)* is the mean distance from *i* to all data samples in it's own cluster and *b(i)* is the mean distance from *i* to all the samples of the closest neighboring cluster.

The silhouette coefficient are all in the range [-1,1] , where a point with a value of close to or equal to 1 will signify that the data object is similar to the ones in it's cluster and dissimilar to other clusters, where a value of close or equal to -1 will signify that the data object is incorrectly classified.

Apart from the individual silhouette coefficients a mean of all silhouette coefficients is often calculated to show the overall strength of the cluster classification performed, this value is sometimes referred to as the average silhouette score.

Apart from showing the strength and value of the clusterings, the silhouette analysis can be used to determine the most suitable k number of clusters. Part of this process involves silhouette plots containing the silhouette coefficients for each cluster sorted from highest to lowest. This process has been performed for the k-means implemented for this assignment with k number of centroids 2 through to 5 with two features for ease of graphical representation. The two sets of plots below shows the strongest clustering (k = 3) and the weakest (k = 5).





For the clustering of the features "Finishing" and "SlidingTackle" all four k values actually provided relatively positive average silhouette scores as seen below:

```
For n_clusters = 2 The average silhouette_score is : 0.56
For n_clusters = 3 The average silhouette_score is : 0.65
For n_clusters = 4 The average silhouette_score is : 0.58
For n_clusters = 5 The average silhouette_score is : 0.52
```

However, if we examine the silhouette plot for 5 clusters, it can be determined that cluster number 2 is a very good cluster. All it's silhouette coefficient are way above the mean and the values seem to in general be very close to each other. This makes sense when looking at the corresponding scatterplot showing cluster 2 to be isolated from all the other clusters in the left corner. As such these are clearly a set of coherent data objects. This means that a subset of the fifa players have both a low score in the finishing and sliding tackle features, potentially goalkeepers. Cluster 1 on the other hand is less so. Appx. a fourth of the silhouette coefficients are above the mean and they all vary from around 0.6 to just below 0. Again when looking at the scatter plot this becomes apparent from the close positioning of the cluster in between two other clusters and how spread out the values are.

The plots for 3 clusters however, is stronger by both having a higher mean and all clusters having half or more of their silhouette coefficients above the mean. As such, based on the silhouette analysis, it would be viable to use three clusters when analyzing these features.

References

Kinsley, H. [Sentdex]. (2016, June 22). Custom K Means - Practical Machine Learning Tutorial with Python p.37 [Video File]. Retrieved from https://www.youtube.com/watch?list=PLQVvvaa0QuDfKTOs3Keq_kaG2P55YRn5v&v=H4JSN_99kig

Scikit-learn: Machine Learning in Python, Pedregosa *et al.*, JMLR 12, pp. 2825-2830, 2011.