In [1]:

```
import pandas as pd
import numpy as np

import matplotlib.pyplot as plt
import category_encoders as ce

from scipy.cluster.hierarchy import fcluster, linkage

from sklearn import datasets
from sklearn.cluster import KMeans
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA
from sklearn.cluster import AgglomerativeClustering

import tensorflow as tf
from keras.preprocessing.sequence import TimeseriesGenerator
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM

from datetime import datetime

import math
```

In [2]:

```
DATA_PATH = 'data/data_after_EDA.csv'
END_DATE = '04/24/2022'

# Display all of the columns when data are shown
pd.set_option('display.max_columns', 60)

plt.rcParams['figure.figsize'] = (16, 8)
pd.options.mode.chained_assignment = None
```

In [3]:

```
data =  pd.read_csv(DATA_PATH, sep=',', parse_dates=['doc_date', 'product_since'
], low_memory=False)
```

In [4]:

```
data.head()
```

Out[4]:

	bill_country	setting_currency_id	shop_basket_id	doc_date	exchange_currency_rate	origi
0	BG	1	1136409	2020-04-26	1.9558	
1	BG	1	1136409	2020-04-26	1.9558	
2	BG	1	1136409	2020-04-26	1.9558	
3	BG	1	1136409	2020-04-26	1.9558	
4	BG	1	1136409	2020-04-26	1.9558	

In [5]:

```
# We have clean data from EDA, there are no nulls, no errors when opening the fi
le.
print(data.isna().sum())
```

```
bill_country                     0
setting_currency_id              0
shop_basket_id                   0
doc_date                         0
exchange_currency_rate           0
original_currency_code           0
basket_total_price_with_vat      0
count_basket_items               0
basket_count_products            0
basket_type                      0
item_quantity                    0
item_type                        0
item_unit_price_with_vat         0
item_total_discount_with_vat     0
product_id                       0
product_code                     0
catalog_category_id              0
catalog_brand_id                 0
product_name                     0
product_status                   0
reviews_count                    0
reviews_average_score_price      0
reviews_average_score_quality    0
reviews_average_score_properties 0
reviews_average_score_overall    0
reviews_average_score            0
is_in_stock                      0
is_ended                         0
is_new                           0
is_boosted                       0
product_purchase_price           0
eshop_stock_count                0
is_fifo                          0
product_name_parameterize        0
product_since                    0
category                         0
tree_path                        0
category_name_parameterized      0
category_status                  0
catalog_segment_id               0
categories_ancestor_ids          0
categories_descendant_ids        0
category_full_name_path          0
default_warranty_period          0
brand_name                       0
brand_parameterized              0
segment_name                     0
segment_parameterized            0
dtype: int64
```

1. Change non-numeric values to numbers

Machine learning models usually work only with numeric values (integers or floats) - that's why we need to change other formats to numbers.

In [6]:

```
data.dtypes
```

Out[6]:

```
bill_country                                object
setting_currency_id                          int64
shop_basket_id                               int64
doc_date                            datetime64[ns]
exchange_currency_rate                     float64
original_currency_code                      object
basket_total_price_with_vat                float64
count_basket_items                           int64
basket_count_products                        int64
basket_type                                 object
item_quantity                                int64
item_type                                   object
item_unit_price_with_vat                   float64
item_total_discount_with_vat               float64
product_id                                   int64
product_code                                 int64
catalog_category_id                        float64
catalog_brand_id                             int64
product_name                                object
product_status                              object
reviews_count                                int64
reviews_average_score_price                float64
reviews_average_score_quality              float64
reviews_average_score_properties           float64
reviews_average_score_overall              float64
reviews_average_score                      float64
is_in_stock                                   bool
is_ended                                      bool
is_new                                        bool
is_boosted                                    bool
product_purchase_price                     float64
eshop_stock_count                          float64
is_fifo                                       bool
product_name_parameterize                   object
product_since                       datetime64[ns]
category                                    object
tree_path                                   object
category_name_parameterized                 object
category_status                             object
catalog_segment_id                         float64
categories_ancestor_ids                     object
categories_descendant_ids                   object
category_full_name_path                     object
default_warranty_period                    float64
brand_name                                  object
brand_parameterized                         object
segment_name                                object
segment_parameterized                       object
dtype: object
```

At first let's start with breaking down dates to four different columns - we can extract day of the month, day of the week, week, month and year. We will still keep the original datetime column in code, because it can be useful to easier access date (rather then creating it from columns). \ We will also add an information about how long the item is available in eshop.

In [7]:

```
years, months, days, weeks, weekdays = [], [], [], [], []
for d_date in data.doc_date:
    years.append(d_date.year)
    months.append(d_date.month)
    days.append(d_date.day)
    weekdays.append(d_date.weekday())
    weeks.append(d_date.week)



data['doc_day'] = days
data['doc_month'] = months
data['doc_year'] = years
data['doc_weekday'] = weekdays
data['doc_week'] = weeks
```

Instead of date showing date when the item was added to the shop, we can just count days representing how long it's been available.

In [8]:

```
days_in_shop = []
for prod_date in data.product_since:
    days_in_shop.append((datetime.strptime(END_DATE, '%m/%d/%Y') - prod_date).da
ys)

data['days_in_shop'] = days_in_shop
```

The next part is to find columns that already have their natural number representation - i.e. product_name_parameterize is not necesarry column as we have product_id (numeric products identification)

In [9]:

```
# rename columns from format catalog_COLUMN to COLUMN only so it is easier to un
derstand
data.rename(columns={'catalog_category_id' : 'category_id', 'catalog_segment_id'
: 'segment_id', 'catalog_brand_id' : 'brand_id'}, inplace=True)

# rename other id columns with extra words to pure defining id in similar spirit
as with catalog
data.rename(columns={'setting_currency_id' : 'currency_id', 'shop_basket_id' :
'basket_id'}, inplace=True)
```

In [10]:

```
def leave_only_id_column(df : pd.DataFrame(), id_column : str, other_columns : list, inplace : bool = False) -> pd.DataFrame():
    """
    Function that counts and compares if id_columns is proper representation of other given columns. If yes, then drop other columns and leave id_column only.
    Args
        df - pandas DataFrame containing desired columns
        id_column - main column containing identificator, this column will be the only one remaining
        other_columns - list of other columns, those will be compared and possibly dropped
        inplace - If False, return a copy. Otherwise, do operation inplace and return None
    Returns
        pd.DataFrame - DataFrame with removed columns in other_columns or None if inplace is True
    """
    id_col_len = len(data[id_column].unique())

    # How many combinations of id_column -- other_columns there are. Idealy should be the same number as the number of unique ids.
    unique_combinations = len(df[other_columns + [id_column]].drop_duplicates().index)

    other_cols_string = ''
    for name in other_columns:
        other_cols_string += name+', '

    print(f"{id_col_len} - Unique {id_column} amount.")
    print(f"{unique_combinations} - Amount of unique combinations of {id_column} and {other_cols_string}")

    missmatches_amount = abs(id_col_len - unique_combinations)
    print(f"{missmatches_amount} - How many missmatches between {id_column} and other columns.")

    if missmatches_amount == 0:
        if inplace:
            df.drop(labels=other_columns, inplace=inplace, axis=1)
            return None
        else:
            return df.drop(labels=other_columns, inplace=inplace, axis=1)

    else:
        print('There were missmatches, not dropping any columns.')
        return None
```

Because each product id represents one product correctly, we can drop product name as well as parameterized product name. \ We can drop product_code as well for the same reason - product id represents same products as product_code but in different encodings.

In [11]:

```
leave_only_id_column(data, 'product_id', ['product_name', 'product_code'], inpla
ce=True)
print('\n')

leave_only_id_column(data, 'product_id', ['product_name_parameterize'], inplace=
True)
```

```
164386 - Unique product_id amount.
164386 - Amount of unique combinations of product_id and product_nam
e, product_code,
0 - How many missmatches between product_id and other columns.


164386 - Unique product_id amount.
164386 - Amount of unique combinations of product_id and product_nam
e_parameterize,
0 - How many missmatches between product_id and other columns.
```

Similar to products, there is many alike records in data (columns represented by other column), we can take care of all of them.

In [12]:

```
leave_only_id_column(data, 'category_id', ['category', 'category_name_parameteri
zed'], inplace=True)
```

```
2465 - Unique category_id amount.
2465 - Amount of unique combinations of category_id and category, ca
tegory_name_parameterized,
0 - How many missmatches between category_id and other columns.
```

In [13]:

```
leave_only_id_column(data, 'brand_id', ['brand_name', 'brand_parameterized'], in
place=True)
```

```
5326 - Unique brand_id amount.
5326 - Amount of unique combinations of brand_id and brand_name, bra
nd_parameterized,
0 - How many missmatches between brand_id and other columns.
```

In [14]:

```
leave_only_id_column(data, 'currency_id', ['original_currency_code'], inplace=Tr
ue)
```

```
10 - Unique currency_id amount.
10 - Amount of unique combinations of currency_id and original_curre
ncy_code,
0 - How many missmatches between currency_id and other columns.
```

In [15]:

```
leave_only_id_column(data, 'segment_id', ['segment_parameterized', 'segment_nam
e'], inplace=True)
```

```
15 - Unique segment_id amount.
15 - Amount of unique combinations of segment_id and segment_paramet
erized, segment_name,
0 - How many missmatches between segment_id and other columns.
```

In [16]:

```
leave_only_id_column(data, 'tree_path', ['category_full_name_path'], inplace=Tru
e)
```

```
2465 - Unique tree_path amount.
2465 - Amount of unique combinations of tree_path and category_full_
name_path,
0 - How many missmatches between tree_path and other columns.
```

In [17]:

```
# In tree path we want to keep it separated in two columns for now - categorty_d
escendants (parents) and category_ancestors (subcategories). We will check for m
issmatches and drop tree_path if there are none
leave_only_id_column(data, 'tree_path', ['categories_descendant_ids', 'categorie
s_ancestor_ids'])

data.drop(labels='tree_path', axis=1, inplace=True)
```

```
2465 - Unique tree_path amount.
2465 - Amount of unique combinations of tree_path and categories_des
cendant_ids, categories_ancestor_ids,
0 - How many missmatches between tree_path and other columns.
```

Coding remaining string and boolean values to numerics \ With the usage of replace (booleans) and OrdinalEncounter (strings) we will change values to their representation in numbers.

In [18]:

```
data.replace([True, False], [1, 0], inplace=True)
```

In [19]:

```
columns_to_change = ['bill_country',
 'basket_type',
 'item_type',
 'product_status',
 'category_status']

ce_ordinal = ce.OrdinalEncoder(cols=columns_to_change)
data = ce_ordinal.fit_transform(data)

for mapped in ce_ordinal.fit(data).mapping:
    print(f"Column {mapped['col']} has mapping of:")
    print(f"{mapped['mapping']} \n\n")
```

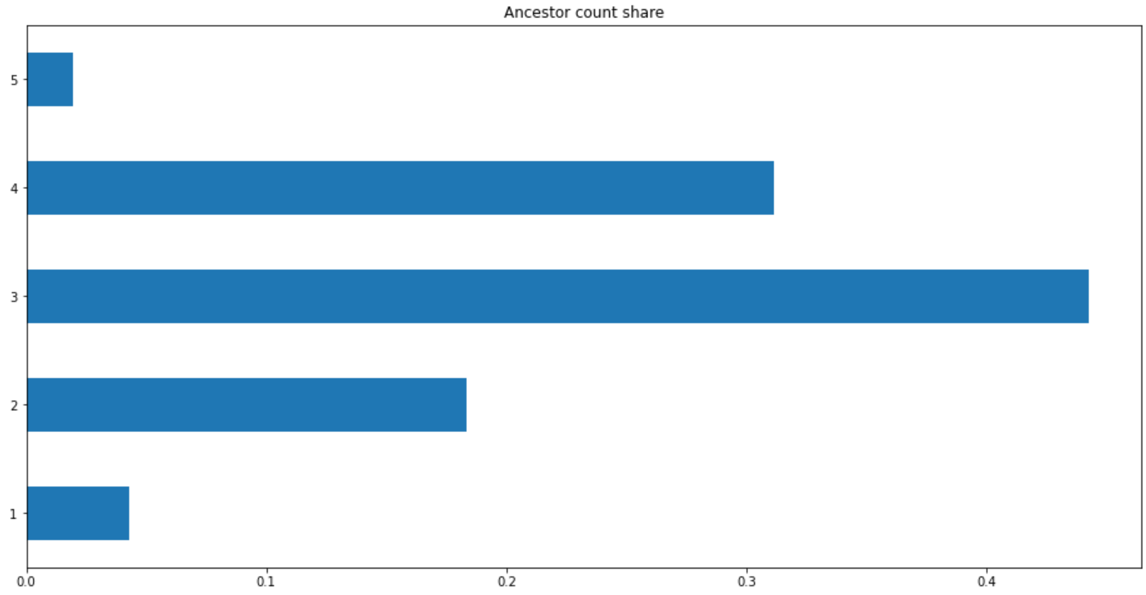
```
Column bill_country has mapping of:
BG      1
CZ      2
SK      3
PL      4
DE      5
SI      6
RO      7
FR      8
AT      9
HU     10
HR     11
IT     12
NL     13
DK     14
SE     15
BE     16
LT     17
GB     18
LV     19
IE     20
CH     21
PT     22
ES     23
LU     24
FI     25
EE     26
GR     27
EL     28
TR     29
UA     30
RS     31
BA     32
NaN    -2
dtype: int64


Column basket_type has mapping of:
standard            1
internal            2
express_checkout    3
NaN                -2
dtype: int64


Column item_type has mapping of:
standard           1
set                2
digital_licence    3
NaN               -2
dtype: int64


Column product_status has mapping of:
active      1
archived    2
ended       3
inactive    4
NaN        -2
dtype: int64
```

```
Column category_status has mapping of:
active      1
inactive    2
NaN        -2
dtype: int64
```

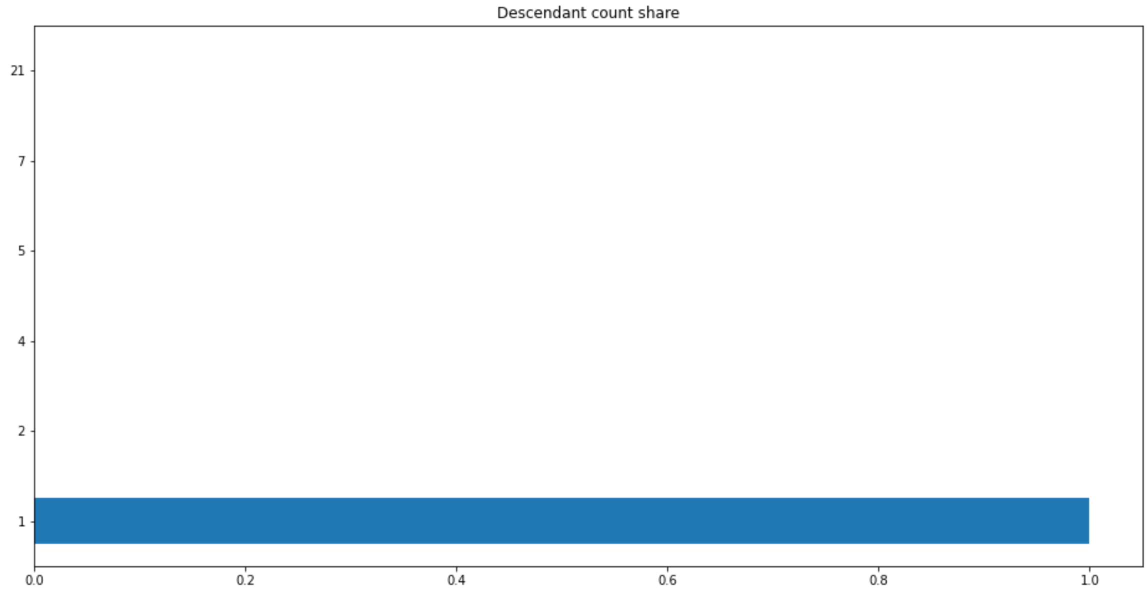
Instead of saving full path to category we just want to know how deep given category is and how many subcategories it has. We will convert arrays of ancestors/ descendants into numbers representing amounts of ids in given lists.

In [20]:

```
ancestor_count = [len(i.split(',')) for i in data.categories_ancestor_ids]
descendant_count = [len(i.split(',')) for i in data.categories_descendant_ids]

data['ancestor_count'] = ancestor_count
data['descendant_count'] = descendant_count

data.ancestor_count.value_counts(normalize=True).sort_index().plot(kind='barh',
title='Ancestor count share')
plt.show()

data.descendant_count.value_counts(normalize=True).sort_index().plot(kind='barh'
, title='Descendant count share')
plt.show()

data.drop(labels=['categories_ancestor_ids', 'categories_descendant_ids'], axis=
1, inplace=True)
```



Ancestor count share
5
4
3
2
1
0.0
0.1
0.2
0.3
0.4



Descendant count share
21
7
5
4
2
1
0.0
0.2
0.4
0.6
0.8
1.0

In [21]:

```
data.columns
```

Out[21]:

```
Index(['bill_country', 'currency_id', 'basket_id', 'doc_date',
       'exchange_currency_rate', 'basket_total_price_with_vat',
       'count_basket_items', 'basket_count_products', 'basket_type',
       'item_quantity', 'item_type', 'item_unit_price_with_vat',
       'item_total_discount_with_vat', 'product_id', 'category_id',
'brand_id',
       'product_status', 'reviews_count', 'reviews_average_score_pri
ce',
       'reviews_average_score_quality', 'reviews_average_score_prope
rties',
       'reviews_average_score_overall', 'reviews_average_score', 'is
_in_stock',
       'is_ended', 'is_new', 'is_boosted', 'product_purchase_price',
       'eshop_stock_count', 'is_fifo', 'product_since', 'category_st
atus',
       'segment_id', 'default_warranty_period', 'doc_day', 'doc_mont
h',
       'doc_year', 'doc_weekday', 'doc_week', 'days_in_shop', 'ances
tor_count',
       'descendant_count'],
      dtype='object')
```

2. Dealing with outliers

In this part we want to delete outliers, as those might negatively influenece machine learning algorithm. That is why we want to delete at least the first iteration of outliers. There is ~5% values as outliers in the first iteration, which, we consider, is reasonable price to pay for cleaner and more useful data.\ We are considering values further than *3x standard deviations (σ) from the mean (μ)* as outliers in our preprocessing.

In [22]:

```
def delete_outliers(df : pd.DataFrame) -> pd.DataFrame:
    """
    Function deletes rows containing outlier value in any of the columns and ret
urns adjusted dataframe
    Args
        df - dataframe containing columns to check for outliers
    Returns
        DataFrame without outlier values
    """
    for cols in df.columns:
        # Check for each column in the dataframe
        data_frame = df[cols]
        data_mean, data_std = np.mean(data_frame), np.std(data_frame)  # Outlier
> mean+3*std OR outlier < mean-3*std

        # Outliers percentage definition
        cut_off = data_std * 3
        lower, upper = data_mean - cut_off, data_mean + cut_off

        # Identify and remove outliers
        outliers = [False if x < lower or x > upper else True for x in data_fram
e]

        # Information for the user about deleting rows based on given column
        if outliers.count(False) > 0:
            print(f'Identified outliers: {outliers.count(False)} in column: {col
s}')
        df = df[outliers]

    return df
```

In [23]:

```
check_outliers_columns = ['basket_total_price_with_vat',
                          'count_basket_items',
                          'basket_count_products',
                          'item_quantity',
                          'item_unit_price_with_vat',
                          'item_total_discount_with_vat',
                          'reviews_count',
                          'reviews_average_score_price',
                          'reviews_average_score_quality',
                          'reviews_average_score_properties',
                          'reviews_average_score_overall',
                          'reviews_average_score',
                          'product_purchase_price',
                          'eshop_stock_count',
                          'ancestor_count',
                          'descendant_count']
```

In [24]:

```
for col in check_outliers_columns:
    data[col] = delete_outliers(data[[col]])
    data.dropna(inplace=True)
```

```
Identified outliers: 6 in column: basket_total_price_with_vat
Identified outliers: 16525 in column: count_basket_items
Identified outliers: 38830 in column: basket_count_products
Identified outliers: 42171 in column: item_quantity
Identified outliers: 50815 in column: item_unit_price_with_vat
Identified outliers: 37948 in column: item_total_discount_with_vat
Identified outliers: 17914 in column: reviews_count
Identified outliers: 70247 in column: product_purchase_price
Identified outliers: 14862 in column: eshop_stock_count
Identified outliers: 607 in column: descendant_count
```

3. Clustering

The last part of preprocessing in this notebook will be to cluster products into clusters. \ For this we are using KMeans clustering at first - to divide data into smaller clusters, which are then clustered again with the usage of Hierarchical clustering.

Total amount of clusters used now is 303 (101 * 3), but this can be easily changed in the code.

In [25]:

```
kmcluster_amount = 101
hierarchical_cluster_amount = 3
```

At first determine which columns are viable for the clustering. We want to cluster products with products, no sales - so we are choosing attributes of products from the DataFrame.

In [26]:

```
kmeansable = data[[
 'item_type',
 'product_id',
 'category_id',
 'brand_id',
 'product_status',
 'reviews_count',
 'reviews_average_score_price',
 'reviews_average_score_quality',
 'reviews_average_score_properties',
 'reviews_average_score_overall',
 'reviews_average_score',
 'is_in_stock',
 'is_ended',
 'is_new',
 'product_purchase_price',
 'eshop_stock_count',
 'is_fifo',
 'category_status',
 'segment_id',
 'default_warranty_period',
 'ancestor_count',
 'descendant_count',
 'days_in_shop'
]].drop_duplicates()
```

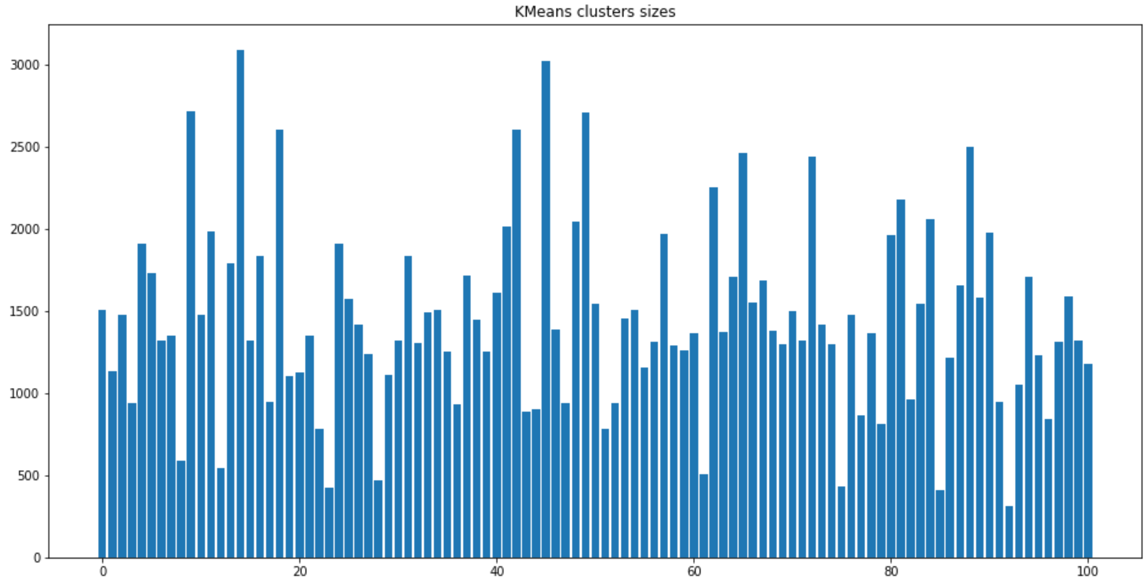
At first we will use kMeans clustering, as agglomerative clustering can be done on large dataset easier than other types.

In [27]:

```
# Declaring Model
model = KMeans(n_clusters=kmcluster_amount)

# Fitting Model
model.fit(kmeansable)

# Prediction on the entire data
all_predictions = model.predict(kmeansable)

kmeansable['kmeans_cluster'] = all_predictions
```

In [28]:

```
unique, counts = np.unique(all_predictions, return_counts = True)
plt.bar(unique, counts)
plt.title('KMeans clusters sizes')
plt.show()
```



In [29]:

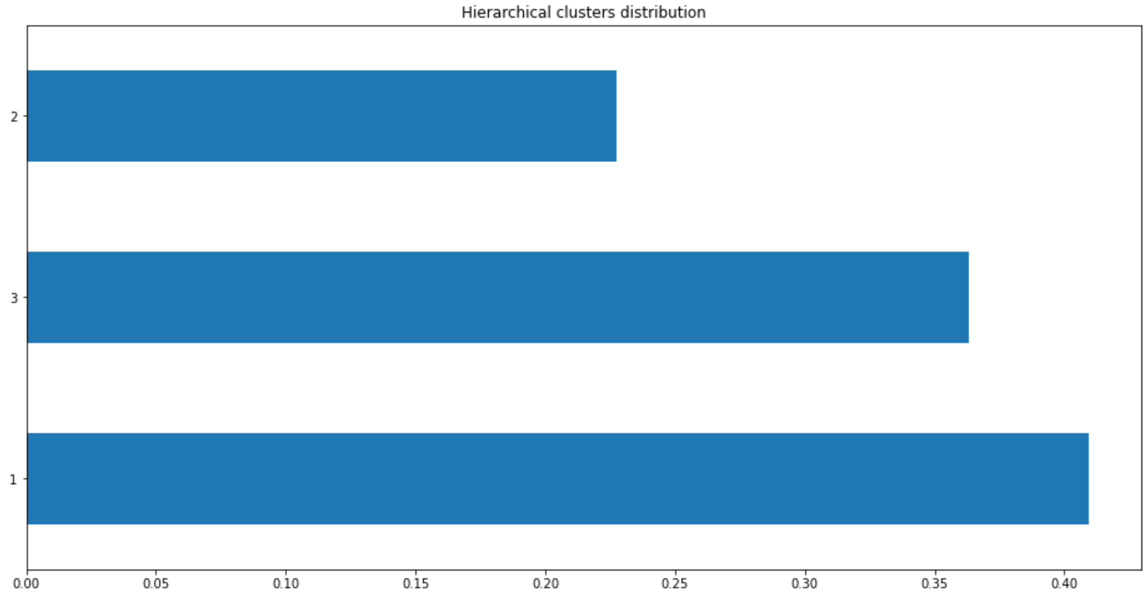
```
hier_id, hier_clust, k_clust = [], [], []
for k_cluster in range(kmcluster_amount):
    cluster_tester = kmeansable[kmeansable.kmeans_cluster.__eq__(k_cluster)]

    distance_matrix = linkage(cluster_tester, method = 'ward', metric = 'euclide
an')
    tmp = fcluster(distance_matrix, hierarchical_cluster_amount, criterion='maxc
lust')

    hier_id.extend(cluster_tester.product_id.values)
    hier_clust.extend(tmp)
    k_clust.extend([k_cluster for  i in range (len(tmp))])
```

In [30]:

```
hierarched = pd.DataFrame({'hier_id' : hier_id, 'cluster_hierar' : hier_clust,
'kmeans_cluster' : k_clust}).drop_duplicates()
```

In [31]:

```
hierarched.cluster_hierar.value_counts(normalize=True).plot(kind='barh', title=
'Hierarchical clusters distribution')
plt.show()
```



In [32]:

```
data = data.merge(hierarched, left_on='product_id', right_on='hier_id').drop(axi
s=1, labels=['hier_id'])
```

In [33]:

```
data
```

Out[33]:

	bill_country	currency_id	basket_id	doc_date	exchange_currency_rate	basket_tota
0	1	1	1136409	2020-04-26	1.9558	
1	1	1	1571607	2020-08-07	1.9558	
2	1	1	1661756	2020-08-29	1.9558	
3	1	1	1701910	2020-09-09	1.9558	
4	1	1	262709	2019-06-06	1.9558	
...	...	...	...	...	...	
2713505	12	6	3375295	2021-08-16	1.0000	
2713506	3	6	3057913	2022-02-03	1.0000	
2713507	9	6	3325485	2021-08-04	1.0000	
2713508	11	9	1728853	2020-09-17	7.5415	
2713509	2	4	4623935	2022-04-08	24.5120	

2713510 rows × 44 columns

This dataset is great now, as it is only containing numbers and dates - it shows no critical information too deeply.

In [34]:

```
data.to_csv('data/data_after_preprocessing.csv', index=False)
```