Classical Machine Learning Assignment

In [61]:
```
import pandas as pd
pd.set_option('display.precision', 2)
pd.set_option('display.max_columns', None)
pd.set_option('display.width', None)
```

We import `pandas`, a library for the manipulation of tabulated data in Machine Learning. We set a few options including putting `max_columns` to `None` so that all of them can be seen in the Jupyter notebook without having to select subsets of columns. This will mean the Jupyter notebook will be easier to read but that the pdf will have some columns cropped out. Please see, the Jupyter notebook if you want to see the full data displayed.

In [62]:
```
df = pd.read_csv('CML_2024-09-09#3-wheat_genome_1.csv')
df
```

Out[62]:

	ZYTI	WIBD	KVJI	CSTT	OFSQ	RXUY	GWEE	NANX	PMLC	JIEK	GXSO	PTAX	RLCO
0	7.12e+07	66780003.23249369	98615834.8241663	52159482.56117151	82784798.00617953	6.06e+07	7.50e+07	8.37e+07	5.02e+07	67148304.13857159	7.71e+07	8.27e+07	68347063.23106696
1	5.80e+07	52206572.93108849	98309387.02801965	64901682.897473596	57695764.36746716	6.96e+07	9.30e+07	9.51e+07	6.38e+07	91303795.85450855	1.12e+08	9.82e+07	64604823.260036215
2	8.93e+07	59058481.656934984	90340950.98559198	68251937.91355611	89272597.48780008	6.70e+07	8.26e+07	9.48e+07	6.09e+07	53998583.56195885	6.46e+07	8.04e+07	59588103.00683621
3	4.16e+07	87279722.31068213	101620049.21358526	90873708.86792825	94655232.50324139	4.40e+07	7.62e+07	6.79e+07	4.83e+07	68304099.42017053	8.87e+07	1.00e+08	70052071.93627621
4	9.06e+07	57388025.488219306	57568362.19853688	75526475.58607598	60713729.34205614	8.32e+07	8.56e+07	8.17e+07	9.73e+07	78009778.11793171	5.50e+07	5.34e+07	50981332.5625033
...	...	...	...	...	...	...	...	...	...	...	...	...	...
4995	7.41e+07	58262507.06873764	92701179.7179384	67796505.77780084	60996313.33816352	1.05e+08	3.67e+07	6.85e+07	9.24e+07	53654575.16920183	1.07e+08	9.00e+07	85839241.93596187
4996	6.63e+07	44913372.921941906	93559494.10271989	54065530.2835989	56083538.15939477	6.06e+07	7.73e+07	9.62e+07	6.80e+07	99407331.31693117	9.73e+07	9.74e+07	61454056.53538433
4997	9.74e+07	51594995.715354934	65552552.49277553	98008359.76560771	76960297.88906516	8.18e+07	6.96e+07	8.30e+07	8.47e+07	90739941.20183507	4.73e+07	7.65e+07	93275227.52404302
4998	5.86e+07	66843238.02653567	71275786.0738799	55771212.44004866	72847960.13200027	1.11e+08	5.37e+07	6.26e+07	9.16e+07	58169036.77614904	9.01e+07	8.24e+07	84858524.17996946
4999	5.58e+07	76902968.80040008	82714128.91204472	55603146.54898293	75568004.44787523	6.76e+07	7.99e+07	9.47e+07	4.97e+07	61735599.656930335	6.81e+07	9.00e+07	70151739.14748453

5000 rows × 45 columns

We read the Landrace Wheat dataset into `pandas` and through a sampling of the first 5 and last 5 rows we can see there are 40 genetic marker features and 5 response measurements. It looks like the features are floating point numbers in the millions, and the responses floating point numbers in the billions however given there are 5000 rows, too many to manually inspect, we need other ways to better understand the data.

In [63]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5000 entries, 0 to 4999
Data columns (total 45 columns):
 #   Column   Non-Null Count  Dtype  
---  ------   --------------  -----  
 0   ZYTI     5000 non-null   float64
 1   WIBD     4999 non-null   object 
 2   KVJI     5000 non-null   object 
 3   CSTT     5000 non-null   object 
 4   OFSQ     4998 non-null   object 
 5   RXUY     4999 non-null   float64
 6   GWEE     5000 non-null   float64
 7   NANX     4999 non-null   float64
 8   PMLC     4998 non-null   float64
 9   JIEK     5000 non-null   object 
 10  GXSO     4999 non-null   float64
 11  PTAX     5000 non-null   float64
 12  RLCO     5000 non-null   object 
 13  RNOZ     5000 non-null   object 
 14  QRWO     5000 non-null   float64
 15  PSSJ     5000 non-null   object 
 16  WBYY     4999 non-null   object 
 17  BDGK     5000 non-null   float64
 18  FWKW     4999 non-null   float64
 19  PFKF     5000 non-null   float64
 20  ZKOY     4998 non-null   float64
 21  XBUG     5000 non-null   float64
 22  OMXX     5000 non-null   float64
 23  IXKB     4999 non-null   float64
 24  YKJO     4999 non-null   float64
 25  GQUZ     4999 non-null   float64
 26  VKRW     4999 non-null   float64
 27  DDTT     4999 non-null   float64
 28  QXYM     5000 non-null   float64
 29  MHEC     4997 non-null   float64
 30  TUXT     4999 non-null   object 
 31  UZSD     4999 non-null   float64
 32  AOQU     5000 non-null   float64
 33  RJWA     5000 non-null   float64
 34  VTQI     5000 non-null   float64
 35  ROQE     4998 non-null   object 
 36  ZMHO     5000 non-null   object 
 37  RTDN     5000 non-null   object 
 38  MJXP     5000 non-null   object 
 39  XYKI     4998 non-null   float64
 40  RESP__0  4999 non-null   object 
 41  RESP__1  5000 non-null   object 
 42  RESP__2  4999 non-null   float64
 43  RESP__3  4999 non-null   float64
 44  RESP__4  4999 non-null   float64
dtypes: float64(29), object(16)
memory usage: 1.7+ MB
```

Using `.info` we can get a summary of the dataframe where we see that our initial impression of the data was inaccurate. Viewing as a scrollable element allows us to see all the data.

Firstly, we can see that some feature columns have `null` data such as `MHEC` which has 3 `null` data and 4997 `non-null`, not evident from our initial 10 row sampling of the data. Similarly, for the reponse variables `RESP__0` has 1 `null` and 4999 `non-null`.

Secondly, in addition to the 29 `float64` numbers we also have 16 `object` data types in both the marker feature such as `TUXT` and responses `RESP__1`. Where `object` is a catch-all data type (`dtype`) which occurs when 'multiple dtypes in a single column, the dtype of the column will be chosen to accommodate all of the data types (object is the most general).'[1].

[1] https://pandas.pydata.org/pandas-docs/stable/user_guide/basics.html#basics-dtypes

In [64]:
```
for col in df.columns:
    if df[col].dtype == 'object':
        print(df[col].apply(type).value_counts())
```

```
WIBD
<class 'str'>      4999
<class 'float'>       1
Name: count, dtype: int64
KVJI
<class 'str'>    5000
Name: count, dtype: int64
CSTT
<class 'str'>    5000
Name: count, dtype: int64
OFSQ
<class 'str'>      4998
<class 'float'>       2
Name: count, dtype: int64
JIEK
<class 'str'>    5000
Name: count, dtype: int64
RLCO
<class 'str'>    5000
Name: count, dtype: int64
RNOZ
<class 'str'>    5000
Name: count, dtype: int64
PSSJ
<class 'str'>    5000
Name: count, dtype: int64
WBYY
<class 'str'>      4999
<class 'float'>       1
Name: count, dtype: int64
TUXT
<class 'str'>      4999
<class 'float'>       1
Name: count, dtype: int64
ROQE
<class 'str'>      4998
<class 'float'>       2
Name: count, dtype: int64
ZMHO
<class 'str'>    5000
Name: count, dtype: int64
RTDN
<class 'str'>    5000
Name: count, dtype: int64
MJXP
<class 'str'>    5000
Name: count, dtype: int64
RESP__0
<class 'str'>      4999
<class 'float'>       1
Name: count, dtype: int64
RESP__1
<class 'str'>    5000
Name: count, dtype: int64
```

We investigate all the columns that have `dtype` `object` and gettings counts of all the types in that column. Note that although the column has to be a type that encompasses all other types, the individual values still preserve more nuanced types

First, we have those which are mixed types such as `str` and `float`. For example, `WIBD` has 4999 `str` and one `float`. The obvious hypothesis is that the `null` values are being interpreted as floats as we can see in `df.info` it has a count of 4999 non-null values.

In [65]:
```
for col in df.columns:
    if df[col].dtype == 'object':
        val_types = df[col].apply(type)
        if val_types.nunique() > 1:
            float_indices = df.index[df[col].apply(type) == float].tolist()
            null_indices = df.index[df[col].isnull()].tolist()
            floats_same_nulls = float_indices == null_indices
            print(f"{col}: {floats_same_nulls}, {float_indices} floats, {null_indices} nulls")
```

```
WIBD: True, [1032] floats, [1032] nulls
OFSQ: True, [191, 2187] floats, [191, 2187] nulls
WBYY: True, [215] floats, [215] nulls
TUXT: True, [3138] floats, [3138] nulls
ROQE: True, [1738, 2247] floats, [1738, 2247] nulls
RESP__0: True, [1644] floats, [1644] nulls
```

We confirm this hypothesis by showing that all the mixed columns have the float types in exactly the same places as where the nulls we saw in `df.info` were found i.e. it is only an `object` because we have non-handled the `null` yet. For example, `WIBD` has a single float value at index `1032` which is also its `null` index.

Second, there is the question of what to do with the strings. The reason that `pandas` sometimes casts to strings (technically `object`) is that it cannot convert to floats as they do not provide enough precision[2].

However, we cannot use strings to analyse the data so we have a choice between casting to float anyway or perhaps using `Decimal`. The advantage of the former is that `pandas` uses `numpy` under the hood which is optimised for floats including lower-memory footprint, hardward supported with vectorised operations and it allows us to use functions which work with floats such as `df.describe`. The advantages of the latter is exact precision which in some contexts might matter.

[2] https://pandas.pydata.org/pandas-docs/stable/user_guide/io.html#type-inference

In [66]:
```
def safe_float(x):
    try:
        return float(x)
    except ValueError:
        return x

for col in df.columns:
    if df[col].dtype == 'object':
        col_float = df[col].apply(lambda x: safe_float(x) if not pd.isna(x) else x)
        non_float_mask = ~col_float.apply(lambda x: isinstance(x, float) or pd.isna(x))

        if non_float_mask.any():
            print(f"\nColumn: {col}")
```

```
            non_float_vals = df[col][non_float_mask].index
            print(col_float[non_float_mask].tolist())
```

```
Column: WIBD
['<Err>']

Column: KVJI
['<Err>']

Column: CSTT
['<Err>', '<Err>']

Column: OFSQ
['<Err>']

Column: JIEK
['<Err>']

Column: RLCO
['<Err>']

Column: RNOZ
['<Err>']

Column: PSSJ
['<Err>']

Column: WBYY
['<Err>']

Column: TUXT
['<Err>']

Column: ROQE
['<Err>']

Column: ZMHO
['<Err>']

Column: RTDN
['<Err>', '<Err>']

Column: MJXP
['<Err>', '<Err>']

Column: RESP__0
['<Err>']

Column: RESP__1
['<Err>', '<Err>']
```

In order to use the data we try casting to float. Initially, we tried the simplest approach which is to do `df['col'].astype(float)` but this fails. To understand why we use a `safe_float` function which simply wraps each `float` cast in a `try-except` block. We then use this method as a `lambda` to each value in the column noting that those that fail will be left as `x`, which is important as we want to be able to see the data that couldn't be casted to understand the dataset better.

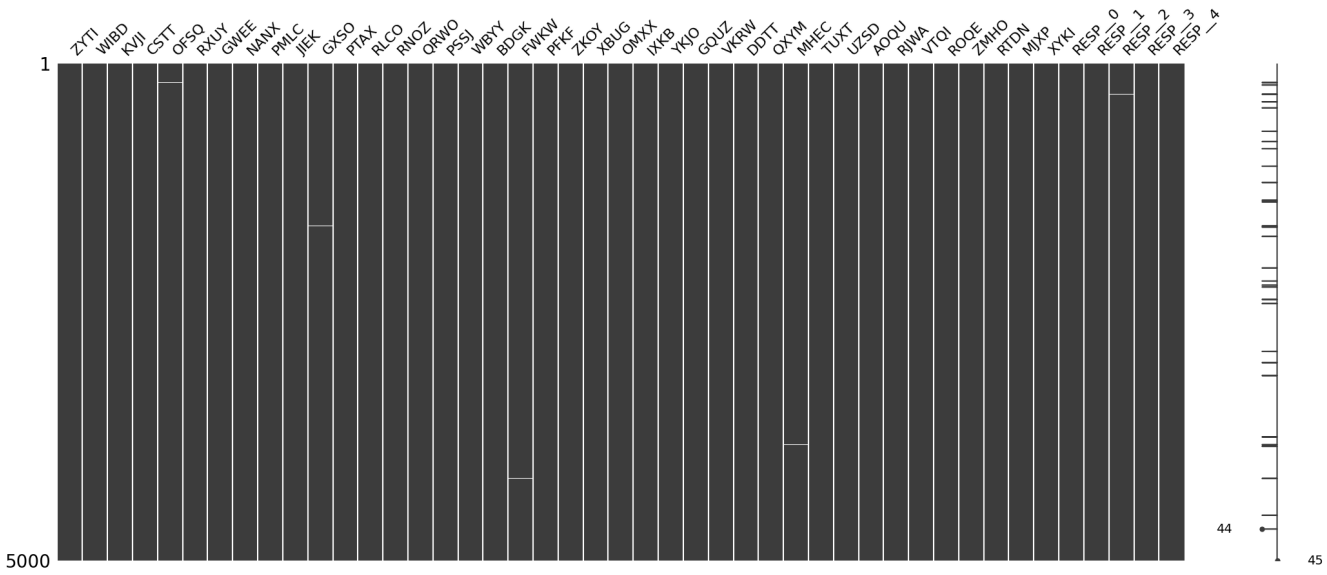
Then, we create a mask to find the indices of values in the columns that are neither `null` which we have already investigated or successfully cast to `float`, i.e. to find the failed casts. Printing them out for each column, we find that it is the same problem of `<Err>` for all, and in fact inspecting the original dataset we can see these entries.

So, in summary we have found that both `<Err>` and `null` in the data. The next choice we have is whether to treat errors and `null` the same and in each case decide whether to

1. Remove the row of data entirely
2. Impute the value

In [67]:
```
import missingno as msno
msno.matrix(df)
```

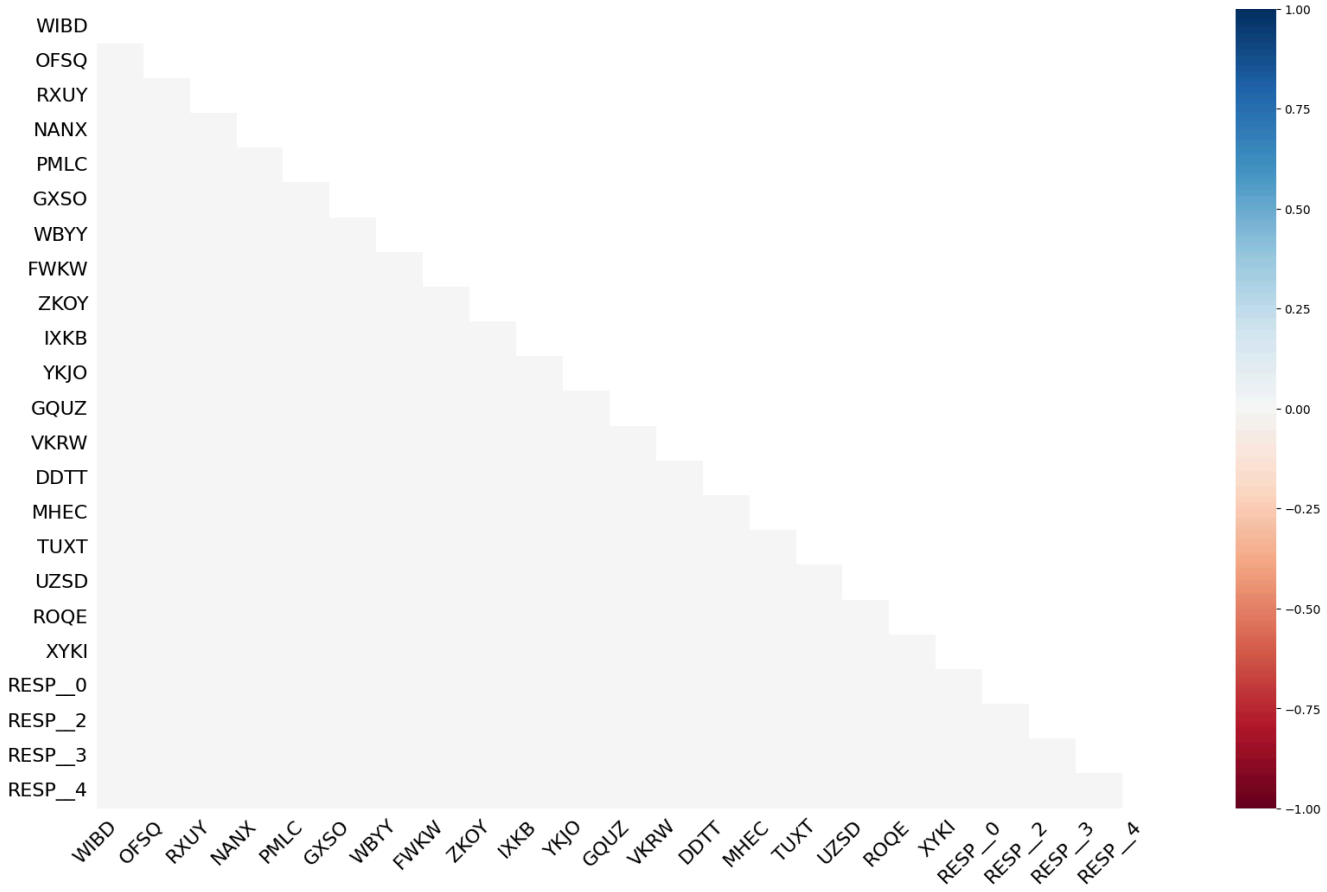
Out[67]: <Axes: >



We try using the `matrix` function from the `missingno` library but although we can see in the bar chart on the right a large number of missing values we can only see five of them in the actual chart, which is a common problem related to the amount of data compared to the number of pixels. This means we are unable to see if there are any patterns in the distribution of `null`. We tried `msno.matrix(df.sample(250))` but then only one `null` happens to be present. We could use `msno.bar(df)` but this would only repeat the `null` count information we have already from `df.info()`

In [68]:
```
msno.heatmap(df)
```

Out[68]: <Axes: >



We try to see if there is any correlation in the missing numbers but do not find anything.

In [69]:
```
df_no_nan = df.dropna()
print(df_no_nan.shape)
```

```
(4970, 45)
```

One of the approaches we could take is bulk deletion of any rows with missing values. We can see from the above that we lose 30 rows which as a proportion of 5000 rows of data is just 0.6% of the data.

In [70]:
```
df_err_mask = (df == '<Err>').any(axis=1)
df_no_err = df.drop(df[df_err_mask].index)
print(df_no_err.shape)
```

```
(4980, 45)
```

Using a similar process we drop any rows with `<Err>` values and see we lose 20 rows.

In [71]:
```
df_no_err_nan = df_no_err.dropna()
print(df_no_err_nan.shape)
```

```
(4950, 45)
```

If we drop both the errors and the `null` we find that we lose 50 rows of data i.e. there was no overlap in the errors and the `null`. Losing data is never ideal - even if it is just 1% of the data.

Here we need to make a judgement call. We have been given limited information about the dataset where we only know that we have 40 genetic markers and 5 response variables but we do not know specifically what they represent and nor do we have any working hypotheses about any relationships in the data. Furthermore, we do not know much about how the data was generated or for example why some data is missing and why some data was `<Err>`. If the errors come from random data collection errors then it might be appropriate to treat them as just more missing data, but given we are not told it could be that all the data for a given sample is collected at the same time and so an error in one value actually means all that the other values are also compromised. This decision is further justified by the fact that the errors make up a relatively small proportion of the data we decide to completely drop the error rows.

In [72]:
```
import numpy as np
df_resp_na_mask = df_no_err.iloc[:, -5:].isna().any(axis=1)
print(df_resp_na_mask.sum())
df_no_err_resp = df_no_err.drop(df_no_err[df_resp_na_mask].index)
print(df_no_err_resp.shape)
```

```
4
(4976, 45)
```

In an effort to retain as much data as possible we treat missing values in the response variables and the explanatory variables differently.

For the response variables we delete them (known as listwise complete case deletion) because imputing them can lead to signficant bias[3]. In doing this we can see we have dropped 4 rows, an acceptably small amount of data loss from our 5000 sample.

[3] https://en.wikipedia.org/wiki/Imputation_(statistics)

In [73]:
```
from decimal import Decimal

def safe_decimal(x):
    try:
        return Decimal(x)
    except (ValueError, TypeError):
        return x

for col in df_no_err_resp.columns:
    if df_no_err_resp[col].dtype == 'object':
```

```
        # cast strings to float and decimal
        col_float = df_no_err_resp[col].dropna().apply(safe_float)
        col_decimal = df_no_err_resp[col].dropna().apply(safe_decimal)

        # cast float to decimal
        col_float = col_float.apply(Decimal)

        float_decimal_diff = col_float != col_decimal
        if float_decimal_diff.any():
            diff_indices = float_decimal_diff[float_decimal_diff].index.tolist()
            float_diffs = col_float.loc[diff_indices]
            decimal_diffs = col_decimal.loc[diff_indices]
            print(f"Float diffs in {col}")
            max_diff = Decimal(0)
            max_diff_index = 0
            for i in range(len(float_diffs)):
                diff = abs(float_diffs.iloc[i] - decimal_diffs.iloc[i])
                if diff > max_diff:
                    max_diff = diff
                    max_diff_index = i

            print(f"Max diff: {max_diff} where {float_diffs.iloc[max_diff_index]} != {decimal_diffs.iloc[max_diff_index]}")
```

```
Float diffs in WIBD
Max diff: 0.0000046826171875 where 109156811357.5259246826171875 != 109156811357.52592
Float diffs in KVJI
Max diff: 0.0000046875 where -118098088887.4189453125 != -118098088887.41895
Float diffs in CSTT
Max diff: 0.00001455078125 where -162216975264.58758544921875 != -162216975264.5876
Float diffs in OFSQ
Max diff: 0.0000021875 where 142949236984.4892578125 != 142949236984.48926
Float diffs in JIEK
Max diff: 0.000002763671875 where 178946755275.296417236328125 != 178946755275.29642
Float diffs in RLCO
Max diff: 0.000001435546875 where 161853626366.520721435546875 != 161853626366.52072
Float diffs in RNOZ
Max diff: 0.000002275390625 where 126073385674.965667724609375 != 126073385674.96567
Float diffs in PSSJ
Max diff: 0.0000142578125 where -197017749143.1671142578125 != -197017749143.1671
Float diffs in WBYY
Max diff: 0.000004326171875 where 185202097981.529815673828125 != 185202097981.52982
Float diffs in TUXT
Max diff: 0.00001240234375 where 152953352189.59478759765625 != 152953352189.5948
Float diffs in ROQE
Max diff: 0.00000478515625 where -194479254725.15240478515625 != -194479254725.1524
Float diffs in ZMHO
Max diff: 0.0000046044921875 where 100697248396.2063446044921875 != 100697248396.20634
Float diffs in RTDN
Max diff: 8.7890625E-8 where -144693497917.132659912109375 != -144693497917.13266
Float diffs in MJXP
Max diff: 9.765625E-8 where -162831873525.41510009765625 != -162831873525.4151
Float diffs in RESP__0
Max diff: 0.00000380859375 where 36856196793.27984619140625 != 36856196793.27985
Float diffs in RESP__1
Max diff: 0.0000134765625 where 192457552679.2161865234375 != 192457552679.2162
```

In [74]:
```
Decimal(0.0000046826171875)/Decimal(109156811357.5259246826171875) * 100
```

Out[74]:
```
Decimal('4.289807598137715388151535604E-15')
```

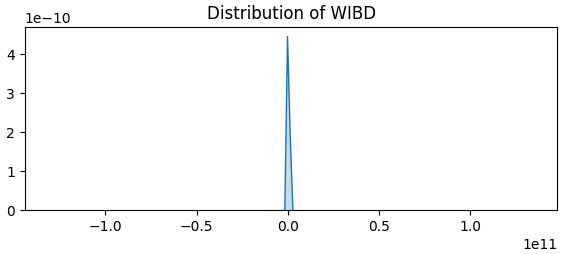
To help us decide whether to use floats or decimals we cast the strings to `float` and `Decimal` and then cast the `float` to `Decimal` to allow us to compare them. We then for each column find the absolute maximum difference and print out the `float` and `Decimal` values to look at. Although the differences are found in 16 of the columns the loss of information seems very minimal compared to the size of the values. For example, the `WIBD` column has a difference of `0.0000046826171875` compared to the actual value of `109156811357.5259246826171875` is a tiny number with 14 zeroes of a percent. We therefore conclude that if is safe not to use `Decimal` and instead use `float`.

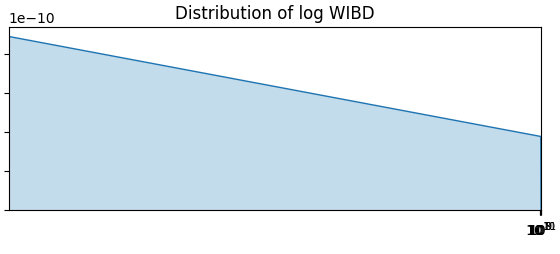
In [75]:
```
df_float = df_no_err_resp.copy()
for col in df_float.columns:
    if df_float[col].dtype == 'object':
        df_float[col] = df_float[col].apply(lambda x: safe_float(x) if not pd.isna(x) else x)

df_float.info()
```

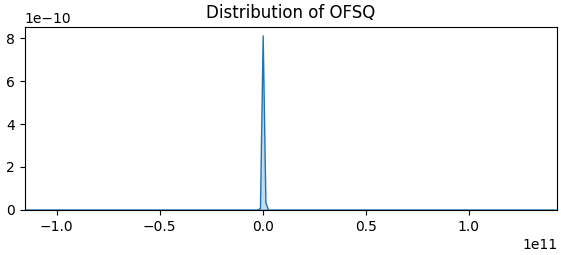
```
<class 'pandas.core.frame.DataFrame'>
Index: 4976 entries, 0 to 4999
Data columns (total 45 columns):
 #   Column   Non-Null Count  Dtype  
---  ------   --------------  -----  
 0   ZYTI     4976 non-null   float64
 1   WIBD     4975 non-null   float64
 2   KVJI     4976 non-null   float64
 3   CSTT     4976 non-null   float64
 4   OFSQ     4974 non-null   float64
 5   RXUY     4975 non-null   float64
 6   GWEE     4976 non-null   float64
 7   NANX     4975 non-null   float64
 8   PMLC     4974 non-null   float64
 9   JIEK     4976 non-null   float64
 10  GXSO     4975 non-null   float64
 11  PTAX     4976 non-null   float64
 12  RLCO     4976 non-null   float64
 13  RNOZ     4976 non-null   float64
 14  QRWO     4976 non-null   float64
 15  PSSJ     4976 non-null   float64
 16  WBYY     4975 non-null   float64
 17  BDGK     4976 non-null   float64
 18  FWKW     4975 non-null   float64
 19  PFKF     4976 non-null   float64
 20  ZKOY     4974 non-null   float64
 21  XBUG     4976 non-null   float64
 22  OMXX     4976 non-null   float64
 23  IXKB     4975 non-null   float64
 24  YKJO     4975 non-null   float64
 25  GQUZ     4975 non-null   float64
 26  VKRW     4975 non-null   float64
 27  DDTT     4975 non-null   float64
 28  QXYM     4976 non-null   float64
 29  MHEC     4973 non-null   float64
 30  TUXT     4975 non-null   float64
 31  UZSD     4975 non-null   float64
 32  AOQU     4976 non-null   float64
 33  RJWA     4976 non-null   float64
 34  VTQI     4976 non-null   float64
 35  ROQE     4974 non-null   float64
 36  ZMHO     4976 non-null   float64
 37  RTDN     4976 non-null   float64
 38  MJXP     4976 non-null   float64
 39  XYKI     4974 non-null   float64
 40  RESP__0  4976 non-null   float64
 41  RESP__1  4976 non-null   float64
 42  RESP__2  4976 non-null   float64
 43  RESP__3  4976 non-null   float64
 44  RESP__4  4976 non-null   float64
dtypes: float64(45)
memory usage: 1.7 MB
```

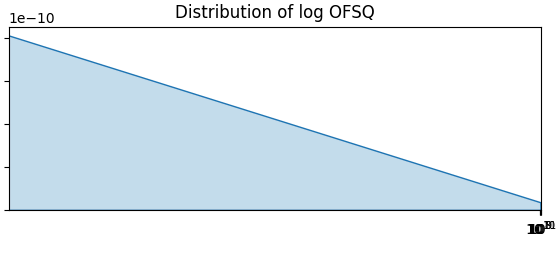
Now we have removed the errors we can retry casting to float again which we see is successful where all columns are now floats. It is worth noting that most columns have 4976 non-null values but not all as we still have the missing values to deal with.

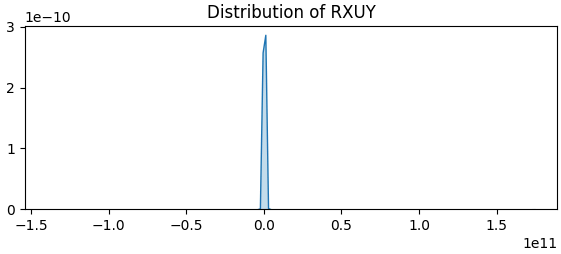
In [76]:
```python
import seaborn as sns
import matplotlib.pyplot as plt

cols_to_plot = df_float.columns[df_float.isna().any()].to_list()
fig, axes = plt.subplots(nrows=len(cols_to_plot), ncols=2, figsize=(12, 50))
for i, col in enumerate(cols_to_plot):
    data = df_float[col].dropna()

    # kde plot
    sns.kdeplot(data, ax=axes[i][0], fill=True)
    axes[i][0].set_title(f'Distribution of {col}')
    axes[i][0].set_xlabel('')
    axes[i][0].set_ylabel('')
    axes[i][0].set_xlim(data.min(), data.max())

    # log plot
    sns.kdeplot(data, ax=axes[i][1], fill=True)
    axes[i][1].set_title(f'Distribution of log {col}')
    axes[i][1].set_xlabel('')
    axes[i][1].set_ylabel('')
    axes[i][1].set_xlim(data.min(), data.max())
    axes[i][1].set_xscale('log')

plt.tight_layout()
plt.show()
```

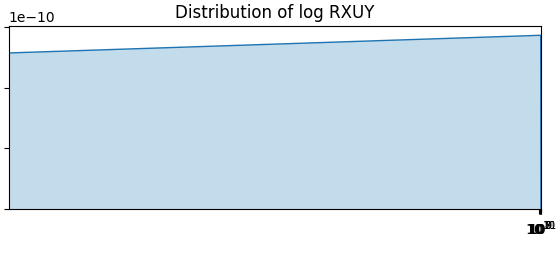

Distribution of WIBD
1e−10
1e11

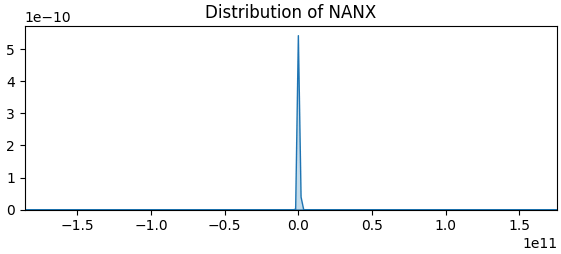

Distribution of log WIBD
1e−10

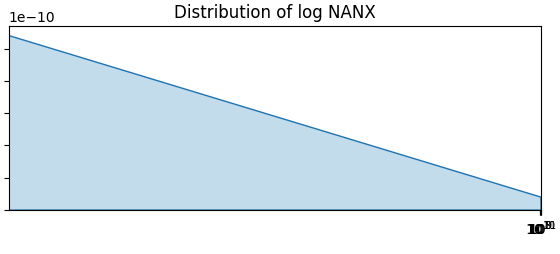

Distribution of OFSQ
1e−10
1e11

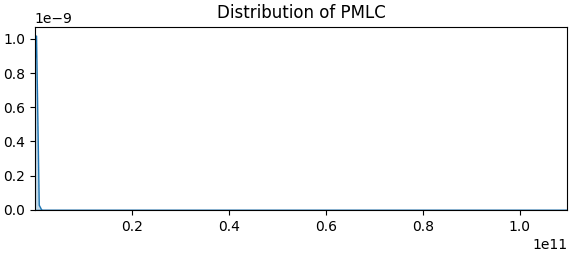

Distribution of log OFSQ
1e−10

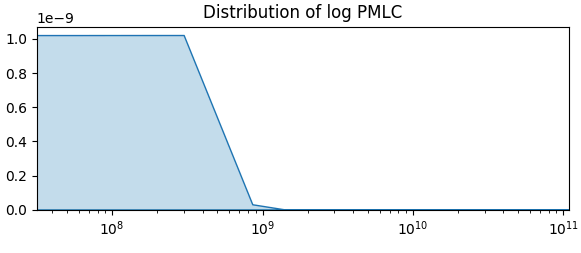

Distribution of RXUY
1e−10
1e11

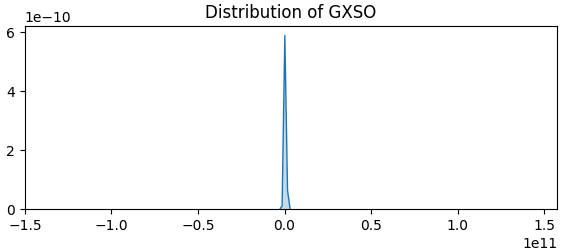

Distribution of log RXUY
1e−10

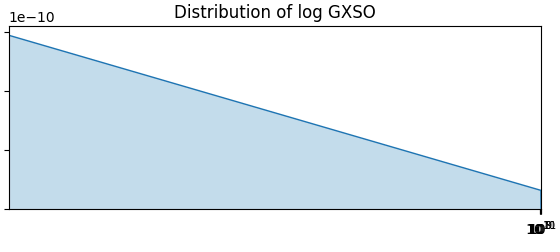

Distribution of NANX
1e−10
1e11

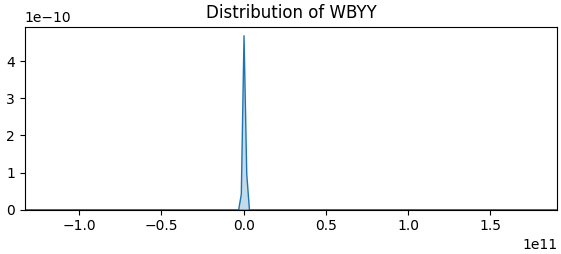

Distribution of log NANX
1e−10

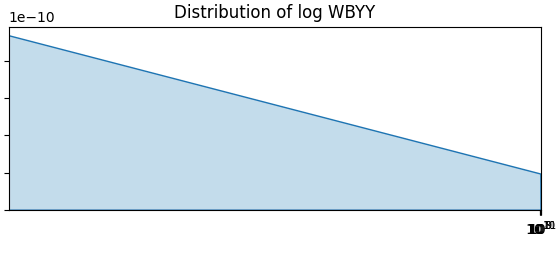

Distribution of PMLC
1e−9
1e11

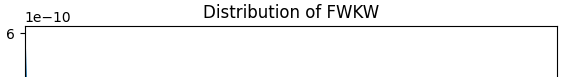

Distribution of log PMLC
1e−9

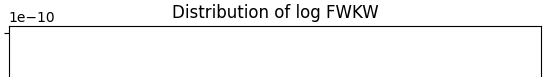

Distribution of GXSO
1e−10
1e11

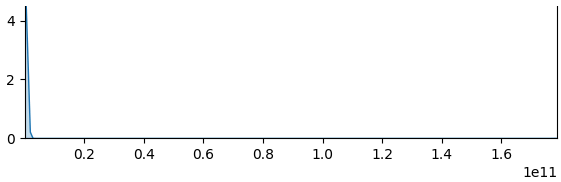

Distribution of log GXSO
1e−10

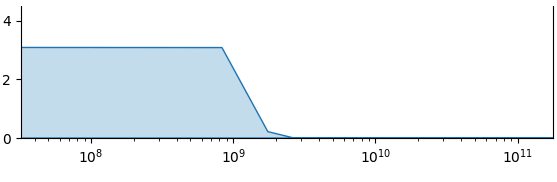

Distribution of WBYY
1e−10
1e11

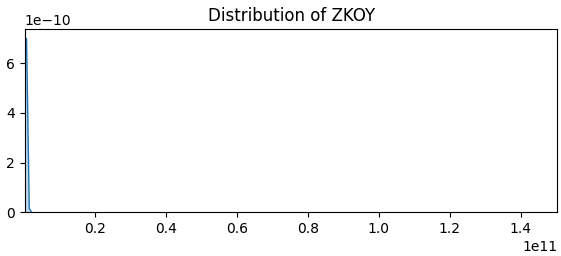

Distribution of log WBYY
1e−10

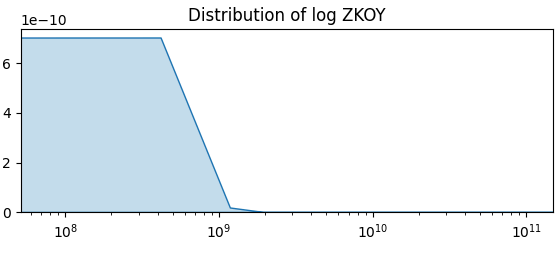

Distribution of FWKW
1e−10

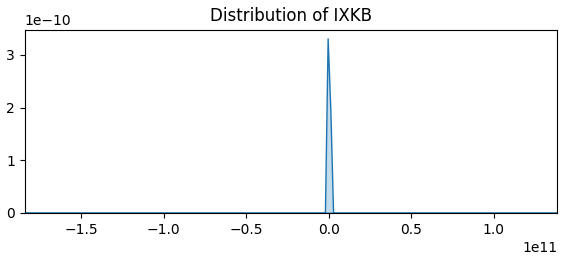

Distribution of log FWKW
1e−10

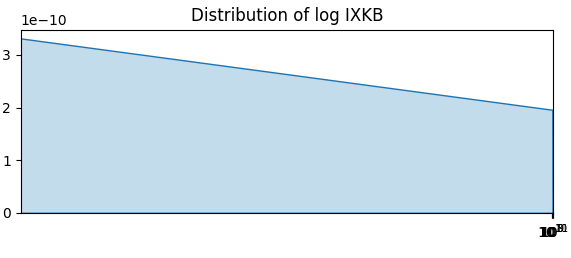


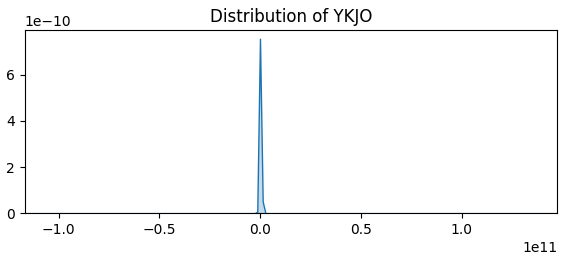


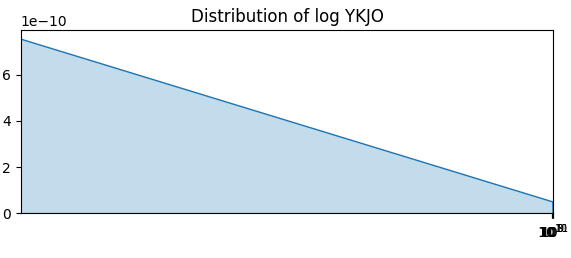

Distribution of ZKOY
1e−10
1e11

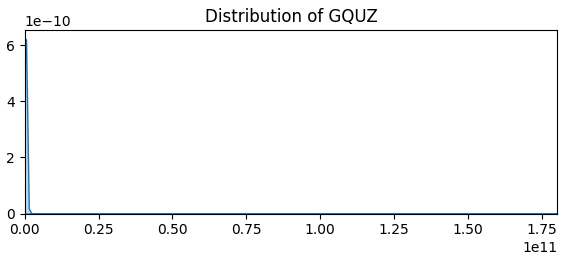

Distribution of log ZKOY
1e−10

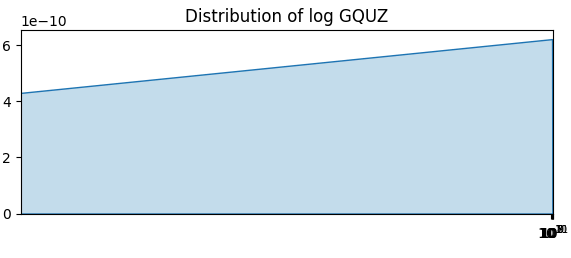

Distribution of IXKB
1e−10
1e11

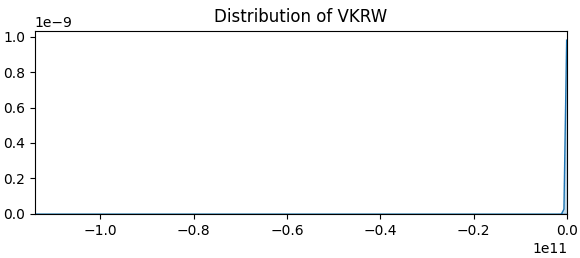

Distribution of log IXKB
1e−10

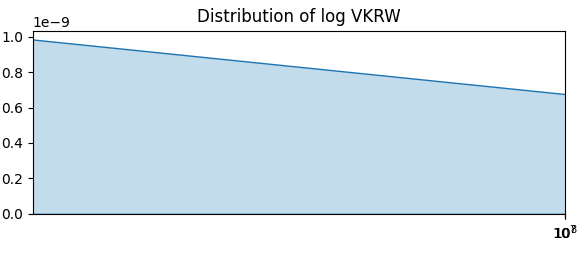

Distribution of YKJO
1e−10
1e11

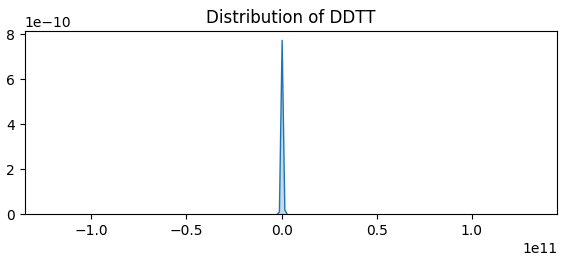

Distribution of log YKJO
1e−10

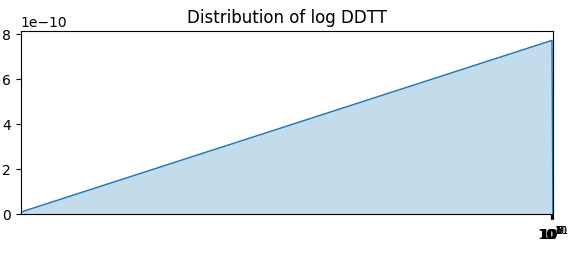

Distribution of GQUZ
1e−10
1e11

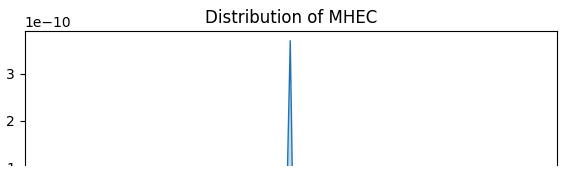

Distribution of log GQUZ
1e−10

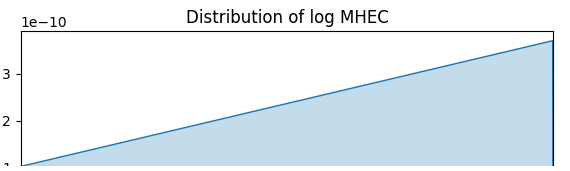

Distribution of VKRW
1e−9
1e11

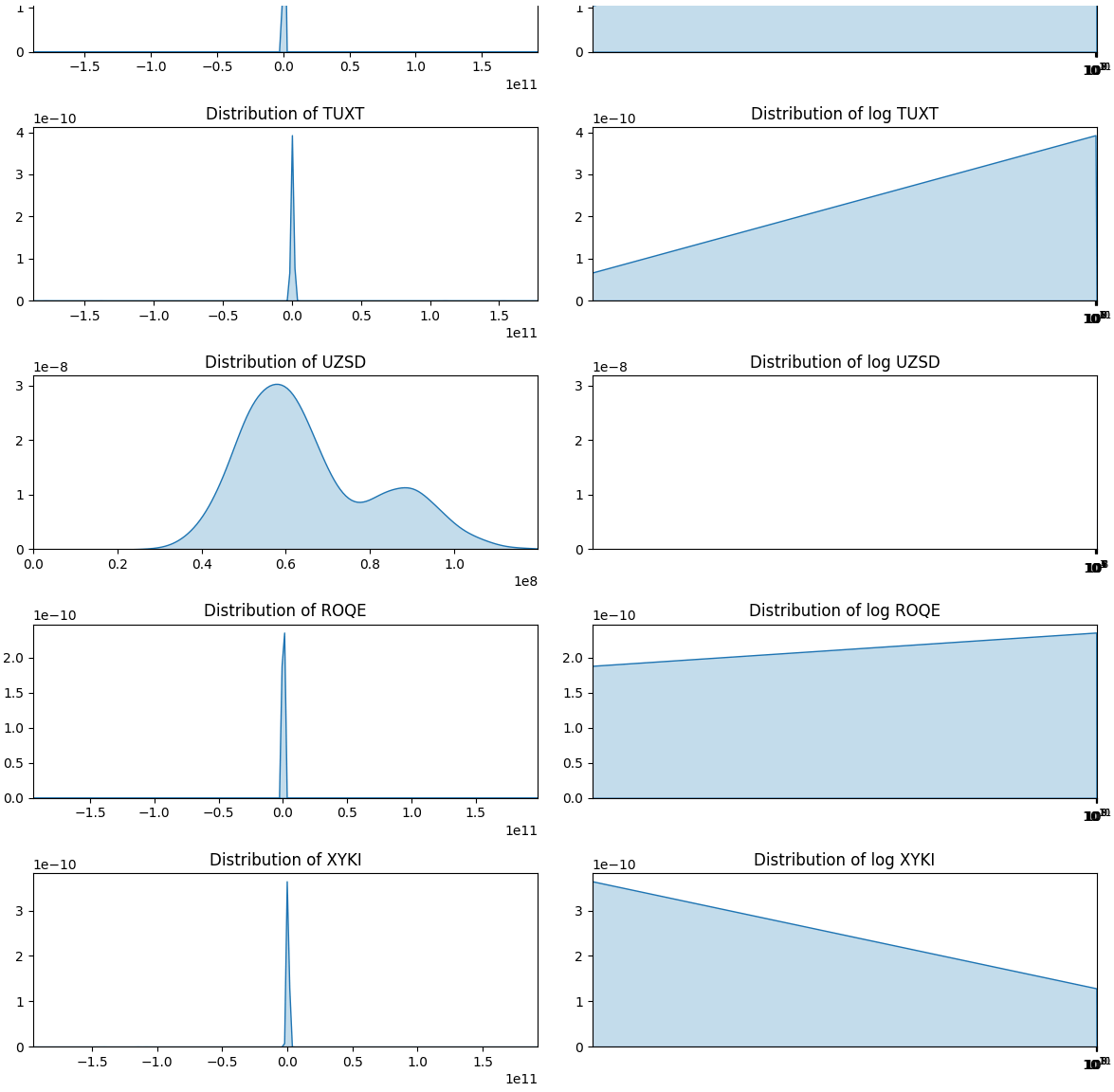

Distribution of log VKRW
1e−9


Distribution of DDTT
1e−10
1e11


Distribution of log DDTT
1e−10


Distribution of MHEC
1e−10


Distribution of log MHEC
1e−10



For the explanatory variables we impute the missing values. Deletion could introduce bias if the missing data is not Missing Completely At Random (MCAR)[4] although admittedly visually inspecting the heatmap it seems this was not an issue.

For imputation we have a choice between imputing with[4]:

- Mean: Advantage is it preserves the mean of the variable but is sensitive to outliers which can skew the statistic.
- Median: It is less sensitive to outliers so appropriate for skewed distributions.
- Mode: Best for categorical data.

To choose between them we get `cols_to_plot` which are all the columns with null explanatory variables that we want to impute. It turns out we have 19 of these and we plot each of them with two graphs. The first is a Kernal Density Estimation (KDE) plot which allows us to see the distribution of the data. We could have chosen to use a histogram which is a simpler bar chart type plot but this has the disadvantage of being quite reliant on choosing the right bin width[5]. However, even though we have scaled the x-axes of each graph individually to the min and max it is still quite hard to see the distribution of the data.

Thus, we try transforming each kde in the panel on the right by using `.set_xscale('log')`. This makes very skewed data easier to see and if the underlying data generating function is logarithmic it makes it look linear. This can be proven mathematically as follows. Given an underlying $y = log(x)$ function if we set the scale to `log` our function becomes

$$y = log(10^x)$$

where we assume that the base is 10. Then using the logarithm property $log(a^b) = blog(a)$ this simplifies to

$$y = xlog(10)$$

but this is just a linear equation where `log(10)` is a constant[6]. Note that even if the logarithms don't match you will still end up with a linear equation but just with different slopes. This explains why the kdes which are smooth curves over the underlying data look linear in many of the graphs above such as `WIBD` and `OBSQ`. If one of the graphs had looked normal then that would be justfication to use the `mean` for the imputed value rather than the `median`, but even `UZSD` is bimodal. We also see a piece-wise logarihmic functions such as with `FWKW` and `ZKOY`, but the story is still the same where we will use the median in all cases as it is more robust to the outliers found in skewed distributions.

[4] https://en.wikipedia.org/wiki/Missing_data

[5] https://en.wikipedia.org/wiki/Kernel_density_estimation

[6] https://en.wikipedia.org/wiki/List_of_logarithmic_identities

In [77]:
```
df_impute = df_float.copy()
for col in df_impute.columns:
    if df_impute[col].dtype == 'float64':
        median_val = df_impute[col].median()
        df_impute[col].fillna(median_val, inplace=True)
```

```
df_impute.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 4976 entries, 0 to 4999
Data columns (total 45 columns):
 #   Column   Non-Null Count  Dtype  
---  ------   --------------  -----  
 0   ZYTI     4976 non-null   float64
 1   WIBD     4976 non-null   float64
 2   KVJI     4976 non-null   float64
 3   CSTT     4976 non-null   float64
 4   OFSQ     4976 non-null   float64
 5   RXUY     4976 non-null   float64
 6   GWEE     4976 non-null   float64
 7   NANX     4976 non-null   float64
 8   PMLC     4976 non-null   float64
 9   JIEK     4976 non-null   float64
 10  GXSO     4976 non-null   float64
 11  PTAX     4976 non-null   float64
 12  RLCO     4976 non-null   float64
 13  RNOZ     4976 non-null   float64
 14  QRWO     4976 non-null   float64
 15  PSSJ     4976 non-null   float64
 16  WBYY     4976 non-null   float64
 17  BDGK     4976 non-null   float64
 18  FWKW     4976 non-null   float64
 19  PFKF     4976 non-null   float64
 20  ZKOY     4976 non-null   float64
 21  XBUG     4976 non-null   float64
 22  OMXX     4976 non-null   float64
 23  IXKB     4976 non-null   float64
 24  YKJO     4976 non-null   float64
 25  GQUZ     4976 non-null   float64
 26  VKRW     4976 non-null   float64
 27  DDTT     4976 non-null   float64
 28  QXYM     4976 non-null   float64
 29  MHEC     4976 non-null   float64
 30  TUXT     4976 non-null   float64
 31  UZSD     4976 non-null   float64
 32  AOQU     4976 non-null   float64
 33  RJWA     4976 non-null   float64
 34  VTQI     4976 non-null   float64
 35  ROQE     4976 non-null   float64
 36  ZMHO     4976 non-null   float64
 37  RTDN     4976 non-null   float64
 38  MJXP     4976 non-null   float64
 39  XYKI     4976 non-null   float64
 40  RESP__0  4976 non-null   float64
 41  RESP__1  4976 non-null   float64
 42  RESP__2  4976 non-null   float64
 43  RESP__3  4976 non-null   float64
 44  RESP__4  4976 non-null   float64
dtypes: float64(45)
memory usage: 1.7 MB
```

We have now imputed all the nulls with the median value for that column of data giving us 4976 non-null data points for every column.

In imputing our missing values we only used that column of data i.e. the median of all the other values which is known as univariate imputation.

Multivariate imputation in contrast treats each column as the dependent variable and uses all the other values to estimate it. This can, for example, be done in scikit-learn using an `IterativeImputer` which models features in a round-robin approach[7]

However, this approach is dangerous particularly as we know so little about the data we are analysing, therefore we opt for the safest approach of median univariate imputation.

[7] https://scikit-learn.org/stable/modules/impute.html#multivariate-feature-imputation

1. Is the wheat in this dataset homogeneous, random or does it fall into distinct types?

In [78]:
```
df_impute.describe()
```

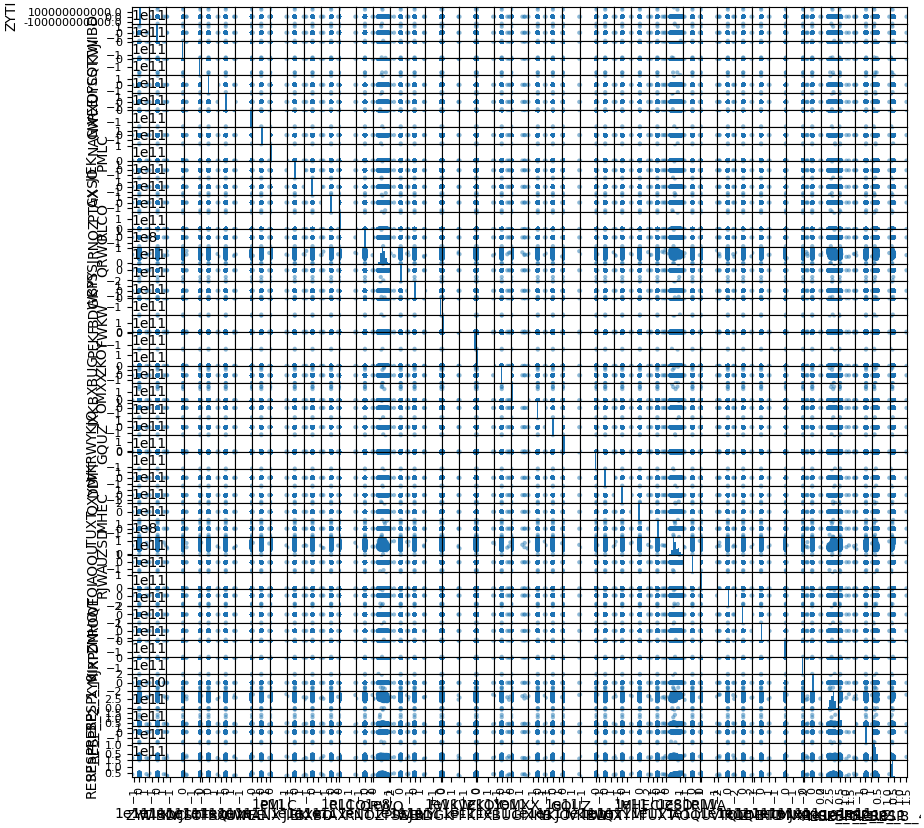
Out[78]:

	ZYTI	WIBD	KVJI	CSTT	OFSQ	RXUY	GWEE	NANX	PMLC	JIEK	GXSO	PTAX	RLCO	RNOZ	QRWO	PSSJ	WBYY	BDGK
count	4.98e+03	4.98e+03	4.98e+03	4.98e+03	4.98e+03	4.98e+03	4.98e+03	4.98e+03	4.98e+03	4.98e+03	4.98e+03	4.98e+03	4.98e+03	4.98e+03	4.98e+03	4.98e+03	4.98e+03	4.98e+03
mean	1.10e+08	9.11e+07	3.48e+07	-3.23e+07	7.43e+07	1.14e+08	4.79e+07	7.52e+07	9.59e+07	8.62e+07	5.22e+07	3.91e+07	1.02e+08	6.73e+07	6.10e+07	3.04e+07	9.50e+07	4.37e+07
std	3.71e+09	3.31e+09	2.40e+09	4.24e+09	2.61e+09	4.25e+09	1.91e+09	3.63e+09	1.56e+09	3.31e+09	3.51e+09	3.87e+09	2.29e+09	2.72e+09	1.18e+07	3.78e+09	4.55e+09	2.68e+09
min	-1.34e+11	-1.44e+11	-1.21e+11	-1.92e+11	-1.15e+11	-1.54e+11	-1.35e+11	-1.86e+11	3.15e+07	-1.50e+11	-1.50e+11	-1.82e+11	3.02e+07	-1.45e+11	0.00e+00	-1.97e+11	-1.33e+11	-1.89e+11
25%	6.25e+07	5.50e+07	7.21e+07	5.88e+07	6.05e+07	5.83e+07	5.86e+07	6.45e+07	5.69e+07	6.83e+07	5.67e+07	6.28e+07	5.63e+07	5.66e+07	5.25e+07	5.78e+07	5.93e+07	6.37e+07
50%	7.17e+07	6.52e+07	8.50e+07	6.73e+07	6.95e+07	6.75e+07	7.67e+07	7.72e+07	7.32e+07	8.32e+07	7.04e+07	8.11e+07	6.51e+07	7.18e+07	6.02e+07	7.13e+07	6.79e+07	8.76e+07
75%	8.29e+07	8.13e+07	9.55e+07	8.57e+07	7.76e+07	8.65e+07	8.97e+07	8.91e+07	9.09e+07	9.26e+07	9.38e+07	9.29e+07	8.18e+07	8.55e+07	6.94e+07	8.51e+07	8.09e+07	9.70e+07
max	1.79e+11	1.48e+11	1.22e+08	1.19e+08	1.43e+11	1.89e+11	1.20e+08	1.76e+11	1.10e+11	1.79e+11	1.58e+11	1.36e+11	1.62e+11	1.26e+11	1.06e+08	1.23e+11	1.91e+11	1.25e+08

Now we have cleaned the data we can start trying to understand the relationships within it. First, we can use `.describe()` where now that we have converted all data points to float we can see all the columns. What we notice is that many columns have a large range in values from very negative numbers to very large numbers, for example `ZYTI` `min` is in the `negative billions` and `max` in the positive `billions`. Others like `QRWO` have a `min` of 0 and a `max` in the millions. Although this tabular information is rich, it can be hard to process however, so instead we shall use packages to help us visualize the data instead.

In [79]:
```
pd.plotting.scatter_matrix(df_impute, figsize=(10, 10), diagonal='hist')
plt.tight_layout
plt.show()
```

```
c:\Python311\Lib\site-packages\pandas\plotting\_matplotlib\misc.py:122: RuntimeWarning: invalid value encountered in cast
  if np.all(locs == locs.astype(int)):
```



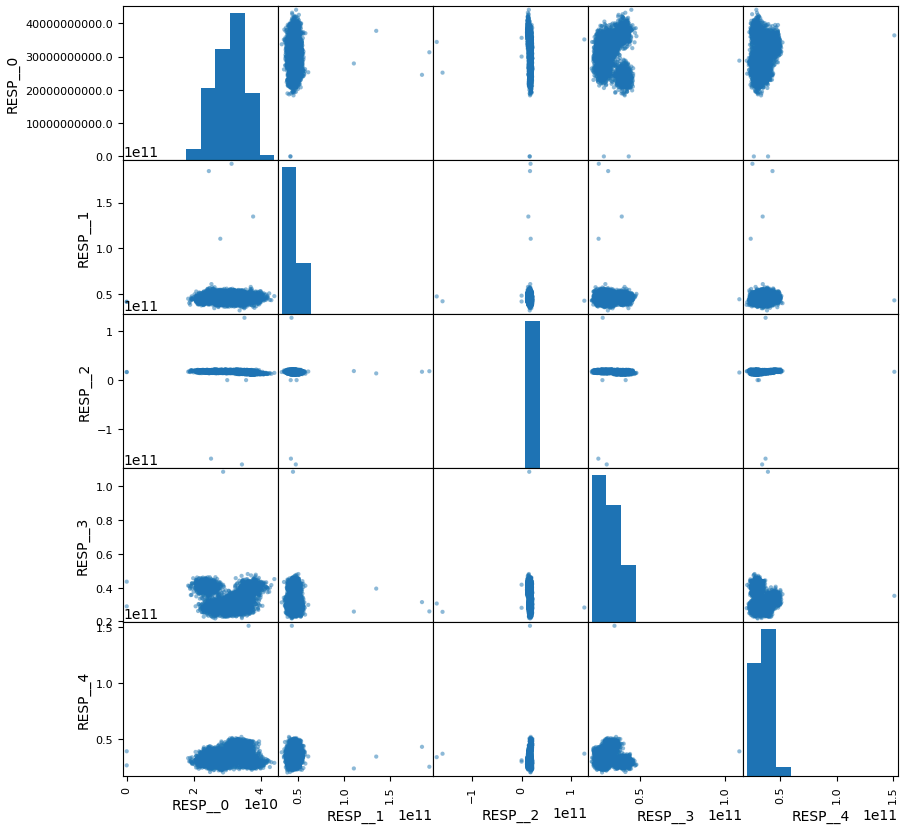
For an overview of data distribution and potential relationships between features we can use `scatter_matrix` which creates a graph for every pairwise combination of columns. Unfortunately, this is completely unreadable.

We have also triggered a warning which is because pandas is trying to cast our floats to integers `c:\Python311\Lib\site-packages\pandas\plotting\_matplotlib\misc.py:122: RuntimeWarning: invalid value encountered in cast if np.all(locs == locs.astype(int))`. The problem is that we ard generating very large and very small numbers which the underlying library cannot handle - later on we shall explore potentially using standardisation, where features are scaled to have similar ranges[8]

[7] https://www.geeksforgeeks.org/how-to-fix-runtimewarning-invalid-value-encountered-in-double_scalars/

In [80]:
```
responses = ['RESP__0', 'RESP__1', 'RESP__2', 'RESP__3', 'RESP__4']
pd.plotting.scatter_matrix(df_impute[responses], figsize=(10, 10), diagonal='hist')
plt.tight_layout
plt.show()
```

```
c:\Python311\Lib\site-packages\pandas\plotting\_matplotlib\misc.py:122: RuntimeWarning: invalid value encountered in cast
  if np.all(locs == locs.astype(int)):
```



If the responses were uniformly distributed data we would expect the diagonal histogram plots to be a flat, rectangular blocks where each column (with some random variation) is approximately the same height. The off-diagonal scatter plots correspondingly should be evenly spread out with no obvious clusters or empty spaces.

If the responses were randomly distributed following a normal (or perhaps near-normal such as t-distribution) we would expect the histogram plots to have the classic bell-shaped curve with most of the values in the middle. Whilst for the off-diagonal plots we would expect a circular cloud of points, as if we are looking down on the hill of points with most of them in the centre.

If the data was homogeneous then the data comes from a single population or process in contrast to heterogeneity. Unlike uniform or random distributions there might still be clusters but these would not be distinct, suggesting different subgroups.

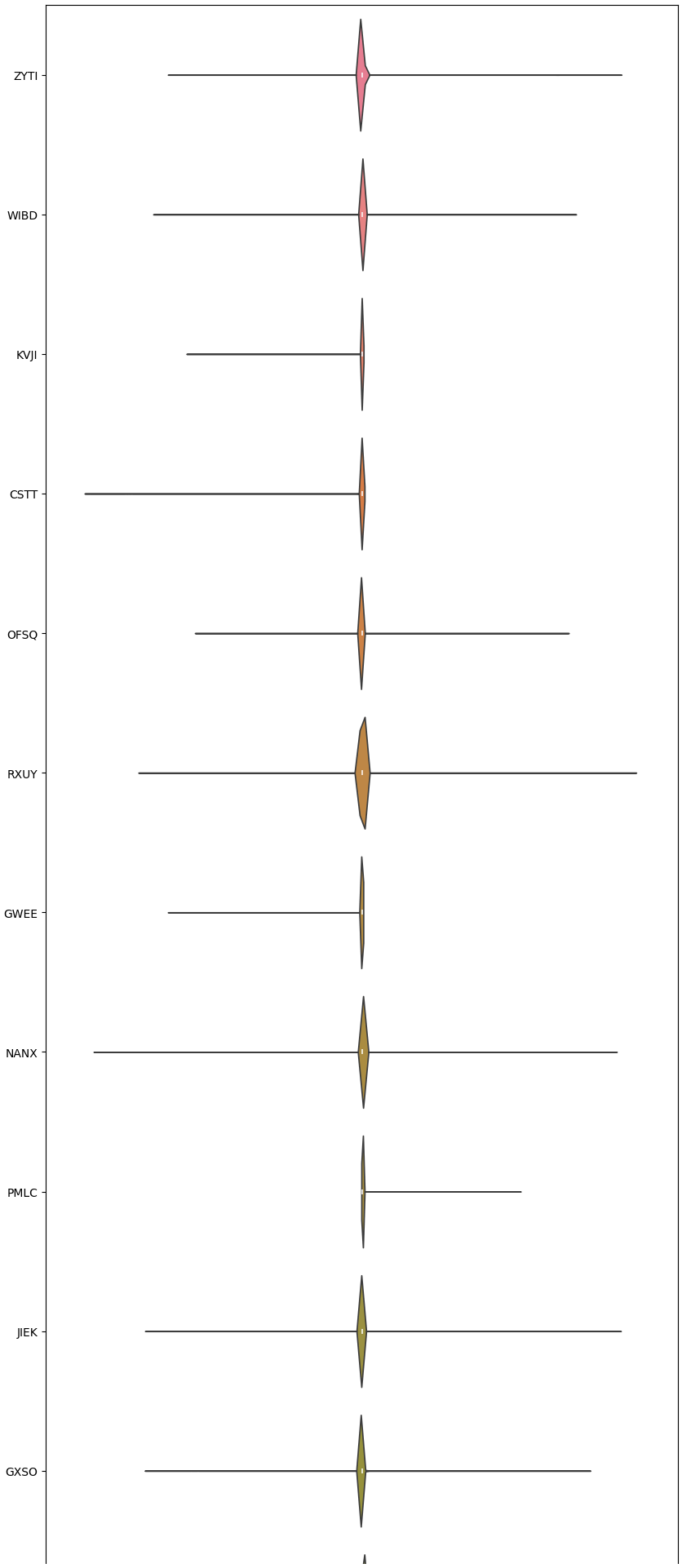
When we look at the response variables we find that the histograms are highly skewed, particularly `RESP__1`, `RESP__3`, and `RESP__4`, to the right (i.e. positive skew) meaning the majority of the data is concentrated on the left side. `RESP__0` is the most normal looking, whilst `RESP__2` is highly concentrated.

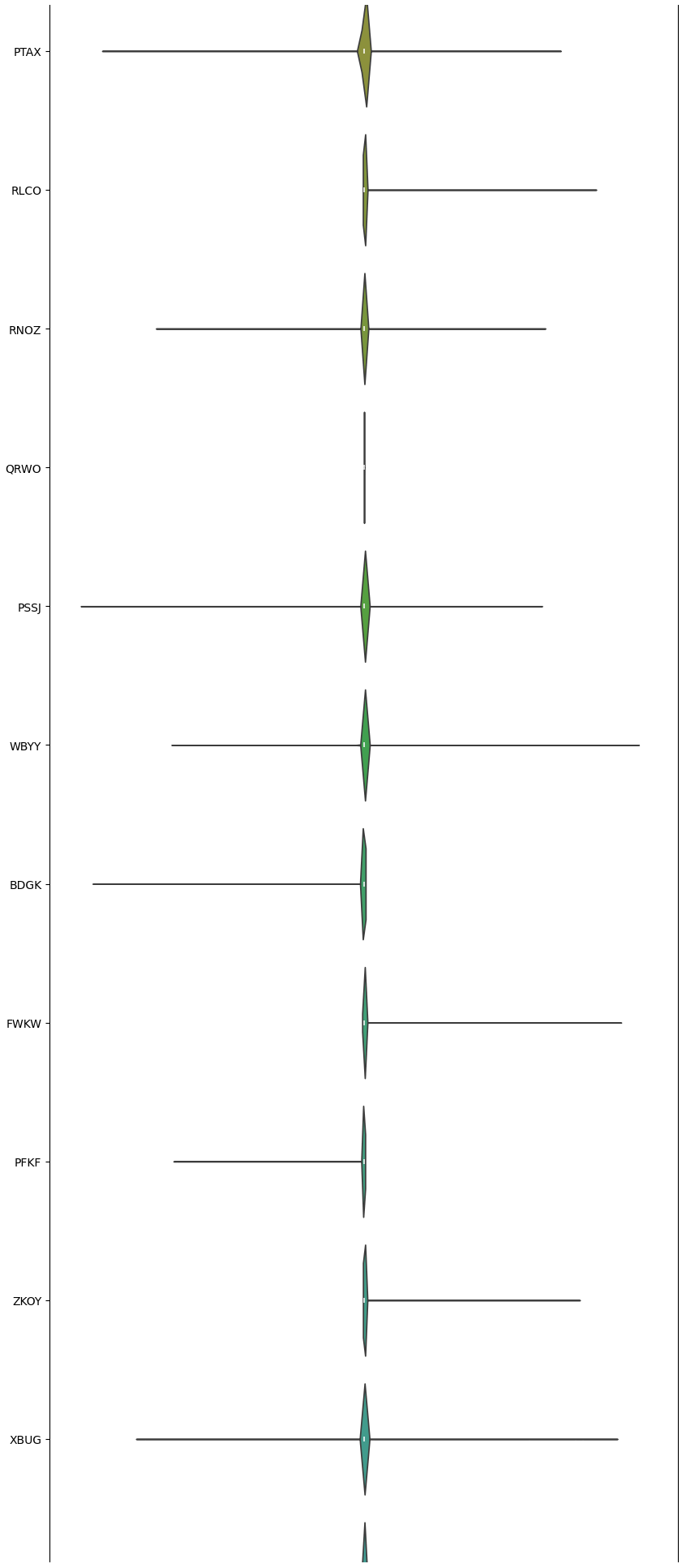
In all cases there are a small number of quite extreme outliers. These are not visible on the histogram plots but can be ascertained by the fact that the scales are mnch wider than the bulk of the values. For example, most `RESP__1` values are around the mean of `4.57e+10` but we can also see that the scale goes up to the maximum of `1.92e+11` quite far to the right. We shall explore the outliers further in later sections.

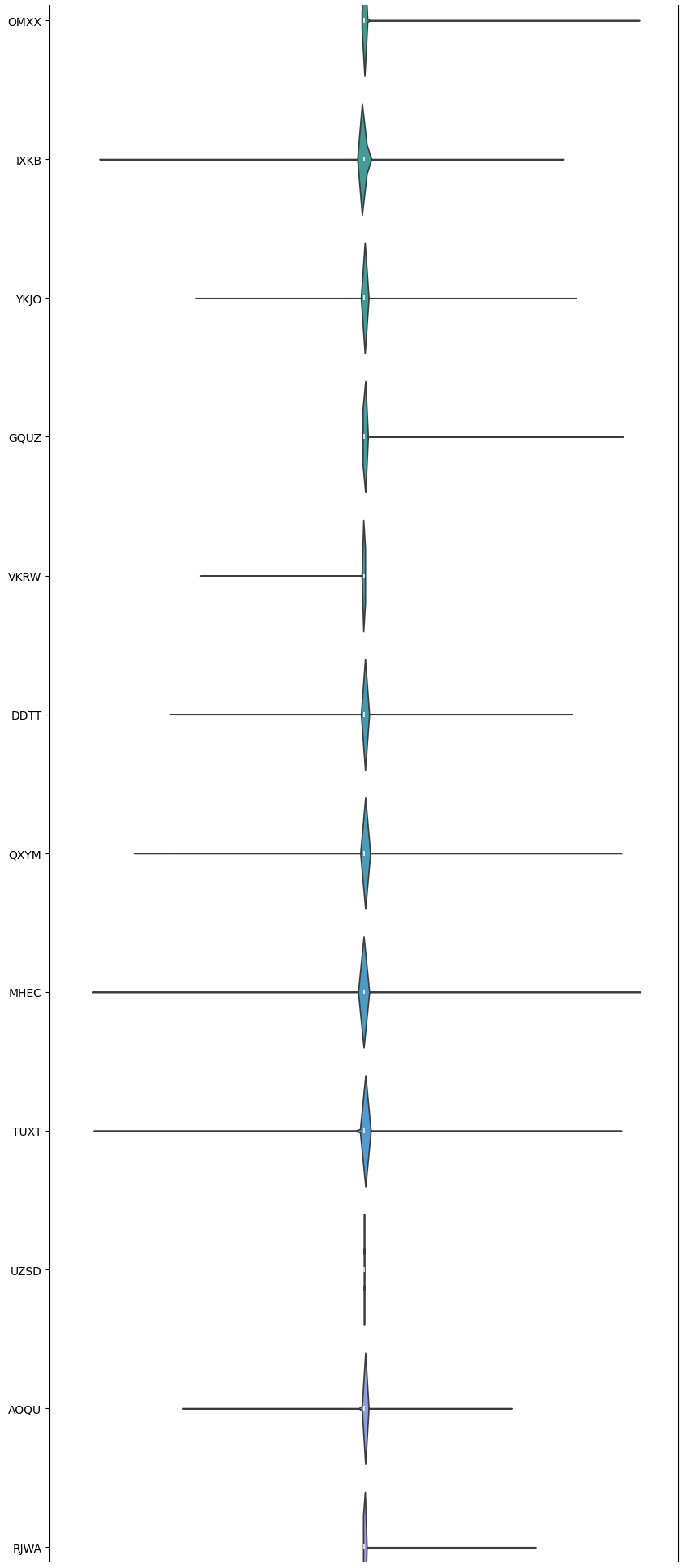
Although not definitive we have our first evidence that perhaps the data is not homogeneous and instead is in distinct types by the fact that we can clearly see three clusters in the `RESP__0` and `RESP__3` scatter matrix and to a lesser extent see a similar clustering in the `RESP__3` and `RESP__4` scatter matrix.

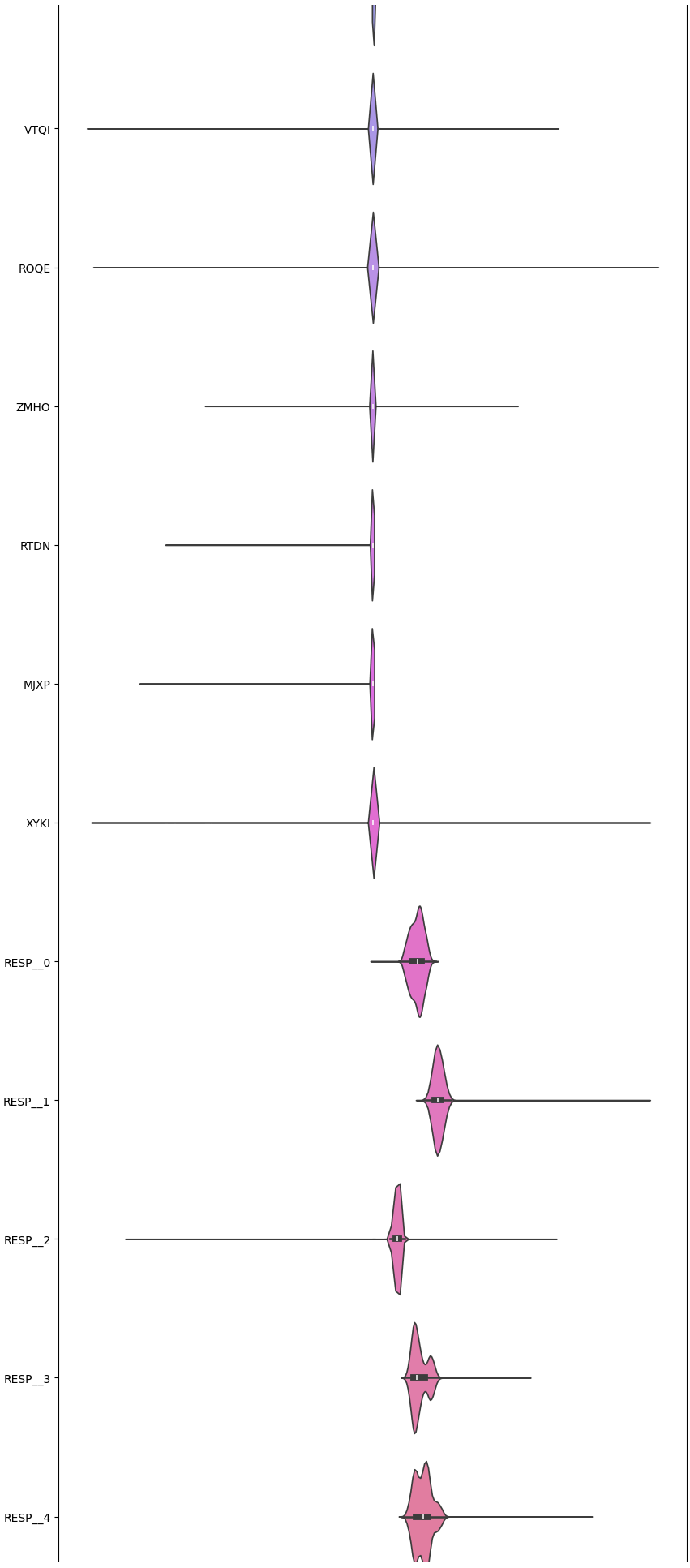
We could try to do something similar with the 40 explanatory variables but the problem is that to see all the relationships, not just the block of five columns they happen to be neighbours with we would have to create a `scatter_matrix` with all 40 variables which would still be very hard to read. Instead, we will use other tools to better understand our data.

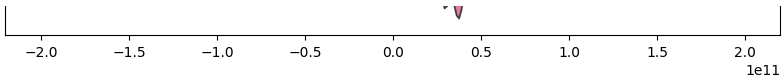
In [81]:
```
plt.figure(figsize=(10,100))
sns.violinplot(data=df_impute, orient='h')
plt.show()
```


ZYTI
WIBD
KVJI
CSTT
OFSQ
RXUY
GWEE
NANX
PMLC
JIEK
GXSO


PTAX
RLCO
RNOZ
QRWO
PSSJ
WBYY
BDGK
FWKW
PFKF
ZKOY
XBUG


OMXX
IXKB
YKJO
GQUZ
VKRW
DDTT
QXYM
MHEC
TUXT
UZSD
AOQU
RJWA


VTQI
ROQE
ZMHO
RTDN
MJXP
XYKI
RESP__0
RESP__1
RESP__2
RESP__3
RESP__4



To better understand the outliers we use violin plots which is similar to a box-plot and to a histogram but allows us to see a smoother distribution of data as it uses the kernel density function we used earlier. What becomes clear is we have very extreme outliers.

We can group the data into 4 groups

- No outliers (at the scale of all the data) such as `UZSD` and `QRWO`
- Negative outliers such as `MJXP` and `VKRW`
- Positive outliers such as `OMXX` and `GQUZ`
- Positive and negative outliers such as `XYKI` and `WIBD`

We now have the difficult choice of whether to remove the outliers or not. Given that the outliers are very extreme perhaps we should remove them as they are likely to bias any data analysis techniques we might use. On the other hand we are told very little about the underlying dataset or the data generating process so perhaps the outliers are because of measurement error and therefore could be safely removed, or perhaps the exact opposite is true and genetics research is extremely accurate and the outliers are the most interesting points to consider. Given that we do not have enough information about the outliers currently we shall try both for now.

In [82]:
```
df_no_outliers = df_impute.copy()
no_outliers = ['QRWO', 'UZSD']
```

Following our visual observation of the data coming in four groups we shall treat each group differently. The group with no outliers is the easiest where we shall not do anything.

In [83]:
```
both_outliers = ['ZYTI', 'WIBD', 'OFSQ', 'RXUY', 'NANX', 'JIEK', 'GXSO', 'PTAX', 'RNOZ', 'PSSJ', 'WBYY', 'XBUG', 'IXKB', 'YKJO', 'DDTT', 'QXYM', 'MHEC', 'TUXT', 'AOQU', 'VTQI', 'ROQE',
n = 5 # also k factor
for col in both_outliers:
    median = df_no_outliers[col].median()
    mad = (df_no_outliers[col] - median).abs().median()
    upper_bound = median + n * mad
    lower_bound = median - n * mad
    upper_cond = df_no_outliers[col] > upper_bound
    lower_cond = df_no_outliers[col] < lower_bound
    df_no_outliers.loc[upper_cond | lower_cond, col] = np.nan
    new_median = df_no_outliers[col].median()
    df_no_outliers[col] = df_no_outliers[col].fillna(new_median)
```

For the group with positive and negative outliers, we could use a Z-Score [9] but the problem with this approach is it uses the mean and standard deviation which are biased by skewed distributions which our analysis of the `scatter_matrix` and imputation of `null` showed us is a problem in our data.

Thus, rather than the formula

$$\mu \pm n * \sigma$$

Instead, use a modified Z-score formulation where we use the median rather than the mean and the median absolute deviation (MAD) rather than the standard deviation where we add a correction k so our estimates of standard deviation are consistent. [10]

$$med \pm n * k * MAD$$

If the data was normally distributed then we would use a k of 1.4826 [10], however as the data is skewed and we are only concerned with extreme outliers we will be conservative and use an `n * k = 3`.

As we did with the `null` before we replace them with the median of the new column without the outliers - note we do not replace with the old median as even this was skewed by the outliers. We could have also considered doing a transformation such as taking logs or a process like winsorization where instead of removing the outliers and replacing them with a median we replace with a capped value, we choose to replace with median as it makes the least assumptions about the underlying data. [11]

[8] https://en.wikipedia.org/wiki/Standard_score

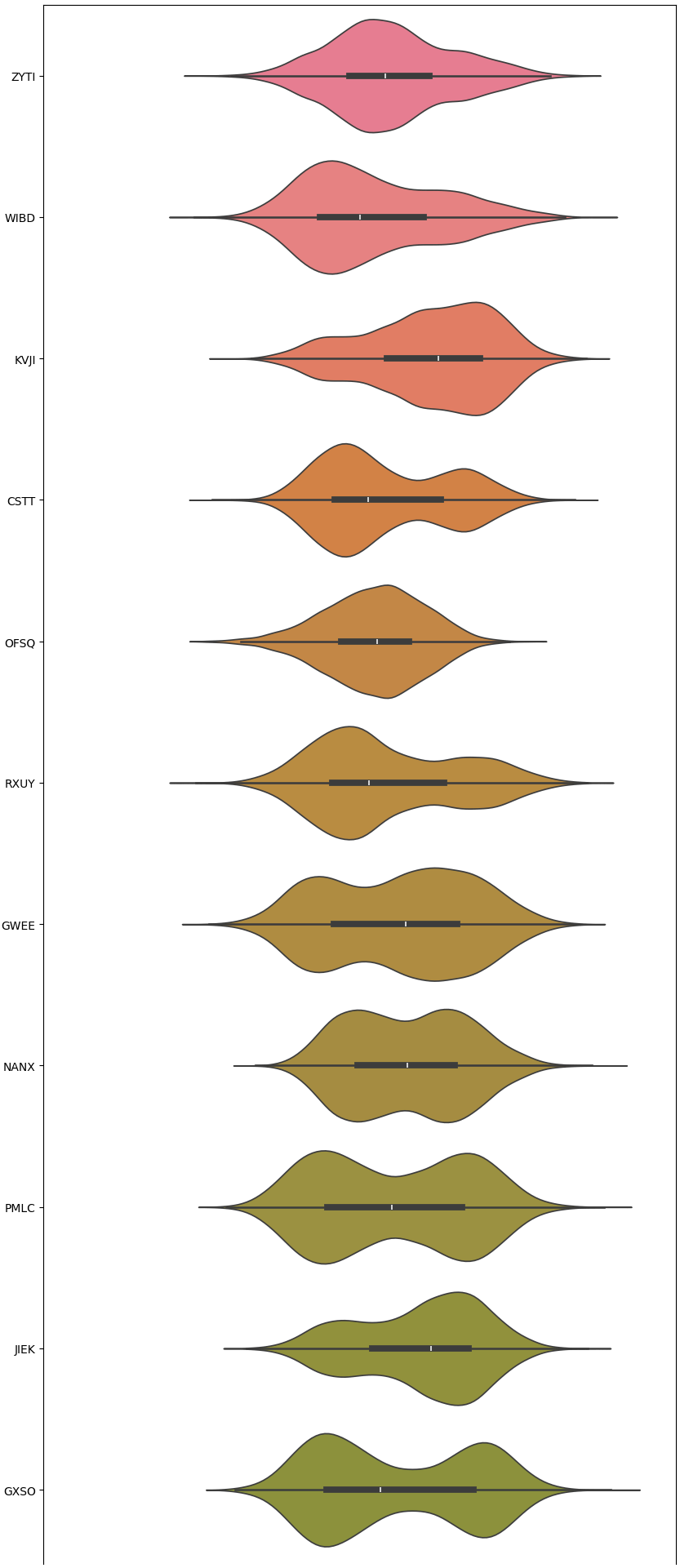
[9] https://en.wikipedia.org/wiki/Median_absolute_deviation

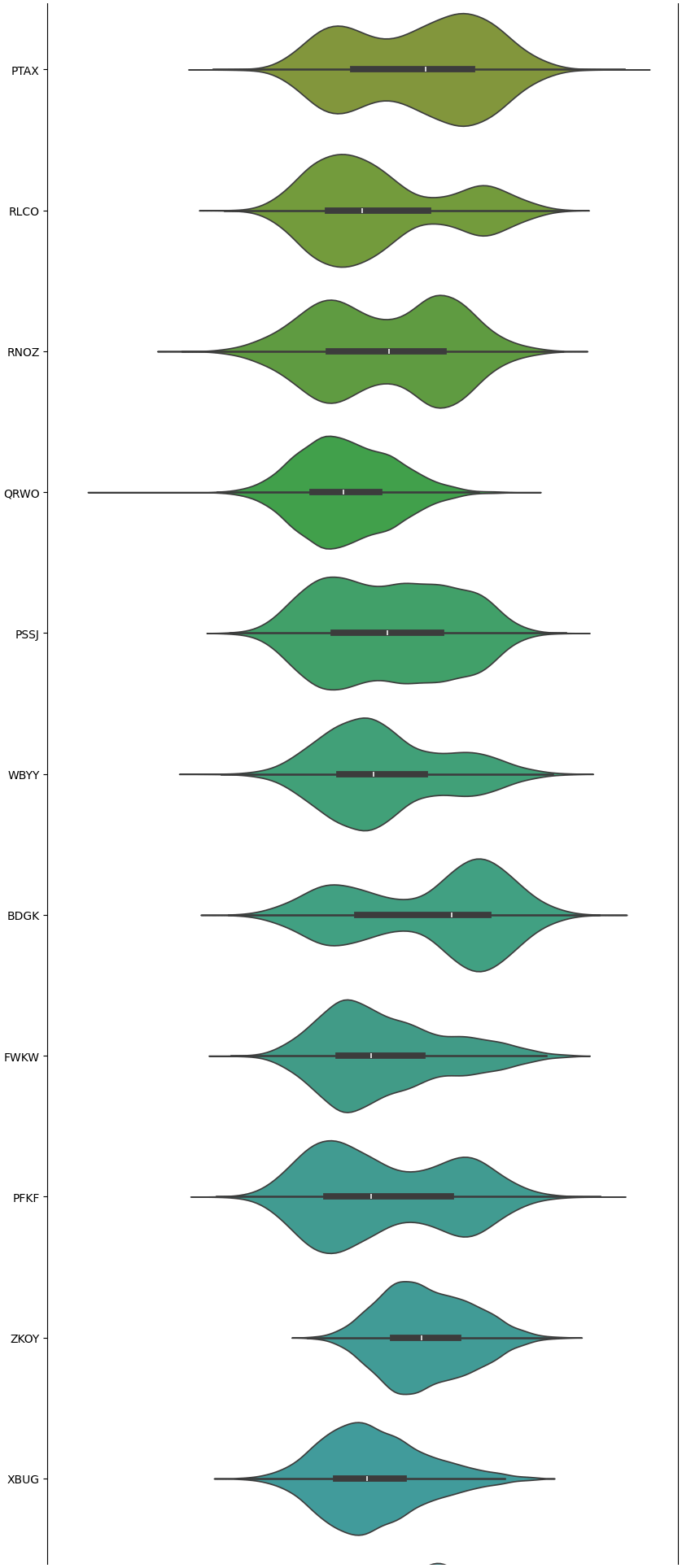
[10] https://en.wikipedia.org/wiki/Winsorizing

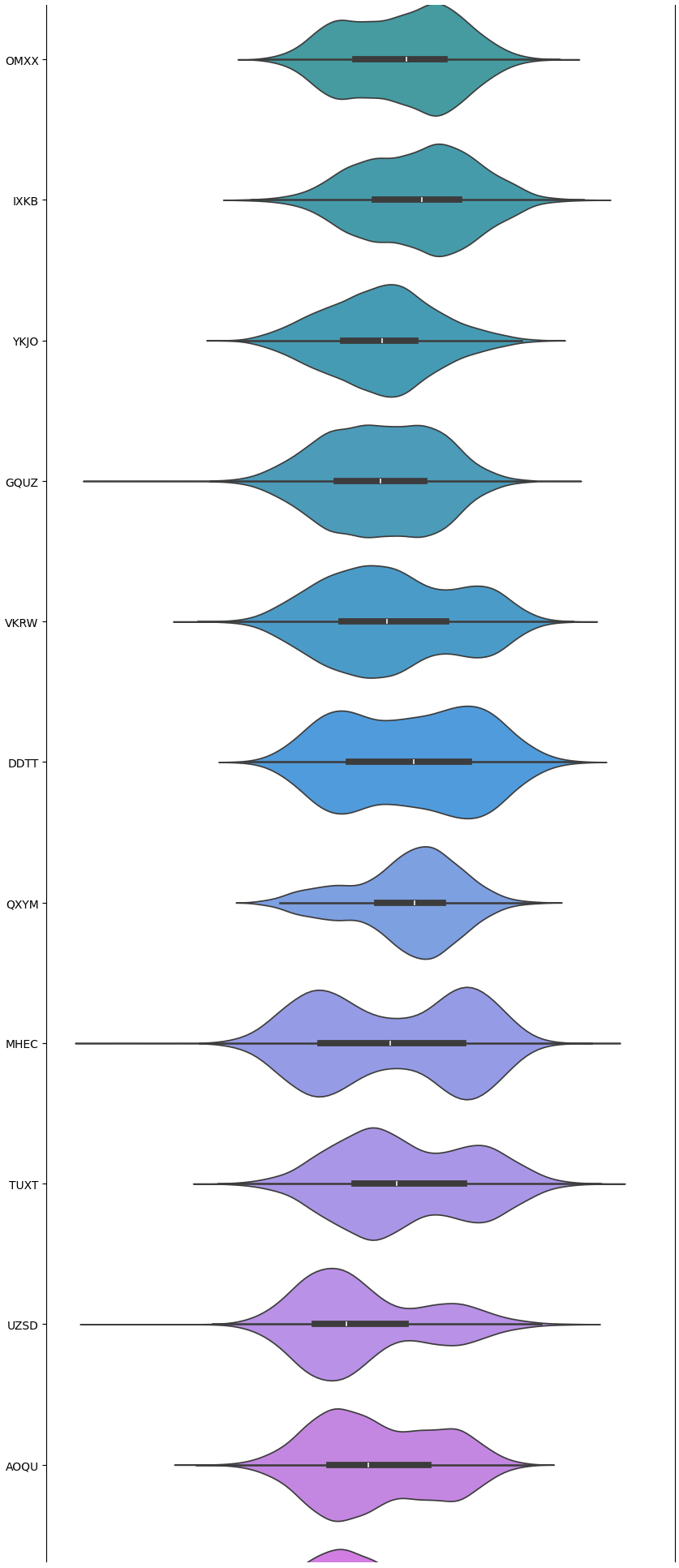
In [84]:
```
n = 5 # also k factor

left_outliers = ['KVJI', 'CSTT', 'GWEE', 'BDGK', 'PFKF', 'VKRW', 'RTDN', 'MJXP', 'RESP__0']
for col in left_outliers:
    median = df_no_outliers[col].median()
    mad = (df_no_outliers[col] - median).abs().median()
    lower_bound = median - n * mad
    lower_cond = df_no_outliers[col] < lower_bound
    df_no_outliers.loc[lower_cond, col] = np.nan
    new_median = df_no_outliers[col].median()
    df_no_outliers[col] = df_no_outliers[col].fillna(new_median)

right_outliers = ['PMLC', 'RLCO', 'FWKW', 'ZKOY', 'OMXX', 'GQUZ', 'RJWA', 'RESP__1', 'RESP__3', 'RESP__4']
for col in right_outliers:
    median = df_no_outliers[col].median()
    mad = (df_no_outliers[col] - median).abs().median()
    upper_bound = median + n * mad
    upper_cond = df_no_outliers[col] > upper_bound
    df_no_outliers.loc[upper_cond, col] = np.nan
    new_median = df_no_outliers[col].median()
    df_no_outliers[col] = df_no_outliers[col].fillna(new_median)
```

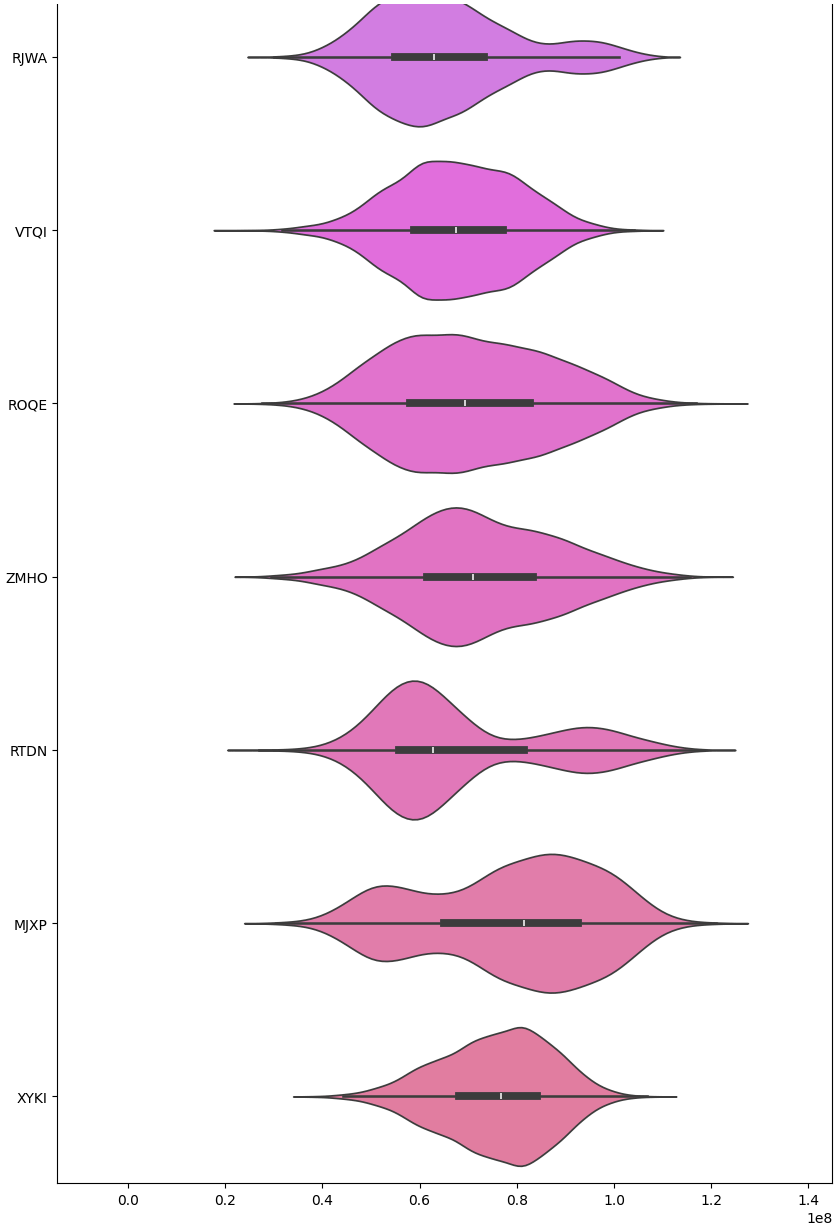
We could try one-sided approaches such as taking a percentile of the data but this is not ideal as we do not want to make assumptions about the number of outliers. Instead we modify our median-MAD approach but only apply it to one side.

In [85]:
```
num_cols = len(no_outliers) + len(both_outliers) + len(left_outliers) + len(right_outliers)
print(num_cols)

plt.figure(figsize=(10,90))
sns.violinplot(data=df_no_outliers.iloc[:, :-5], orient='h')
plt.show()
```

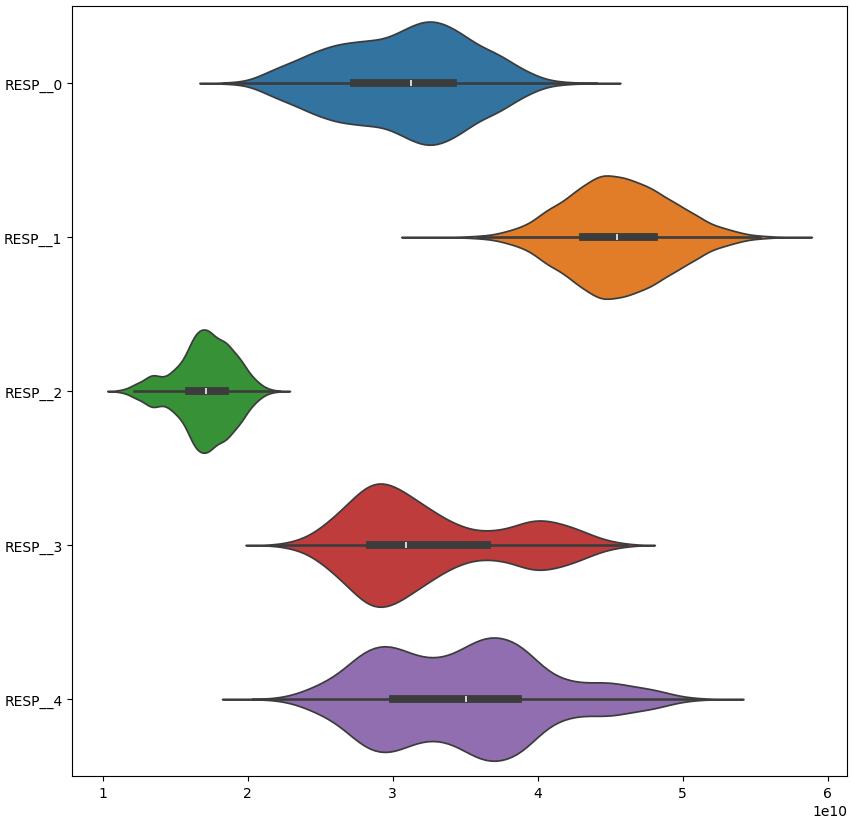

ZYTI
WIBD
KVJI
CSTT
OFSQ
RXUY
GWEE
NANX
PMLC
JIEK
GXSO


PTAX
RLCO
RNOZ
QRWO
PSSJ
WBYY
BDGK
FWKW
PFKF
ZKOY
XBUG


OMXX
IXKB
YKJO
GQUZ
VKRW
DDTT
QXYM
MHEC
TUXT
UZSD
AOQU



In [86]:
```
plt.figure(figsize=(10,10))
sns.violinplot(data=df_no_outliers.iloc[:, -5:], orient='h')
plt.show()
```



Here we just confirm that we have grouped all the columns and haven't missed any of the 45 and we verify our outlier replacement by replotting the violins again where we can see that we have successfully removed most of the outliers.

In particular, we can now see much more clearly that the data is not uniform but instead is left-skewed (`KVJI`), right-skewed (`XBUG`), unimodal (`RESP__1`) and multi-modal (`MHEC`).

In [87]:
```
df_no_outliers.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 4976 entries, 0 to 4999
Data columns (total 45 columns):
 #   Column   Non-Null Count  Dtype  
---  ------   --------------  -----  
 0   ZYTI     4976 non-null   float64
 1   WIBD     4976 non-null   float64
 2   KVJI     4976 non-null   float64
 3   CSTT     4976 non-null   float64
 4   OFSQ     4976 non-null   float64
 5   RXUY     4976 non-null   float64
 6   GWEE     4976 non-null   float64
 7   NANX     4976 non-null   float64
 8   PMLC     4976 non-null   float64
 9   JIEK     4976 non-null   float64
 10  GXSO     4976 non-null   float64
 11  PTAX     4976 non-null   float64
 12  RLCO     4976 non-null   float64
 13  RNOZ     4976 non-null   float64
 14  QRWO     4976 non-null   float64
 15  PSSJ     4976 non-null   float64
 16  WBYY     4976 non-null   float64
 17  BDGK     4976 non-null   float64
 18  FWKW     4976 non-null   float64
 19  PFKF     4976 non-null   float64
 20  ZKOY     4976 non-null   float64
 21  XBUG     4976 non-null   float64
 22  OMXX     4976 non-null   float64
 23  IXKB     4976 non-null   float64
 24  YKJO     4976 non-null   float64
 25  GQUZ     4976 non-null   float64
 26  VKRW     4976 non-null   float64
 27  DDTT     4976 non-null   float64
 28  QXYM     4976 non-null   float64
 29  MHEC     4976 non-null   float64
 30  TUXT     4976 non-null   float64
 31  UZSD     4976 non-null   float64
 32  AOQU     4976 non-null   float64
 33  RJWA     4976 non-null   float64
 34  VTQI     4976 non-null   float64
 35  ROQE     4976 non-null   float64
 36  ZMHO     4976 non-null   float64
 37  RTDN     4976 non-null   float64
 38  MJXP     4976 non-null   float64
 39  XYKI     4976 non-null   float64
 40  RESP__0  4976 non-null   float64
 41  RESP__1  4976 non-null   float64
 42  RESP__2  4976 non-null   float64
 43  RESP__3  4976 non-null   float64
 44  RESP__4  4976 non-null   float64
dtypes: float64(45)
memory usage: 1.7 MB
```

We verify that all the nulls have been replaced. Doing this helped me catch a mistake where an earlier version of the code assumed `fillna()` updates in place which by default it does not.

One thing we considered doing is standardising the data. This can have a large number of benefits including:

- Facilitating regularization
- Improving Convergence
- Avoiding the dominance of large values
- Assisting Distance-Based algorithms.

At their heart all these benefits come from equalizing. For our purposes, our data is actually already quite equal, especially after removing outliers where the explanatory variables are all in the millions range, and similarly for our response variables which are all in the billions range so for now we shall not standardise our data. This is because standardisation can also come with some disadvantages including loss of interpretability, the potential of information loss and perhaps most significantly standardisation, involving the mean and standard deviation, assumes normally distributed data which clearly we do not have.

To test our hypothesis that the data comes in distinct types we will only analyse the explanatory variables. Our first approach is to use principal components analysis (PCA) which is a linear dimensionality reduction technique which tries to find `p` unit (meaning of length one) vectors which best explain the data. This can help us reduce the 40 explanatory variables to a more manageable two or three variables. [12]

It is worth noting that this technique does not pick the two or three most important variables but instead makes new variables which are linear combinations of those variables. Mathematically, these new variables form an orthonormal basis of which means they span (i.e. cover) the entire vector space such that any variable in our data can be expressed as a combination of them. They also have some nice properties like being normalized which means they are unit vectors of length one and are orthogonal which means they are at right-angles and effectively don't try to repeat the work of other vectors. These vectors are known as eigenvectors or principal components and their corresponding eigenvalues represent how much variance in the data is explained by that eigenvector which is what enables us to rank the principal components in terms of importance.[13]

[12] https://en.wikipedia.org/wiki/Principal_component_analysis

[13] https://en.wikipedia.org/wiki/Orthonormal_basis

In [88]:
```
from sklearn.decomposition import PCA

df_x = df_no_outliers.iloc[:, :40]
pca = PCA()
pca.fit(df_x)
df_x
```

Out[88]:

	ZYTI	WIBD	KVJI	CSTT	OFSQ	RXUY	GWEE	NANX	PMLC	JIEK	GXSO	PTAX	RLCO	RNOZ	QRWO	PSSJ	WBYY	BDGK	FWKW	
0	7.12e+07	6.68e+07	9.86e+07	5.22e+07	8.28e+07	6.06e+07	7.50e+07	8.37e+07	5.02e+07	6.71e+07	7.71e+07	8.27e+07	6.83e+07	8.69e+07	5.14e+07	7.69e+07	1.05e+08	8.91e+07	7.79e+07	6
1	5.80e+07	5.22e+07	9.83e+07	6.49e+07	5.77e+07	6.96e+07	9.30e+07	9.51e+07	6.38e+07	9.13e+07	1.12e+08	9.82e+07	6.46e+07	8.14e+07	5.88e+07	8.97e+07	3.93e+07	5.97e+07	5.60e+07	6
2	8.93e+07	5.91e+07	9.03e+07	6.83e+07	8.93e+07	6.70e+07	8.26e+07	9.48e+07	6.09e+07	5.40e+07	6.46e+07	8.04e+07	5.96e+07	8.64e+07	7.96e+07	8.63e+07	9.92e+07	9.37e+07	7.78e+07	5
3	4.16e+07	8.73e+07	1.02e+08	9.09e+07	9.47e+07	4.40e+07	7.62e+07	6.79e+07	4.83e+07	6.83e+07	8.87e+07	1.00e+08	7.01e+07	6.86e+07	7.84e+07	5.79e+07	7.12e+07	1.01e+08	5.18e+07	6
4	9.06e+07	5.74e+07	5.76e+07	7.55e+07	6.07e+07	8.32e+07	8.56e+07	8.17e+07	9.73e+07	7.80e+07	5.50e+07	5.34e+07	5.10e+07	6.51e+07	5.69e+07	9.71e+07	5.80e+07	9.69e+07	1.02e+08	7
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
4995	7.41e+07	5.83e+07	9.27e+07	6.78e+07	6.10e+07	1.05e+08	3.67e+07	6.85e+07	9.24e+07	5.37e+07	1.07e+08	9.00e+07	8.58e+07	7.79e+07	5.84e+07	8.64e+07	5.73e+07	6.48e+07	7.46e+07	7
4996	6.63e+07	4.49e+07	9.36e+07	5.41e+07	5.61e+07	6.06e+07	7.73e+07	9.62e+07	6.80e+07	9.94e+07	9.73e+07	9.74e+07	6.15e+07	8.99e+07	5.03e+07	7.15e+07	4.01e+07	4.38e+07	5.80e+07	6
4997	9.74e+07	5.16e+07	6.56e+07	9.80e+07	7.70e+07	8.18e+07	6.96e+07	8.30e+07	8.47e+07	9.07e+07	4.73e+07	7.65e+07	9.33e+07	5.24e+07	5.20e+07	7.26e+07	7.14e+07	8.46e+07	7.55e+07	6
4998	5.86e+07	6.68e+07	7.13e+07	5.58e+07	7.28e+07	1.11e+08	5.37e+07	6.26e+07	9.16e+07	5.82e+07	9.01e+07	8.24e+07	8.49e+07	7.81e+07	6.47e+07	9.51e+07	6.43e+07	5.93e+07	7.50e+07	6
4999	5.58e+07	7.69e+07	8.27e+07	5.56e+07	7.56e+07	6.76e+07	7.99e+07	9.47e+07	4.97e+07	6.17e+07	6.81e+07	9.00e+07	7.02e+07	8.25e+07	5.02e+07	7.98e+07	9.54e+07	9.76e+07	8.90e+07	5

4976 rows × 40 columns

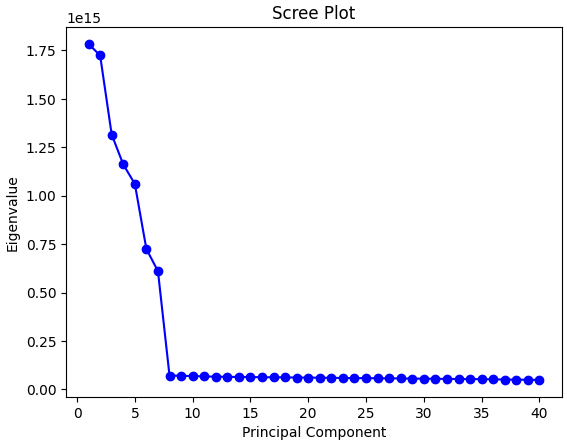
Calling the scikit-learn package method can be very simple but it is important to understand how the algorithm is working under the hood. The key steps are [14]

1. The data is centred by subtracting the mean from each feature. The data might also be scaled to unit variance.
2. The covariance matrix is computed
3. Eigendecomposition is performed on the covariance matrix to find the eigenvalues and eigenvectors. For small matrics this can be done symbolically by computing the characteristic polynomial but in practice they use numerical methods like the power method and the QR algorithm.
4. Sort and select the `p` most important principal components by their eigenvalues
5. Use the eigenvectors to project the data onto the new space and do the clustering or other algorithms on.

[14] https://en.wikipedia.org/wiki/Eigendecomposition_of_a_matrix

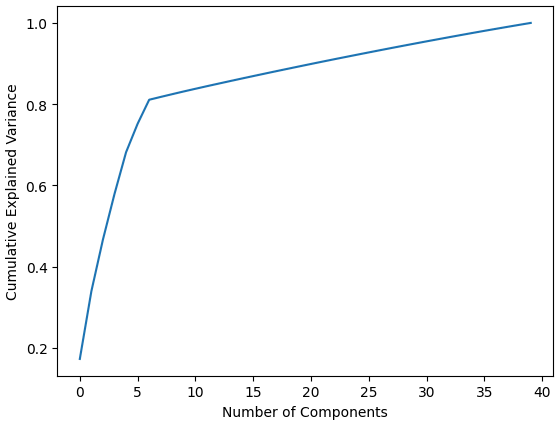
In [89]:
```
plt.plot(range(1, len(pca.explained_variance_) + 1), pca.explained_variance_, 'bo-')
plt.xlabel('Principal Component')
plt.ylabel('Eigenvalue')
plt.title('Scree Plot')
```

Out[89]: `Text(0.5, 1.0, 'Scree Plot')`



This Scree plot shows a promising result where on the y-axis we can see the size of the eigenvalue and on the x-axis we have ranked the principal components (or eigenvectors) by their importance. What we see is that the first seven eigenvalues are clearly much larger than all the other variables which are less than 0.1^x10^15 compared to 1.75x10^15 for the largest, a roughly 17x difference.

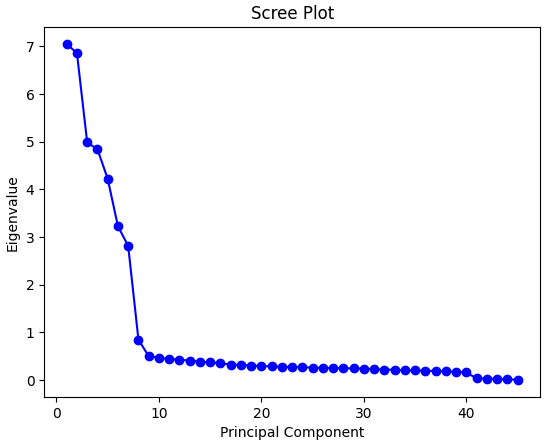
In [90]:
```
plt.plot(np.cumsum(pca.explained_variance_ratio_))
plt.xlabel('Number of Components')
plt.ylabel('Cumulative Explained Variance')
plt.show()
```



This can be further confirmed by an explained variance ratio chart where again we the vast majority of the variance, more than 80%, is explained by the first six principal components.

In [91]:
```
from sklearn.preprocessing import PowerTransformer

transformer = PowerTransformer(method='yeo-johnson')
df_transformed = transformer.fit_transform(df_no_outliers)
pca_transformed = PCA()
pca_transformed.fit(df_transformed)

plt.plot(range(1, len(pca_transformed.explained_variance_) + 1), pca_transformed.explained_variance_, 'bo-')
plt.xlabel('Principal Component')
plt.ylabel('Eigenvalue')
plt.title('Scree Plot')
```

Out[91]:
```
Text(0.5, 1.0, 'Scree Plot')
```



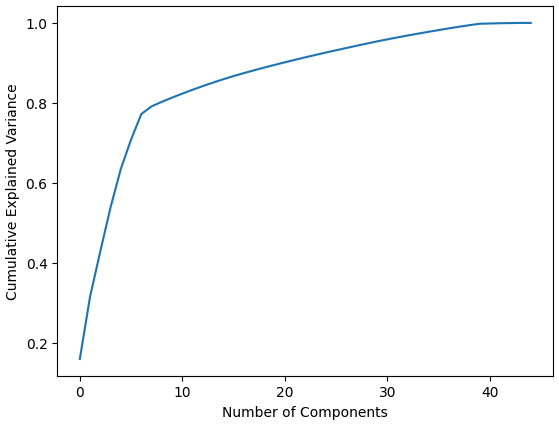
We also want to try transforming the data. We could use the `StandardScaler` which works, as previously described by subtracting the mean and dividing by the standard deviation but it assumes the Gaussian distribution for optimal results which we do not have. We could also consider the `RobustScaler` which uses the median ane interquartile ranges ane is more robust to outliers similar to how we used MAD in our outlier removal process.[15]

There are then a number of transformations which map the data to make them as normal as possible which are known as power transformations. We considered using Box-Cox as ifs algorithm has a direct relationship to logarithmic transformation but unfortunately it only works on positive data. So instead we use the Yeo-Johnson transformation which is more flexible [15].

Interestingly the transformed data performs roughly the same with this time eight principal components with larger eigenvalues than the rest.

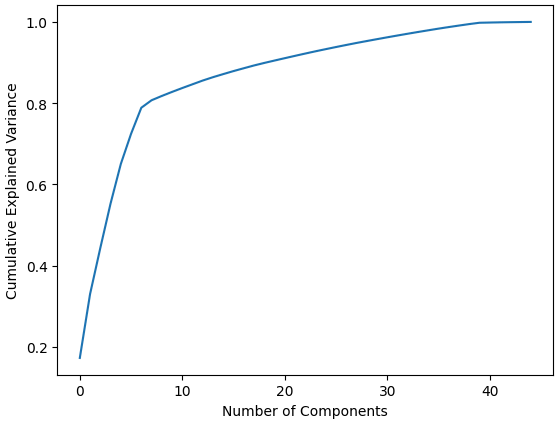
[15] https://scikit-learn.org/stable/modules/preprocessing.html

In [92]:
```
plt.plot(np.cumsum(pca_transformed.explained_variance_ratio_))
plt.xlabel('Number of Components')
plt.ylabel('Cumulative Explained Variance')
plt.show()
```



The amount of explained variance is very similar to our untransformed data with six explaining almost 80% of the variance.

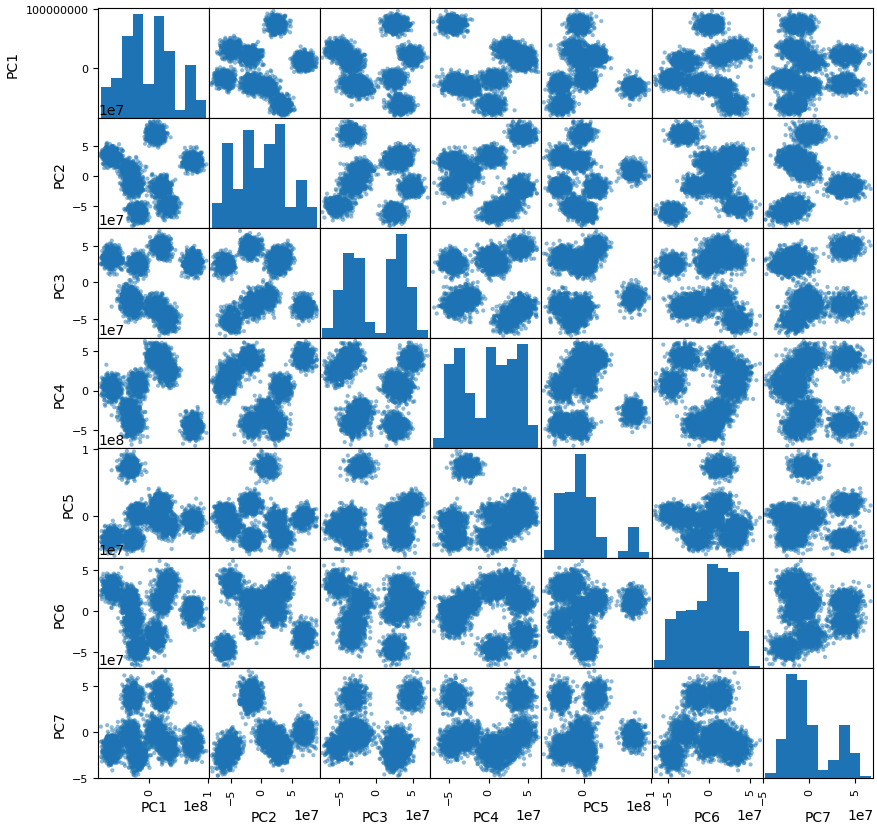
In [93]:
```
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()
df_scaled = scaler.fit_transform(df_no_outliers)
pca_scaler = PCA()
pca_scaler.fit(df_scaled)

plt.plot(np.cumsum(pca_scaler.explained_variance_ratio_))
plt.xlabel('Number of Components')
plt.ylabel('Cumulative Explained Variance')
plt.show()
```



For ths sake of completeness we also try the `StandardScaler` and see a similar story where 80% of the variance is explained by roughly six components.

In [94]:
```
pca = PCA(n_components=7)
np_pca = pca.fit_transform(df_x)
df_pca = pd.DataFrame(np_pca, columns=[f"PC{i}" for i in range(1, 8)])
pd.plotting.scatter_matrix(df_pca, figsize=(10, 10))
```

Out[94]:
```
array([[<Axes: xlabel='PC1', ylabel='PC1'>,
        <Axes: xlabel='PC2', ylabel='PC1'>,
        <Axes: xlabel='PC3', ylabel='PC1'>,
        <Axes: xlabel='PC4', ylabel='PC1'>,
        <Axes: xlabel='PC5', ylabel='PC1'>,
        <Axes: xlabel='PC6', ylabel='PC1'>,
        <Axes: xlabel='PC7', ylabel='PC1'>],
       [<Axes: xlabel='PC1', ylabel='PC2'>,
        <Axes: xlabel='PC2', ylabel='PC2'>,
        <Axes: xlabel='PC3', ylabel='PC2'>,
        <Axes: xlabel='PC4', ylabel='PC2'>,
        <Axes: xlabel='PC5', ylabel='PC2'>,
        <Axes: xlabel='PC6', ylabel='PC2'>,
        <Axes: xlabel='PC7', ylabel='PC2'>],
       [<Axes: xlabel='PC1', ylabel='PC3'>,
        <Axes: xlabel='PC2', ylabel='PC3'>,
        <Axes: xlabel='PC3', ylabel='PC3'>,
        <Axes: xlabel='PC4', ylabel='PC3'>,
        <Axes: xlabel='PC5', ylabel='PC3'>,
        <Axes: xlabel='PC6', ylabel='PC3'>,
        <Axes: xlabel='PC7', ylabel='PC3'>],
       [<Axes: xlabel='PC1', ylabel='PC4'>,
        <Axes: xlabel='PC2', ylabel='PC4'>,
        <Axes: xlabel='PC3', ylabel='PC4'>,
        <Axes: xlabel='PC4', ylabel='PC4'>,
        <Axes: xlabel='PC5', ylabel='PC4'>,
        <Axes: xlabel='PC6', ylabel='PC4'>,
        <Axes: xlabel='PC7', ylabel='PC4'>],
       [<Axes: xlabel='PC1', ylabel='PC5'>,
        <Axes: xlabel='PC2', ylabel='PC5'>,
        <Axes: xlabel='PC3', ylabel='PC5'>,
        <Axes: xlabel='PC4', ylabel='PC5'>,
        <Axes: xlabel='PC5', ylabel='PC5'>,
        <Axes: xlabel='PC6', ylabel='PC5'>,
        <Axes: xlabel='PC7', ylabel='PC5'>],
       [<Axes: xlabel='PC1', ylabel='PC6'>,
        <Axes: xlabel='PC2', ylabel='PC6'>,
        <Axes: xlabel='PC3', ylabel='PC6'>,
        <Axes: xlabel='PC4', ylabel='PC6'>,
        <Axes: xlabel='PC5', ylabel='PC6'>,
        <Axes: xlabel='PC6', ylabel='PC6'>,
        <Axes: xlabel='PC7', ylabel='PC6'>],
       [<Axes: xlabel='PC1', ylabel='PC7'>,
        <Axes: xlabel='PC2', ylabel='PC7'>,
        <Axes: xlabel='PC3', ylabel='PC7'>,
        <Axes: xlabel='PC4', ylabel='PC7'>,
        <Axes: xlabel='PC5', ylabel='PC7'>,
        <Axes: xlabel='PC6', ylabel='PC7'>,
        <Axes: xlabel='PC7', ylabel='PC7'>]], dtype=object)
```



Having established from the scree plot and variance chart that seven components can explain most of the variance we can then see the clear clustering that was not possible to see when we tried to draw all 40 features in the scatter matrix. This is most obvious when looking at the `PC1-PC2` chart where we can clearly see eight clusters. In some other charts less clusters can be discerned for example in `PC5-PC7` we can only see six clusters.

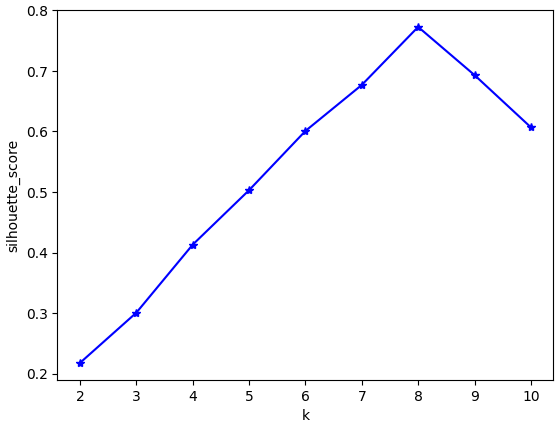
We could have tried other algorithms. For example, hierarchical clustering which does not require specifying the number of clusters up front. However, given the limited amount of information we have about the dataset we have no apriori reason to think that the data is hierarchical [16]. Other approaches include DBScan which is Density-Based Spatial Clustering of Applications with Noise which is best used when the data has arbitrarily shaped clusters and also doesn't require specifying the number of clusters up front but this is not necessary as the scatter matrix above looks like all the clusters are roughly the same size [17]. Finally, we could have used Gaussian Mixture Models which works well when you have multi-modal data made up of Gaussian-like data but again that does not seem to be what we have, instead a lot of our data looks more logarithmic in nature [18].

[16] https://en.wikipedia.org/wiki/Hierarchical_clustering

[17] https://en.wikipedia.org/wiki/DBSCAN

[18] https://en.wikipedia.org/wiki/Mixture_model

In [95]:
```
from sklearn.cluster import KMeans
from sklearn import metrics

def k_silhouette(X, clusters): # [11] i.e. code taken from Dr Collins - Classical Machine Learning Workbook
    K = range(2, clusters + 1)
    score = []
    for k in K:
        kmeans = KMeans(n_clusters=k)
        kmeans.fit(X)
        labels = kmeans.labels_
        score.append(metrics.silhouette_score(X, labels, metric='euclidean'))

    plt.plot(K, score, 'b*-')
    plt.xlabel('k')
    plt.ylabel('silhouette_score')

    plt.show()

k_silhouette(df_pca, 10)
```



The `KMeans` algorithm involves initializing the number of clusters `k` and randomly selecting the number the `k` points to serve as the initial centroids. Then the distance, in this case euclidean, is calculated between every point and every centroid assigning each point to its nearest centroid. Then each centroid is re-calculated by taking the mean of all the data points in that cluster. This process it repeated until there the centroids stabilise or a maximum number of iterations is reached. [19].

However, the algorithm is very dependent upon the number of clusters chosen at the beginning. Given that there is no way to know up front what the optimal number is instead we use trial-and-error to see how each number of clusters performs. To compare the performance of each clustering we use the silhouette score which is a number between -1 and 1 where high numbers indicate that the object is well matched to its cluster. Typically a silhouette width of 0.7 is considered strong, a value over 0.5 is reasonable and over 0.25 is considered weak [20].

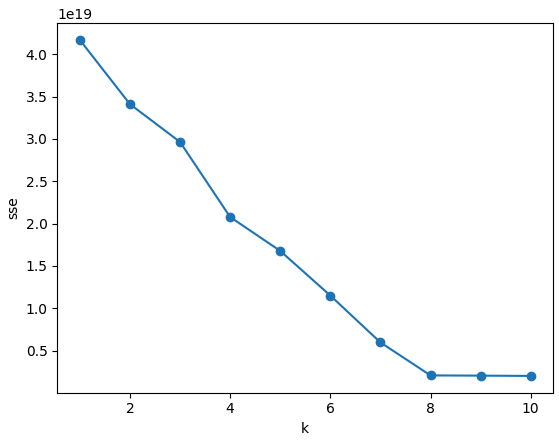
In our case, our results are show a strong silhouette score with a very clear peak at 8 clusters and a score of almost 0.8 [20]. If is worth noting that one of the reasons we perform PCA first is that the curse of dimensionality means that doing K-means on the original 40 features would not only be computationally expensive but also more ineffective as the distances between points become more similar in higher dimensional spaces [21].

[19] https://en.wikipedia.org/wiki/K-means_clustering

[20] https://en.wikipedia.org/wiki/Silhouette_(clustering)

[21] https://en.wikipedia.org/wiki/Curse_of_dimensionality

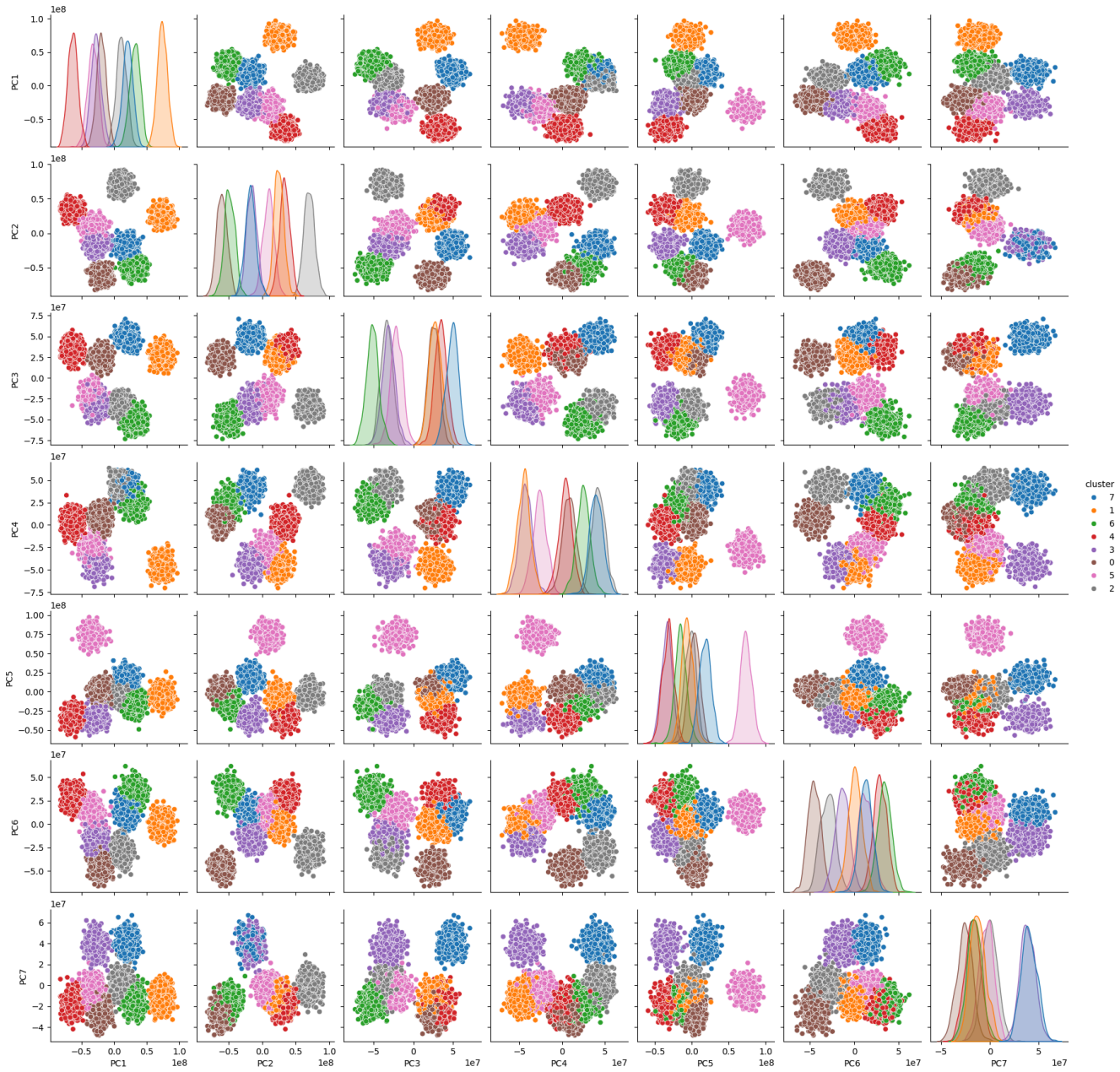
In [96]:
```
sse = []
for k in range(1, 11):
    kmeans = KMeans(n_clusters=k)
    kmeans.fit(df_pca)
    sse.append(kmeans.inertia_)
x = range(1, 11)
plt.xlabel('k')
plt.ylabel('sse')
plt.plot(x, sse, 'o-')
plt.show()
```



We use the 'Elbow' method which involves plotting a chart with the sum of the squares of the distances between each point and its cluster centre. Here we can clearly see that the 'elbow' is around eight as well where the SSE drops belw 0.5e15 versus more than 4e15 for a single cluster.

In [97]:
```
kmeans = KMeans(n_clusters=8)
cluster_predict = kmeans.fit_predict(df_pca)
df_predict = df_pca.copy()
df_predict["cluster"] = cluster_predict.astype(str)
sns.pairplot(df_predict, hue='cluster')
```

Out[97]:
```
<seaborn.axisgrid.PairGrid at 0x2909abb1f50>
```



Finally, we can use our kmeans prediction for each cluster on our seven PCA features to clearly see the eight distinct types of data represented by the eight colours. Now we can see why that some of the clusters are not separable in every pairplot. For example `PC5-PC7` looks like a paw with three seperate clusters above with purple, blue and pink, but the others are overlapping and so look like one large cluster in the bottom left. Whereas when we consider the two most important principal components we can `PC1-PC2` we can already see the eight distinct clusters.

Therefore we conclude that the wheat data is not homogeneous or random but that there are eight distinct types.

2. If the wheat is homogenenous or random, then identify the overall characteristics of the complete sample.

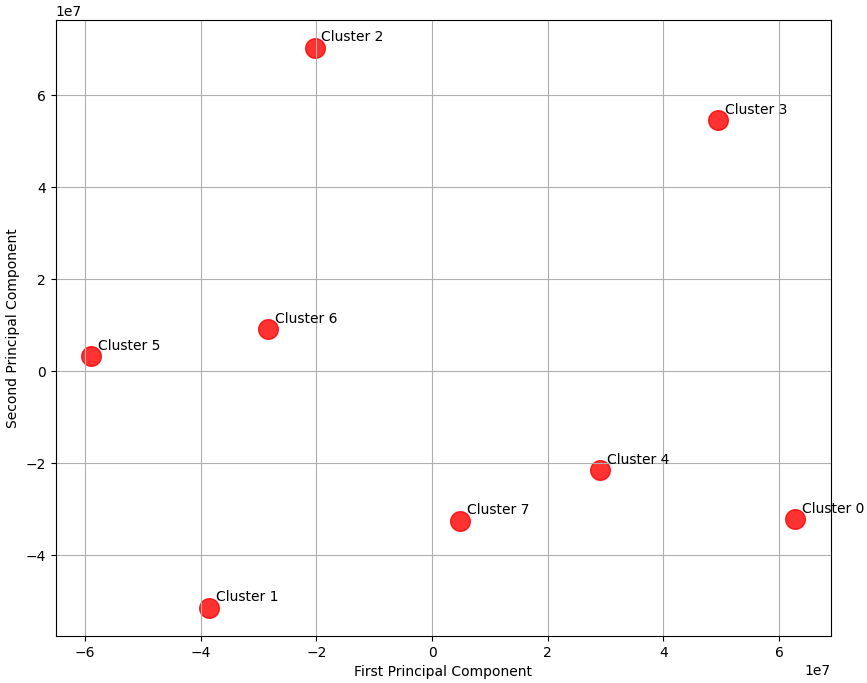
We concluded in the previous section that the data is not homogeneous or random and so answer question 3 instead.

3. If there are distinct types of wheat in the dataset, then identify the characteristics of each type of wheat.

In [98]:
```
pca = PCA(n_components=7)
df_pca = pca.fit_transform(df_x)

kmeans = KMeans(n_clusters=8)
kmeans.fit(df_pca)

pca_centres = kmeans.cluster_centers_
feature_centres = pca.inverse_transform(pca_centres)
feature_centres
```

Out[98]:
```
array([[88448001.1333175 , 61342371.38274013, 55779041.07009545,
        62915837.24315398, 50615440.57440168, 95051962.57258865,
        76637613.83933033, 72924160.82353662, 94764651.42291065,
        75646259.61444706, 52637290.81998856, 56103507.91466573,
        62889184.33427719, 53239141.20688049, 52907066.28346581,
        93036337.19079626, 58977557.15911847, 95584848.47205909,
        96476965.21871266, 86578181.19801939, 89716909.7302    ,
        88949235.23142767, 84350807.3550347 , 70087779.78973761,
        77888769.13596703, 63433933.51861309, 74531516.1879485 ,
        83140642.03988124, 86849012.42039655, 91946351.94717573,
        55318474.65979967, 58758148.1399405 , 53295452.46426091,
        56850866.29440966, 78496260.14545742, 76441699.76149927,
        65954934.78056125, 57208442.1433517 , 97212395.868873  ,
        63594843.35139872],
       [72892274.3682851 , 87122048.1358604 , 82176919.83513469,
        59002312.11712766, 77830640.34214726, 53567606.82803463,
        94746710.38687158, 59130831.19676776, 51467898.93520284,
        95518232.60773627, 60755804.33932257, 62967020.99487875,
        51726141.52262151, 83953223.29547688, 66908714.38993482,
        60558156.03383322, 72086632.72967443, 98601696.67078254,
        68873412.28426357, 90979271.34844723, 74928822.74020688,
        76302048.37738457, 93488427.17263488, 60012876.52371944,
        69616227.43053472, 76310930.5950143 , 96850540.57799242,
        89857539.45578256, 54630595.70761789, 53450788.84985921,
        89919462.11244857, 64165965.93008526, 86748319.49887954,
        73219524.23958212, 64599204.87443362, 67813076.73852381,
        65669949.3389558 , 59947377.05684397, 89962741.27646343,
        87804992.62577216],
       [68846753.95681483, 54232694.9176927 , 92490562.12401703,
        57567767.92371108, 58430472.09014502, 65513042.50014889,
        97039643.25108387, 95509298.65666305, 65101816.66010966,
        94258144.7913048 , 97733716.5426366 , 96063973.97569315,
        68439978.45228103, 92497171.29752153, 52063839.31702869,
        77511011.55095956, 54777307.27067419, 52950193.34069047,
        57886102.09435633, 54065892.02507694, 87483285.80224878,
        62182507.96444229, 65442332.30061423, 97611918.91633435,
        90244523.16785702, 84686009.88520621, 78722480.6893866 ,
        63569167.15235315, 86449853.08706391, 93517417.32740867,
        94892060.77002235, 54752473.17408752, 55099528.51087534,
        54930114.44235059, 56794860.91008878, 58167321.39506388,
        95286803.21113378, 98124097.53048937, 84504319.91864681,
        76195798.56175648],
       [70029920.43341598, 74943473.85539602, 89670532.74653591,
        68098703.08981077, 67922877.49081698, 98075491.95857327,
        52955348.06636768, 67221855.83203977, 92587208.95726386,
        57158954.3729156 , 95394123.1605093 , 90085067.44665983,
        95291141.69263053, 79564091.43609859, 76082313.67049743,
        90941329.26319003, 61117645.57269862, 59081777.78045551,
        75269291.46939188, 70641165.75759548, 95811974.99071774,
        70110448.70760664, 83256799.3220674 , 83213670.51066208,
        72982927.63536006, 55469666.57138341, 54006894.57346066,
        58168649.1661066 , 81059071.99013531, 50911568.00581528,
        66575142.99234235, 51349951.70957495, 67968333.3280002 ,
        61973427.49967094, 80793292.88465844, 86392194.1612533 ,
        50380505.7988096 , 90924717.17968261, 56208322.107401  ,
        84220670.01154138],
       [97130165.6400826 , 55034881.89096344, 69009057.4148529 ,
        94850285.4894659 , 69009497.16490617, 79136846.65249795,
        78835214.36054213, 86160086.32781935, 94215708.29696009,
        87486941.67797714, 54789728.27510196, 81537911.63291118,
        95632206.7795917 , 62038163.43922652, 59325985.90796389,
        60707173.03592721, 69840089.66438456, 97244054.27575968,
        61315784.497266  , 60909064.6186956 , 76206552.4134738 ,
        67885010.68166965, 75638406.20511276, 65858403.88780936,
        56751947.6017639 , 64585020.49812669, 64671845.19114268,
        76138847.59872685, 83362141.3093259 , 52566897.261009  ,
        99902054.09242159, 86749359.69638889, 80791653.92825657,
        50531691.77583265, 51063730.28119735, 51787039.70196834,
        73014639.77083787, 56751145.74444704, 50781841.24422266,
        63446974.21295395],
       [53549077.73088518, 97275309.36307749, 99281701.61051932,
        91250885.94151762, 83314887.46850836, 55009254.25831735,
        61259731.93381738, 62315933.17465385, 56616650.39182454,
        84057750.09629253, 95892124.31468269, 98414426.09061733,
        58206363.45371614, 51552262.08239493, 71751206.21947853,
        58353727.79446261, 66294545.98926888, 91882484.41310704,
        62066391.50483945, 53702775.03937888, 70577709.670233  ,
        56739375.35194807, 59748472.18287794, 84780786.64373136,
        70211446.6034445 , 52414916.84157054, 68308085.8161096 ,
        97409126.68734229, 76828843.87777086, 95217910.26824516,
        70029497.26391391, 64519529.79163358, 89974571.643438  ,
        93256503.70602617, 66596620.02224921, 72978381.72507808,
        79048548.07577112, 62284803.83927371, 73975014.99888073,
        72255950.08092634],
       [73136931.47101814, 67457394.7814256 , 98686778.41508463,
        58439385.88262246, 70257393.39755815, 60951887.51490064,
        84868318.8340625 , 91784722.92127025, 54772146.49581683,
        61030246.89352381, 74970352.9789898 , 82739930.24714054,
        66437536.3463517 , 86323606.06644587, 58342849.19387696,
        77535760.94803534, 94386124.08365542, 92700530.78777266,
        87048525.706248  , 56101787.38114613, 78322504.35072932,
        55485592.05458958, 84448973.45759004, 90277221.0332783 ,
        51430200.28132342, 78327801.4372856 , 95163837.31343625,
        56432413.25053175, 80123773.12356657, 78638248.19731884,
        75405902.14300266, 92731965.11416617, 65744851.5710576 ,
        71971672.62072718, 64924946.67085569, 94692465.68373637,
        62268057.22752561, 55648877.95763533, 96953832.6462976 ,
        76904601.3296504 ],
       [60400589.06806988, 52673472.49295449, 76422702.51947941,
        81090136.26367426, 75674577.53432491, 66105828.90796952,
        51531285.59157993, 80869941.14921609, 80503951.2133245 ,
        87784986.12002164, 60529332.94886982, 59553927.72815239,
        56926798.20010852, 57618252.09764703, 51936057.50121683,
        51863852.25314425, 90412958.88701965, 70123689.3075403 ,
        55855085.05607122, 95510149.58142391, 74362098.50144745,
        64393531.11228432, 58807167.26579897, 82918189.18195662,
        69684073.47647108, 84461548.61499196, 54680451.95208299,
        98569474.5337855 , 63054115.43402275, 63578437.49700574,
        61227915.81397676, 53525624.61283235, 57783218.88684861,
        62657338.02817196, 78425519.52911933, 56435790.18236001,
        84563086.31752393, 63178119.58173623, 78870998.03723364,
        82717493.00269409]])
```

In [99]:
```
pca_viz = PCA(n_components=2)
centres_2d = pca_viz.fit_transform(feature_centres)

plt.figure(figsize=(10, 8))
plt.scatter(centres_2d[:, 0], centres_2d[:, 1], c='red', s=200, alpha=0.8)
for i, (x, y) in enumerate(centres_2d):
    plt.annotate(f'Cluster {i}', (x, y), xytext=(5, 5), textcoords='offset points')
plt.xlabel('First Principal Component')
plt.ylabel('Second Principal Component')
plt.grid(True)
plt.show()
```



In the previous section we concluded that the data came in eight distinct types. However, to do so we used PCA which involved creating linear combinations of the 40 feature variables and then clustering them using KMeans. However, in order to identify the distinct types we need to go backwards and reverse the PCA process using `inverse_transform`. Clearly though this data is still very difficult to interpret as it is just a bunch of numbers. We can plot the cluster centres in 2D space to try and make them much clearer.

In [100…
```
loadings = pca.components_.T
df_loadings = pd.DataFrame(loadings, columns=['PC1', 'PC2', 'PC3', 'PC4', 'PC5', 'PC6', 'PC7'], index=df_x.columns)
df_loadings
```

Out[100…

	PC1	PC2	PC3	PC4	PC5	PC6	PC7
ZYTI	-1.74e-01	0.12	1.39e-01	-4.81e-02	2.44e-01	6.62e-02	-1.06e-01
WIBD	5.63e-02	-0.19	-1.19e-01	3.15e-01	-6.31e-02	-4.31e-02	-2.69e-01
KVJI	2.96e-01	-0.09	-5.10e-02	1.53e-01	9.53e-03	-5.75e-02	1.22e-01
CSTT	-7.34e-02	-0.06	-3.27e-01	-6.92e-02	1.45e-01	1.93e-01	2.63e-02
OFSQ	3.32e-02	-0.17	-1.57e-01	6.76e-02	3.65e-02	-9.38e-02	7.54e-02
RXUY	-1.60e-01	0.35	-3.48e-02	6.99e-02	-8.01e-03	2.84e-02	-6.39e-02
GWEE	9.70e-02	-0.09	3.72e-01	-1.08e-01	1.67e-01	-2.01e-02	-2.53e-01
NANX	1.03e-01	0.09	1.44e-01	-1.91e-01	1.22e-01	1.34e-01	2.70e-01
PMLC	-2.31e-01	0.33	-1.35e-01	-9.20e-02	5.10e-02	6.82e-02	8.98e-04
JIEK	1.38e-02	-0.17	-6.34e-03	-3.12e-01	2.28e-03	-4.00e-02	-2.35e-01
GXSO	3.98e-01	0.09	-1.34e-01	1.35e-01	-9.57e-02	2.12e-03	-8.10e-02
PTAX	3.21e-01	0.05	-1.32e-01	8.92e-02	1.46e-01	1.28e-01	-7.26e-02
RLCO	4.77e-03	0.27	-1.29e-01	5.29e-02	3.13e-01	-2.66e-02	1.61e-02
RNOZ	2.24e-01	0.05	2.33e-01	2.77e-02	1.04e-01	-3.23e-01	3.16e-02
QRWO	3.25e-02	-0.01	-1.27e-01	1.92e-01	3.59e-02	-1.00e-01	-1.11e-01
PSSJ	6.61e-03	0.27	1.76e-01	1.76e-01	-7.91e-02	5.62e-02	-1.05e-01
WBYY	-6.07e-02	-0.16	2.34e-02	3.98e-02	3.56e-02	-4.89e-02	4.84e-01
BDGK	-2.50e-01	-0.24	8.38e-02	1.59e-01	1.70e-01	2.15e-01	-1.14e-01
FWKW	-1.36e-01	0.09	2.16e-01	2.50e-01	-6.66e-02	1.30e-01	-2.61e-04
PFKF	-2.86e-01	-0.04	1.67e-02	-8.24e-02	-2.34e-01	-3.03e-01	2.29e-02
ZKOY	1.38e-02	0.19	5.45e-02	3.94e-02	-4.88e-02	-5.05e-02	-6.12e-02
XBUG	-1.79e-01	0.08	7.75e-02	-9.10e-03	-7.14e-02	-5.47e-02	-2.16e-01
OMXX	-1.09e-01	0.03	1.90e-01	1.89e-01	9.65e-02	-1.80e-01	-1.14e-01
IXKB	2.32e-01	0.07	2.96e-03	-1.99e-02	-1.06e-01	1.08e-01	2.09e-01
YKJO	8.94e-02	0.09	1.60e-02	-1.38e-01	-2.08e-01	-3.66e-02	-2.50e-01
GQUZ	5.26e-02	-0.06	1.80e-01	-2.16e-01	-3.59e-02	-1.42e-01	2.11e-01
VKRW	5.48e-02	-0.19	3.44e-01	9.22e-02	6.27e-02	-2.90e-02	-9.54e-02
DDTT	-1.64e-01	-0.24	-1.87e-01	-1.37e-01	-1.85e-01	5.88e-02	-1.26e-01
QXYM	5.96e-02	0.18	3.95e-02	1.22e-02	6.37e-02	2.71e-01	-3.94e-02
MHEC	1.64e-01	-0.03	1.39e-01	-4.26e-02	-2.68e-01	5.24e-01	-1.25e-01
TUXT	1.41e-01	-0.07	7.33e-02	-1.68e-01	3.60e-01	-1.21e-01	-1.80e-01
UZSD	-2.94e-02	-0.11	1.15e-01	9.94e-02	3.40e-01	1.85e-01	1.94e-01
AOQU	5.67e-03	-0.20	-1.61e-01	1.45e-01	1.88e-01	-5.05e-02	-2.01e-01
RJWA	7.80e-02	-0.21	-1.02e-01	2.08e-01	-1.10e-01	6.99e-02	-3.01e-02
VTQI	-8.94e-02	0.07	-5.29e-02	1.10e-01	-2.36e-01	-6.75e-02	9.05e-02
ROQE	2.74e-02	0.05	1.05e-01	3.83e-01	-7.80e-02	2.74e-02	1.28e-01
ZMHO	1.31e-01	-0.08	-1.51e-02	-3.32e-01	-6.93e-02	1.37e-01	-1.60e-02
RTDN	2.62e-01	0.23	-3.68e-02	-8.40e-02	-8.62e-02	-1.97e-01	-1.25e-01
MJXP	-7.70e-03	-0.13	3.51e-01	2.27e-02	-2.70e-01	8.93e-02	4.41e-02
XYKI	5.89e-02	-0.04	-1.36e-02	4.25e-02	-7.70e-02	-2.78e-01	7.24e-02

In order to do this we extract the components coefficients, or loadings. This helps us understand how much each feature influences each principal component [22].

[22] https://stackoverflow.com/questions/47370795/pca-on-sklearn-how-to-interpret-pca-components

In [101…

```
def get_n_features(loadings_df, n):
    n_features = {}
    for col in loadings_df.columns:
        largest_n = loadings_df[col].abs().nlargest(n)
        n_features[col] = list(zip(largest_n.index, largest_n.values))
    return n_features

n_features = get_n_features(df_loadings, 10)
for pc, features in n_features.items():
    print(f"{pc}:")
    for feature, loading in features:
        print(f"  {feature}: {loading:.4f}")
```

```
PC1:
  GXSO: 0.3976
  PTAX: 0.3212
  KVJI: 0.2956
  PFKF: 0.2862
  RTDN: 0.2620
  BDGK: 0.2497
  IXKB: 0.2324
  PMLC: 0.2310
  RNOZ: 0.2241
  XBUG: 0.1789
PC2:
  RXUY: 0.3533
  PMLC: 0.3270
  PSSJ: 0.2746
  RLCO: 0.2729
  BDGK: 0.2440
  DDTT: 0.2412
  RTDN: 0.2252
  RJWA: 0.2122
  AOQU: 0.2030
  WIBD: 0.1886
PC3:
  GWEE: 0.3723
  MJXP: 0.3505
  VKRW: 0.3442
  CSTT: 0.3271
  RNOZ: 0.2332
  FWKW: 0.2158
  OMXX: 0.1896
  DDTT: 0.1870
  GQUZ: 0.1804
  PSSJ: 0.1764
PC4:
  ROQE: 0.3825
  ZMHO: 0.3318
  WIBD: 0.3151
  JIEK: 0.3118
  FWKW: 0.2504
  GQUZ: 0.2165
  RJWA: 0.2079
  QRWO: 0.1922
  NANX: 0.1907
  OMXX: 0.1894
PC5:
  TUXT: 0.3601
  UZSD: 0.3400
  RLCO: 0.3130
  MJXP: 0.2701
  MHEC: 0.2679
  ZYTI: 0.2436
  VTQI: 0.2360
  PFKF: 0.2342
  YKJO: 0.2079
  AOQU: 0.1879
PC6:
  MHEC: 0.5237
  RNOZ: 0.3228
  PFKF: 0.3031
  XYKI: 0.2778
  QXYM: 0.2708
  BDGK: 0.2152
  RTDN: 0.1966
  CSTT: 0.1931
  UZSD: 0.1852
  OMXX: 0.1795
PC7:
  WBYY: 0.4840
  NANX: 0.2697
  WIBD: 0.2686
  GWEE: 0.2527
  YKJO: 0.2503
  JIEK: 0.2350
  XBUG: 0.2161
  GQUZ: 0.2106
  IXKB: 0.2088
  AOQU: 0.2014
```

Using the scrollable element we see that the largest coefficients are for `PC6` with `0.5237` and `PC7` with `0.4840` then there are a number of others i the 0.3 and 0.2 ranges. What we care about, however, is not so much what features make up each principal component but rather what features distinguish each component from one another. For example, `GXSO` is the top coefficient for `PC1` with 0.3976 and no other principal components have it in their top 10 so it is a very distinguishing feature, whereas `GWEE` is less distinguishing as although it is also the top component for `PC3` with `0.3723` it is also the fourth coefficient for `PC7` with `0.2527`.

In [102…

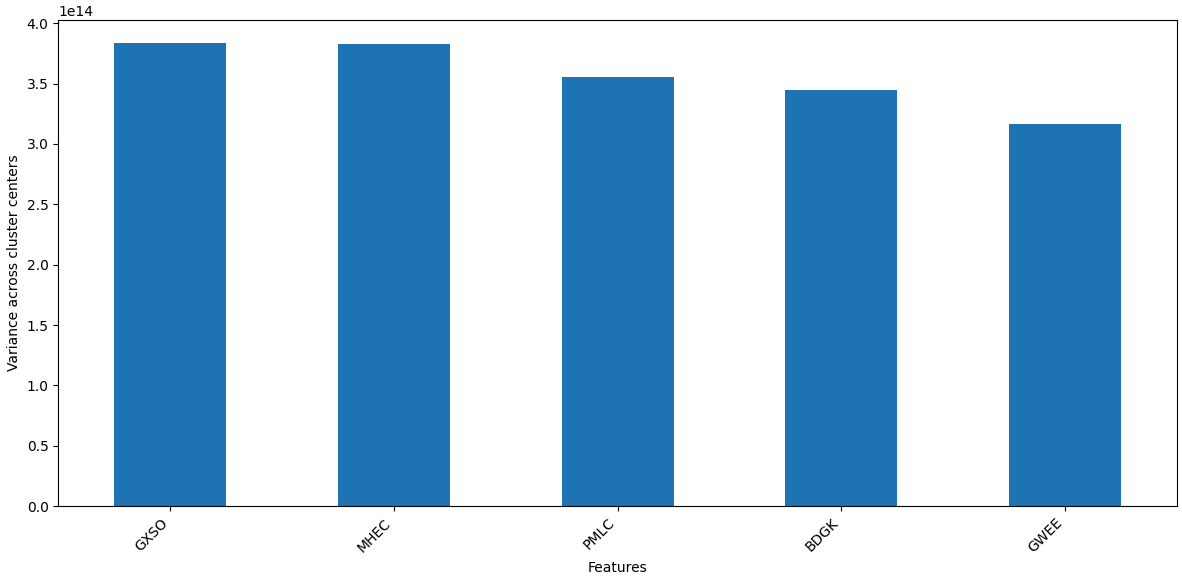
```python
df_cluster_centres = pd.DataFrame(feature_centres, columns=df_x.columns)

def get_distinguishing_features(df_cluster_centres, n):
    distinguishing_features = {}
    for i in range(len(df_cluster_centres)):
        diff_from_mean = df_cluster_centres.iloc[i] - df_cluster_centres.mean()
        top_n = diff_from_mean.abs().nlargest(n)
        relative_importance = top_n / top_n.sum()
        distinguishing_features[f'Cluster {i}'] = list(zip(top_n.index, relative_importance))
    return distinguishing_features

distinguishing_features = get_distinguishing_features(df_cluster_centres, 5)

for cluster, features in distinguishing_features.items():
    print(f"{cluster}:")
    for feature, importance in features:
        print(f"  {feature}: {importance:.4f}")
```

```
Cluster 0:
  KVJI: 0.2255
  FWKW: 0.2148
  RXUY: 0.1940
  PTAX: 0.1854
  PSSJ: 0.1803
Cluster 1:
  VKRW: 0.2182
  PMLC: 0.2071
  QXYM: 0.2036
  GWEE: 0.1860
  PFKF: 0.1851
Cluster 2:
  RTDN: 0.2341
  BDGK: 0.2279
  GXSO: 0.1838
  ZMHO: 0.1808
  GWEE: 0.1734
Cluster 3:
  RXUY: 0.2172
  RLCO: 0.2126
  JIEK: 0.1909
  BDGK: 0.1908
  RTDN: 0.1885
Cluster 4:
  MJXP: 0.2231
  RLCO: 0.2104
  ZYTI: 0.1934
  TUXT: 0.1867
  CSTT: 0.1864
Cluster 5:
  WIBD: 0.2358
  RJWA: 0.2281
  MHEC: 0.1880
  GXSO: 0.1803
  AOQU: 0.1678
Cluster 6:
  UZSD: 0.2287
  ROQE: 0.2048
  WBYY: 0.1988
  VKRW: 0.1852
  DDTT: 0.1825
Cluster 7:
  PFKF: 0.2281
  GWEE: 0.2165
  DDTT: 0.1927
  PSSJ: 0.1815
  WBYY: 0.1812
```

In [103…

```python
df_cluster_centres = pd.DataFrame(feature_centres, columns=df_x.columns)
top_features = df_cluster_centres.var().nlargest(5)
for feature, variance in top_features.items():
    print(f"{feature}: {variance:.4e}")
import matplotlib.pyplot as plt
plt.figure(figsize=(12, 6))
top_features.plot(kind='bar')
plt.xlabel('Features')
plt.ylabel('Variance across cluster centers')
plt.xticks(rotation=45, ha='right')
plt.tight_layout()
plt.show()
```

```
GXSO: 3.8334e+14
MHEC: 3.8279e+14
PMLC: 3.5513e+14
BDGK: 3.4490e+14
GWEE: 3.1649e+14
```

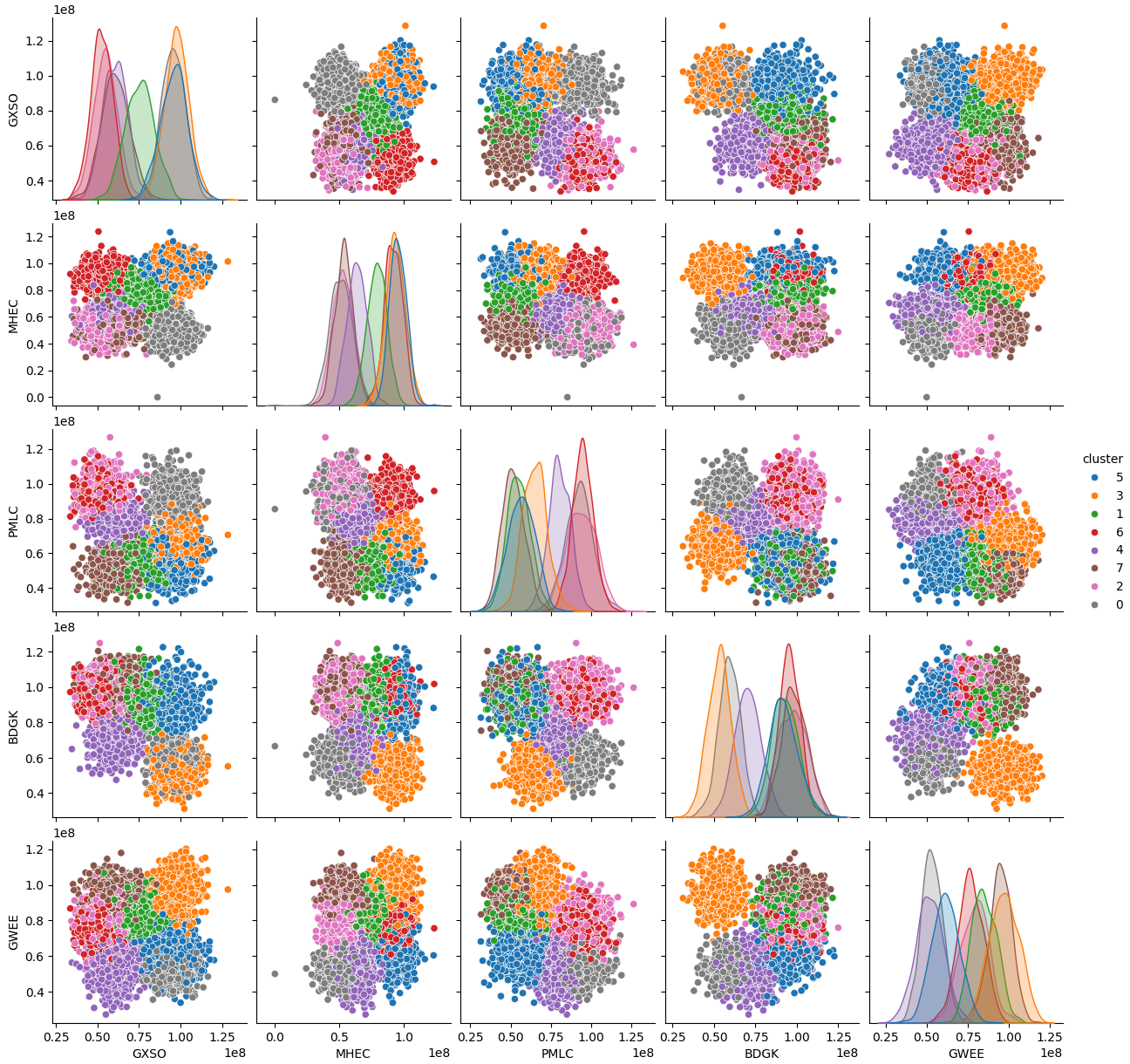


By comparing the variances for each feature of eight cluster centres we can find the top ten features that separate the clusters where we see `GXSO` as we found when comparing the features is number one and `GWEE` is lower at number five.

In [104…

```python
kmeans = KMeans(n_clusters=8)
df_top_features = df_no_outliers[top_features.keys()]
cluster_feature_predict = kmeans.fit_predict(df_top_features)
df_feature_predict = df_top_features.copy()
df_feature_predict["cluster"] = cluster_feature_predict.astype(str)
sns.pairplot(df_feature_predict, hue='cluster')
```

Out[104… <seaborn.axisgrid.PairGrid at 0x290c65f6ad0>

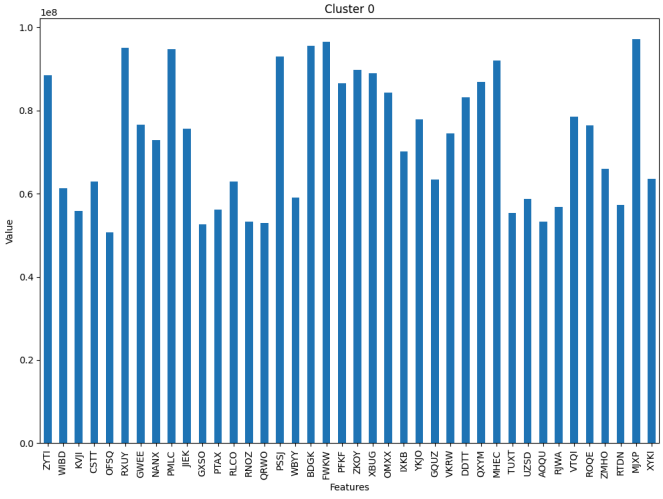


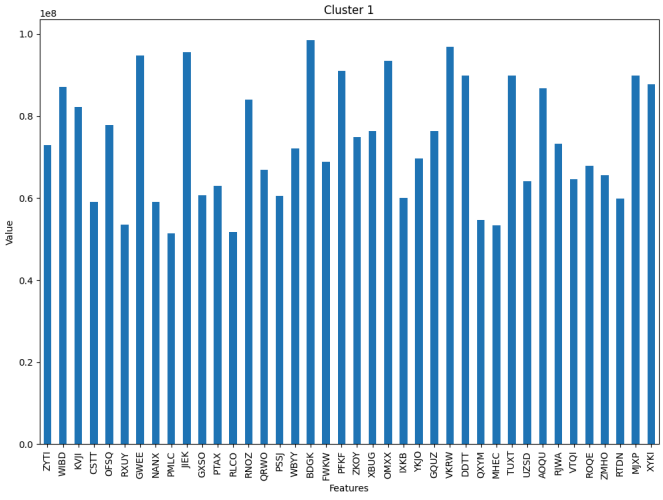
Repeating the process with the top 10 features we find a decent level of separation of the clusters but unsurprisingly it is not as clear as when we use the PCA components for example `GXSO` and `MHEC` looks like four clusters squished together.

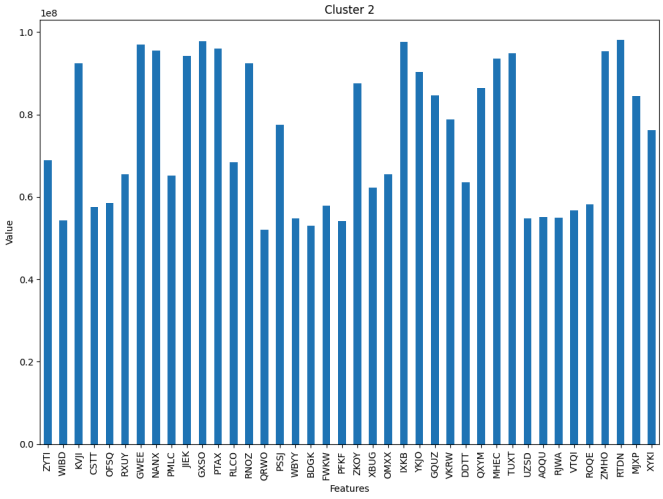
[23] https://towardsdatascience.com/interpretable-k-means-clusters-feature-importances-7e516eeb8d3c

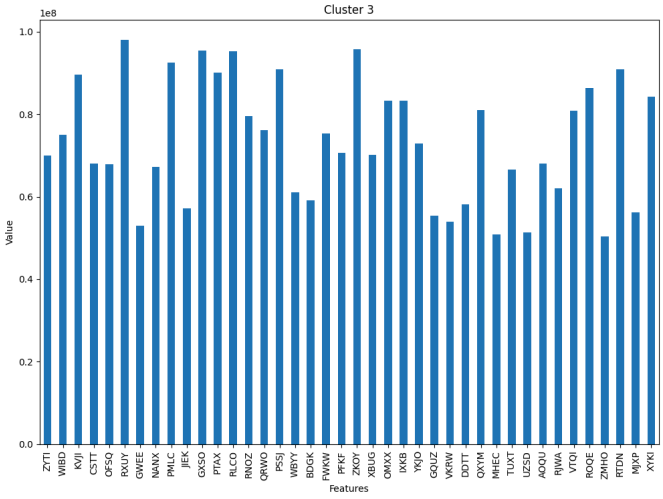
In [105…

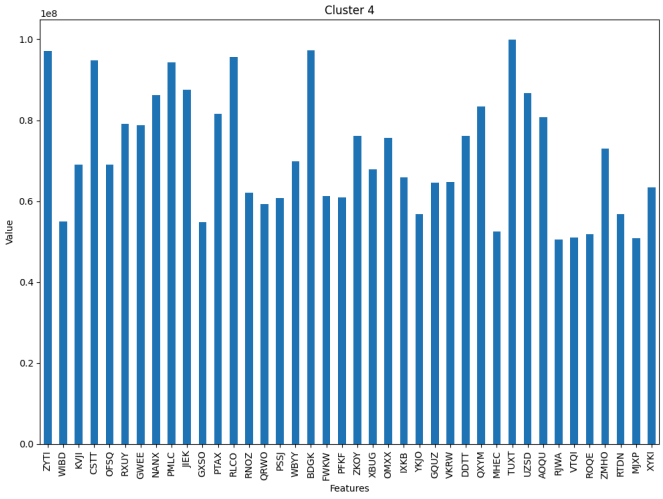
```
fig, axes = plt.subplots(4, 2, figsize=(20, 30))
for i, ax in enumerate(axes.flatten()):
    df_cluster_centres.iloc[i].plot(kind='bar', ax=ax)
    ax.set_title(f'Cluster {i}')
    ax.set_xlabel('Features')
    ax.set_ylabel('Value')
    ax.tick_params(axis='x', rotation=90)

plt.tight_layout()
plt.show()
```

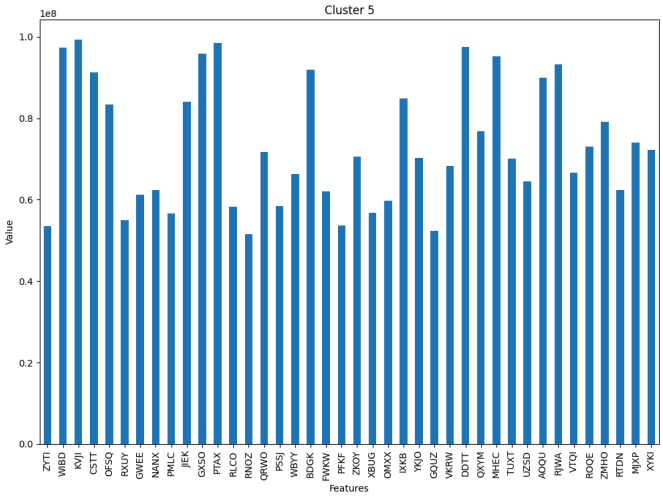

Cluster 0
1e8
Value
Features
ZYTI WIBD KVJI CSTT OFSQ RXUY GWEE NANX PMLC JIEK GXSO PTAX RLCO RNOZ QRWO PSSJ WBYY BDGK FWKW PFKF ZKOY XBUG OMXX IKKB YKJO GQUZ VKRW DDTT QXYM MHEC TUXT UZSD AOQU RJWA VTQI ROQE ZMHO RTDN MJXP XYKI

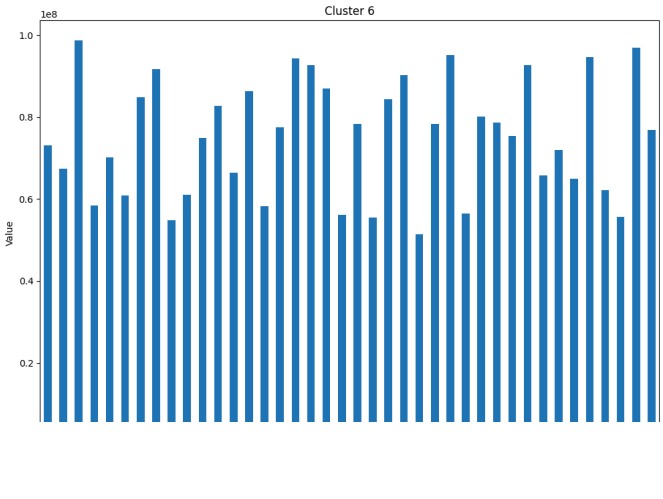

Cluster 1
1e8
Value
Features
ZYTI WIBD KVJI CSTT OFSQ RXUY GWEE NANX PMLC JIEK GXSO PTAX RLCO RNOZ QRWO PSSJ WBYY BDGK FWKW PFKF ZKOY XBUG OMXX IKKB YKJO GQUZ VKRW DDTT QXYM MHEC TUXT UZSD AOQU RJWA VTQI ROQE ZMHO RTDN MJXP XYKI

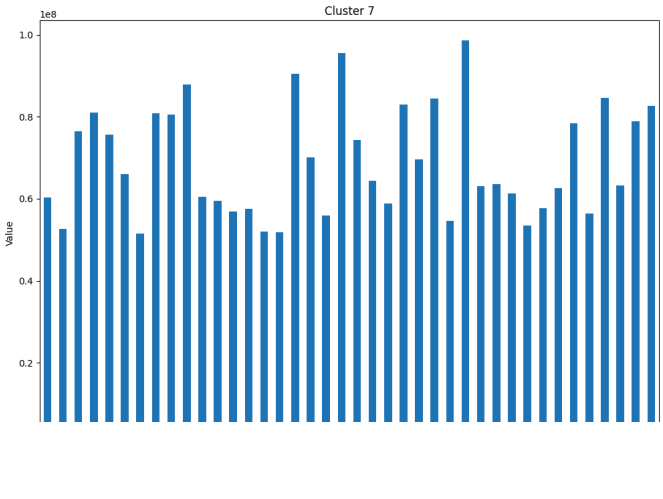

Cluster 2
1e8
Value
Features
ZYTI WIBD KVJI CSTT OFSQ RXUY GWEE NANX PMLC JIEK GXSO PTAX RLCO RNOZ QRWO PSSJ WBYY BDGK FWKW PFKF ZKOY XBUG OMXX IKKB YKJO GQUZ VKRW DDTT QXYM MHEC TUXT UZSD AOQU RJWA VTQI ROQE ZMHO RTDN MJXP XYKI

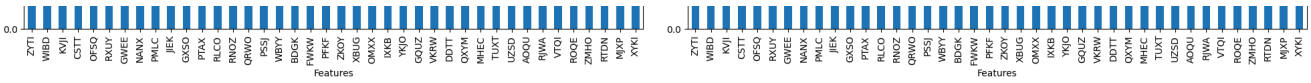

Cluster 3
1e8
Value
Features
ZYTI WIBD KVJI CSTT OFSQ RXUY GWEE NANX PMLC JIEK GXSO PTAX RLCO RNOZ QRWO PSSJ WBYY BDGK FWKW PFKF ZKOY XBUG OMXX IKKB YKJO GQUZ VKRW DDTT QXYM MHEC TUXT UZSD AOQU RJWA VTQI ROQE ZMHO RTDN MJXP XYKI


Cluster 4
1e8
Value
Features
ZYTI WIBD KVJI CSTT OFSQ RXUY GWEE NANX PMLC JIEK GXSO PTAX RLCO RNOZ QRWO PSSJ WBYY BDGK FWKW PFKF ZKOY XBUG OMXX IKKB YKJO GQUZ VKRW DDTT QXYM MHEC TUXT UZSD AOQU RJWA VTQI ROQE ZMHO RTDN MJXP XYKI


Cluster 5
1e8
Value
Features
ZYTI WIBD KVJI CSTT OFSQ RXUY GWEE NANX PMLC JIEK GXSO PTAX RLCO RNOZ QRWO PSSJ WBYY BDGK FWKW PFKF ZKOY XBUG OMXX IKKB YKJO GQUZ VKRW DDTT QXYM MHEC TUXT UZSD AOQU RJWA VTQI ROQE ZMHO RTDN MJXP XYKI


Cluster 6
1e8
Value


Cluster 7
1e8
Value



Therefore, we instead plot the values across all 40 feature variables of each of the eight cluster centres. Of course, to human eyes this is no where near as clear as the sns pairplot of PCA clusters but it does allow us to see the how the different clusters vary in terms of the original features. For example, cluster six and seven both have relatively large `WIBD` values although we can see they are of different scales with roughly 9e7 vs 1e8 for each cluster but then we can see that `PFKF` is relatively low for cluster 7 but more in line with the others for cluster 6.

4. What is the relationship between the input variables (first 40 features) and the 5 response variables?

In [106…
```
from sklearn.model_selection import train_test_split
X = df_no_outliers.iloc[:, :40]
Y = df_no_outliers.iloc[:, 40:]
print(X.shape, Y.shape)
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.2)
print(X_train.shape, X_test.shape, Y_train.shape, Y_test.shape)
```

```
(4976, 40) (4976, 5)
(3980, 40) (996, 40) (3980, 5) (996, 5)
```

We take our original dataframe and divide into the input variables `X` and the response variables `Y`, we can see the shapes have 40 and 5 columns respectively. We then split that data into train and test samples, this allows us to try different models and see how they perform on data they have not be trained on, namely the test split. The downside of this approach is we lose some valuable data to train on so typically we would choose an 80%-20% split in order to keep the bulk of the data for training as we have done here.

In [107…
```
from sklearn.multioutput import MultiOutputRegressor
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error

linear = MultiOutputRegressor(LinearRegression())
linear.fit(X_train, Y_train)
Y_train_predict = linear.predict(X_train)
Y_test_predict = linear.predict(X_test)

rmse_train = np.sqrt(mean_squared_error(Y_train, Y_train_predict))
rmse_test = np.sqrt(mean_squared_error(Y_test, Y_test_predict))

print(f"RMSE: Train = {rmse_train:.2f}, Test = {rmse_test:.2f}")
```

```
RMSE: Train = 856378361.37, Test = 821665532.05
```

The simplest approach is to do linear regression where we take each of the 40 explanatory variables and try to predict what the y value will be. The algorithm involves minimizing the difference between the predictions and the actual y variables. Of course, if we miss we want to treat the misses above and the misses below the same so sometimes the absolute difference is used. However, ordinary least squares (OLS) [24] instead squares the differences which still treats misses above and below the same but puts more weight on really bad misses because of the squaring effect where $\hat{y}_i$ is the ith prediction.

$$\argmin \sum (y_i - \hat{y}_i)^2$$

In our case, we actually have five response variables to predict and so we effectively need to do the linear regression five times.

We can then assess the quality of our model by calculating the root mean squared error.

$$RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}i)^2}$$

This is calculated, as its name suggests, as the average square-rooted error, but this is just the sum of the squared residuals (SSR) that OLS is trying to minimize above square-rooted and divided by a constant `n`. Thus OLS is guaranteed to minimize RMSE for linear models although with some important assumptions which we will consider later [24].

What we find is the RMSE for train is 856 million and for test 821 million. Although, RMSE is in the units of the original data this can be hare to interpret.

[24] https://en.wikipedia.org/wiki/Ordinary_least_squares

In [108…
```
from sklearn.metrics import r2_score

r2_train = r2_score(Y_train, Y_train_predict)
r2_test = r2_score(Y_test, Y_test_predict)

print(f"R^2: Train = {r2_train:.2f}, Test = {r2_test:.2f}")
```

```
R^2: Train = 0.95, Test = 0.96
```

R^2, also known as the coefficient of determination [25] is a metric between 0 and 1 where 1 indicates the model fits the data perfectly and 0 not at all. Mathematically it is calculated as the ratio of the sum of the squares residuals divided by the total sum of the squares. Intuitively, it represents how much of the variance cannot be explained over all variance, then by subtracting this ratio from 1 we get the amount of variance that can be explained by the independent variables divided by the total variance. Mathematically

$$R^2 = 1 - \frac{\sum (y_i - \hat{y}_i)^2}{\sum (y_i - \bar{y})^2}$$

where again $\hat{y}_i$ is the prediction and $\bar{y}$ is the average.

Our extremely high R^2 of 0.95 and interestingly even higher 0.96 on the test suggests an extremely good fit for the model. Usually, one would expect a lower R^2 on the test data but this could be just due to the random sampling of the train-test split.

It is worth noting however that R^2 does suffer from inflation where it is at least weakly increasing in the number of regressors which, with 40 x variables could be why our R^2 is so high [25].

[25] https://en.wikipedia.org/wiki/Coefficient_of_determination

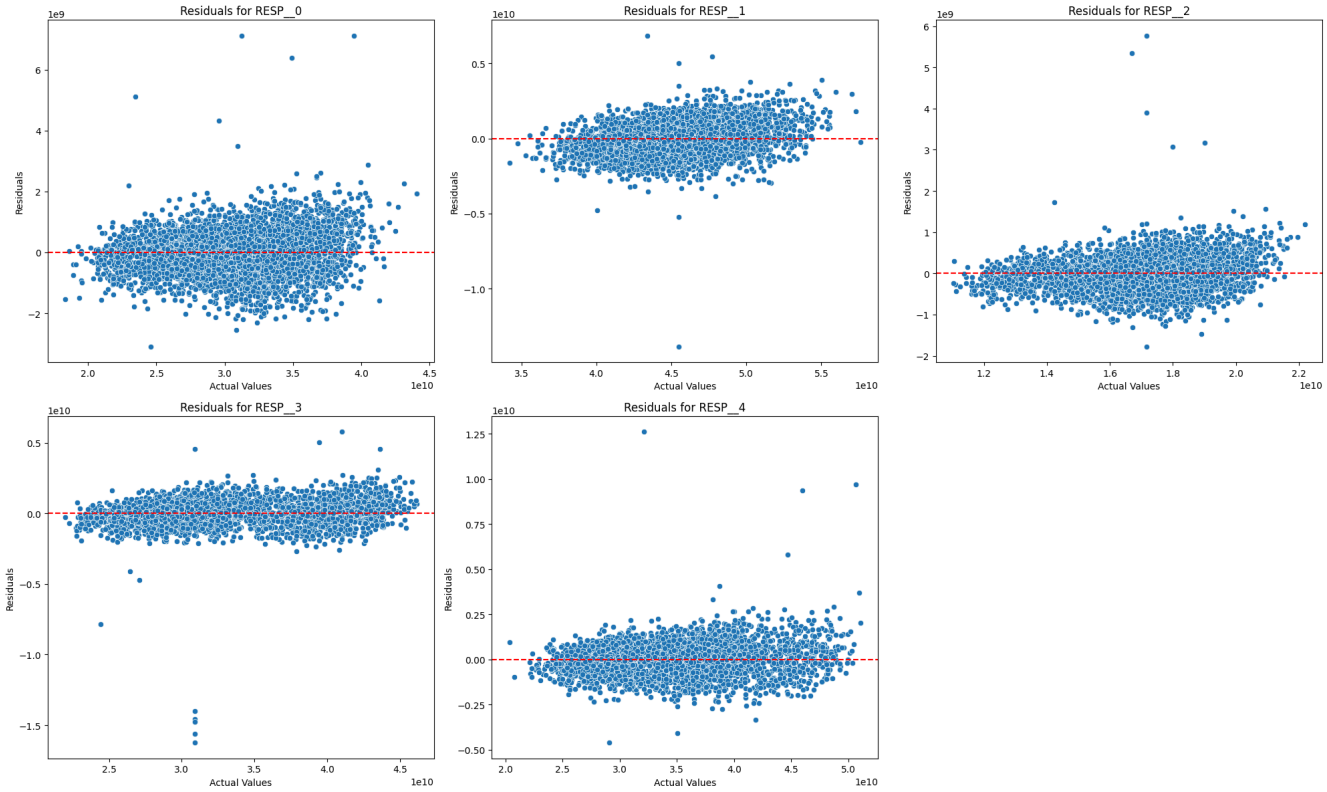
In [109…
```
def adjusted_r2(y_true, y_pred, X):
    r2 = r2_score(y_true, y_pred)
    n = len(y_true)
    p = X.shape[1]
    adjusted_r2 = 1 - (1 - r2) * (n - 1) / (n - p - 1)
    return adjusted_r2

ar2_train = adjusted_r2(Y_train, Y_train_predict, X_train)
ar2_test = adjusted_r2(Y_test, Y_test_predict, X_test)

print(f"Adjusted R^2: Train = {ar2_train:.2f}, Test = {ar2_test:.2f}")
```

```
Adjusted R^2: Train = 0.95, Test = 0.95
```

To account for this we calculate the adjusted R^2 which, in a form of regularisation, punishes the model for using a larger number of explanatory variables. Despite this, we find that actually the R^2 is still a very strong 0.95 for both the train and test splits.

In [110…

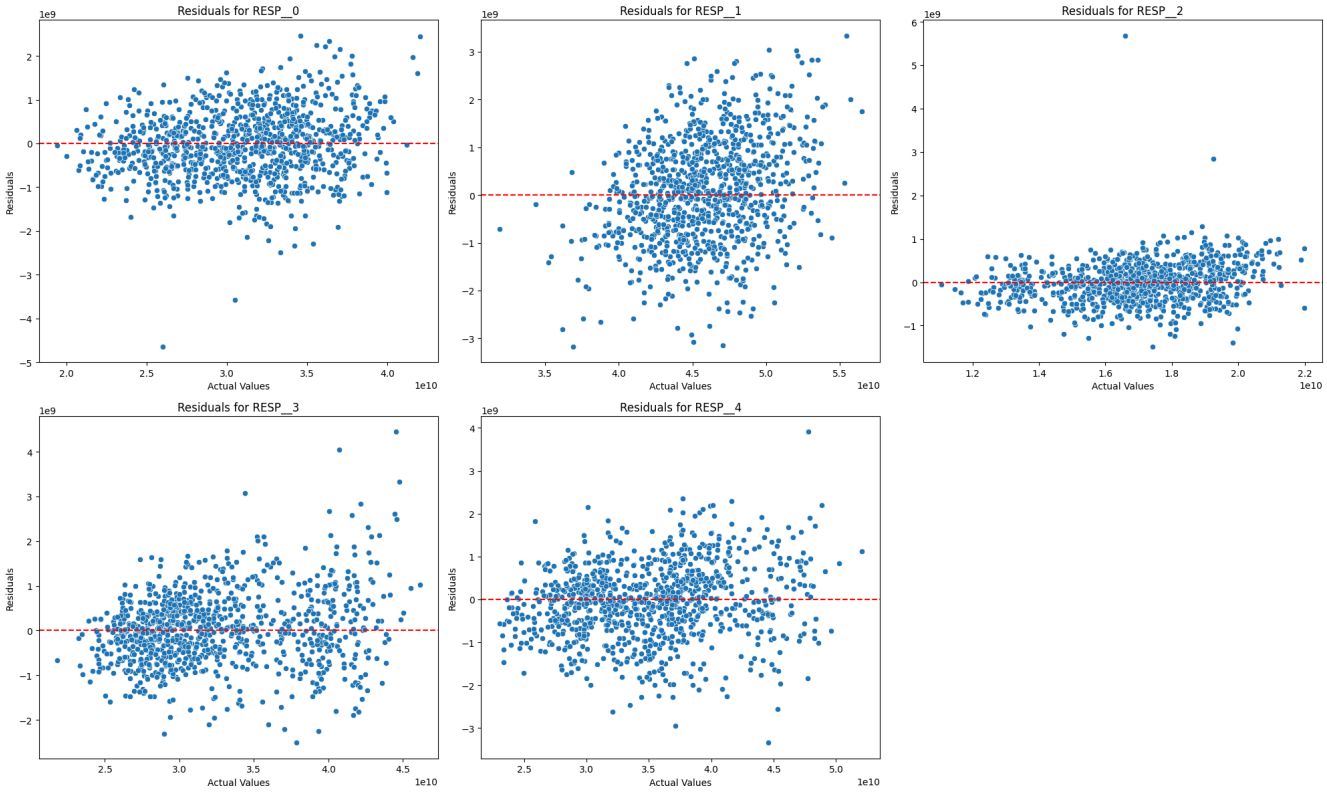
```
Y_train_residuals = Y_train - Y_train_predict
fig, axes = plt.subplots(2, 3, figsize=(20, 12))
axes = axes.flatten()

for i, col in enumerate(Y_train.columns):
    sns.scatterplot(x=Y_train[col], y=Y_train_residuals[col], ax=axes[i])
    axes[i].set_title(f'Residuals for {col}')
    axes[i].set_xlabel('Actual Values')
    axes[i].set_ylabel('Residuals')
    axes[i].axhline(y=0, color='r', linestyle='--')
fig.delaxes(axes[5])
plt.tight_layout()
plt.show()
```



If the data is linear then you would expect the residuals to be a random scatter around the horizontal line at 0. This is exactly what we see in the training data with a few outlier exceptions such as a few large negative residuals for `RESP__3` and large positive residuals for `RESP__2`.

In [111…

```
Y_test_residuals = Y_test - Y_test_predict
fig, axes = plt.subplots(2, 3, figsize=(20, 12))
axes = axes.flatten()
for i, col in enumerate(Y_test.columns):
    sns.scatterplot(x=Y_test[col], y=Y_test_residuals[col], ax=axes[i])
    axes[i].set_title(f'Residuals for {col}')
    axes[i].set_xlabel('Actual Values')
    axes[i].set_ylabel('Residuals')
    axes[i].axhline(y=0, color='r', linestyle='--')
fig.delaxes(axes[5])
plt.tight_layout()
plt.show()
```



The test charts show a similar pattern although admittedly the residuals are not so tight to the 0 line which is what you would expect on unseen test data.

Thus, we can conclude that the relationship between the input variables and the response variables is a linear one.

This is despite the fact that we saw clear clustering in the explanatory variables perhaps the high-dimensional space is allowing for both to occur, or they are happening at different scales, clustering at a micro level and linear at a macro level. It is also possible that there are some interaction effects where despite being linear in response they still have clusterings in the interaction.

In [112…
```
from sklearn.preprocessing import PolynomialFeatures
from sklearn.pipeline import make_pipeline

poly = make_pipeline(PolynomialFeatures(degree=2), MultiOutputRegressor(LinearRegression()))
poly.fit(X_train, Y_train)

Y_train_predict_poly = poly.predict(X_train)
Y_test_predict_poly = poly.predict(X_test)

r2_train_poly = r2_score(Y_train, Y_train_predict_poly)
r2_test_poly = r2_score(Y_test, Y_test_predict_poly)

print(f"Polynomial R^2: Train = {r2_train_poly:.2f}, Test = {r2_test_poly:.2f}")
```

```
Polynomial R^2: Train = 0.97, Test = 0.94
```

To explore this possibility of other interactions we compare our linear model to a polynomial regression. Polynomial regressions capture non-linear effects but counter-intuitively does so despite still using linear regression [26]. It works by transforming the exploanatory variables by taking powers and interaction terms giving us non-linearities which we can then use OLS on.

In the above, we see a similar performance with 0.94 on the test data and slightly better performance on the training data of 0.97 compared to our strictly linear model. However, as the two models perform roughly the same we conclude that the relationship is linear opting for the simpler model.

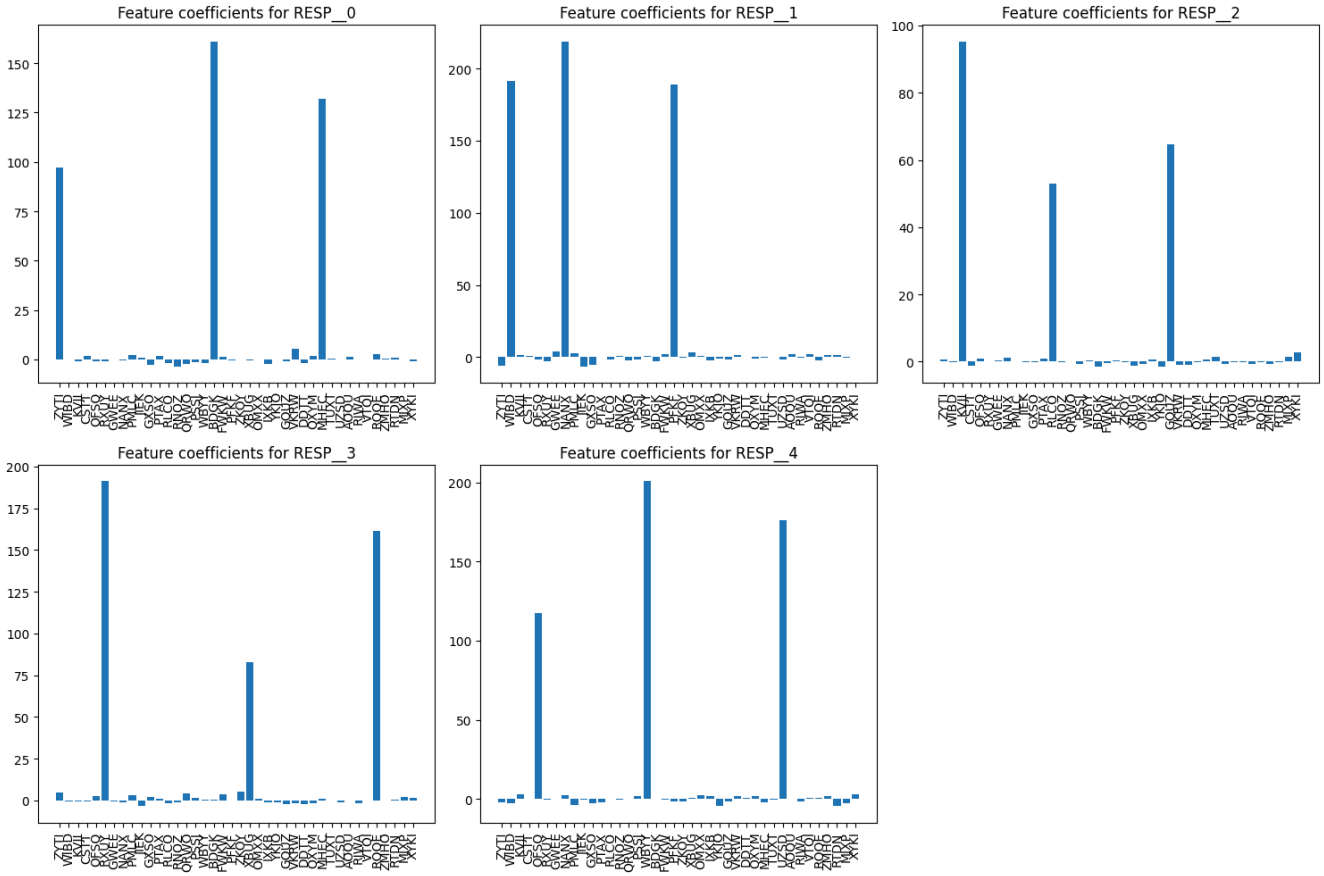
[26] https://en.wikipedia.org/wiki/Polynomial_regression

5. Are there specific features which 'drive' specific responses?

a) If so, how?

b) What are the important features and what is their relative impact?

In [116…
```
plt.figure(figsize=(15, 10))
for i, response in enumerate(Y.columns):
    coefficients = linear.estimators_[i].coef_
    plt.subplot(2, 3, i+1)
    plt.bar(X.columns, coefficients)
    plt.title(f'Feature coefficients for {response}')
    plt.xticks(rotation=90)
plt.tight_layout()
plt.show()

for i, response in enumerate(Y.columns):
    coefficients = linear.estimators_[i].coef_
    feature_importance = pd.DataFrame({'feature': X.columns, 'importance': abs(coefficients)})
    feature_importance = feature_importance.sort_values('importance', ascending=False)

    print(f"\nTop 5 features for {response}:")
    print(feature_importance.head(5))
```




```
Top 5 features for RESP__0:
   feature  importance
17    BDGK      161.15
29    MHEC      131.82
0     ZYTI       97.02
26    VKRW        5.25
13    RNOZ        3.63

Top 5 features for RESP__1:
   feature  importance
7     NANX      218.88
1     WIBD      191.41
19    PFKF      188.92
9     JIEK        6.48
0     ZYTI        5.65

Top 5 features for RESP__2:
   feature  importance
2     KVJI       95.31
25    GQUZ       64.73
12    RLCO       52.87
39    XYKI        2.73
30    TUXT        1.54

Top 5 features for RESP__3:
   feature  importance
5     RXUY      191.41
35    ROQE      161.49
21    XBUG       82.92
20    ZKOY        5.38
0     ZYTI        4.65

Top 5 features for RESP__4:
   feature  importance
16    WBYY      200.83
31    UZSD      176.14
4     OFSQ      117.62
24    YKJO        4.56
37    RTDN        4.50
```

The first approach we can take is simply to look at the coefficients of each variable. What emerges is an extremely clear picture of some variables mattering a lot ane most not at all.

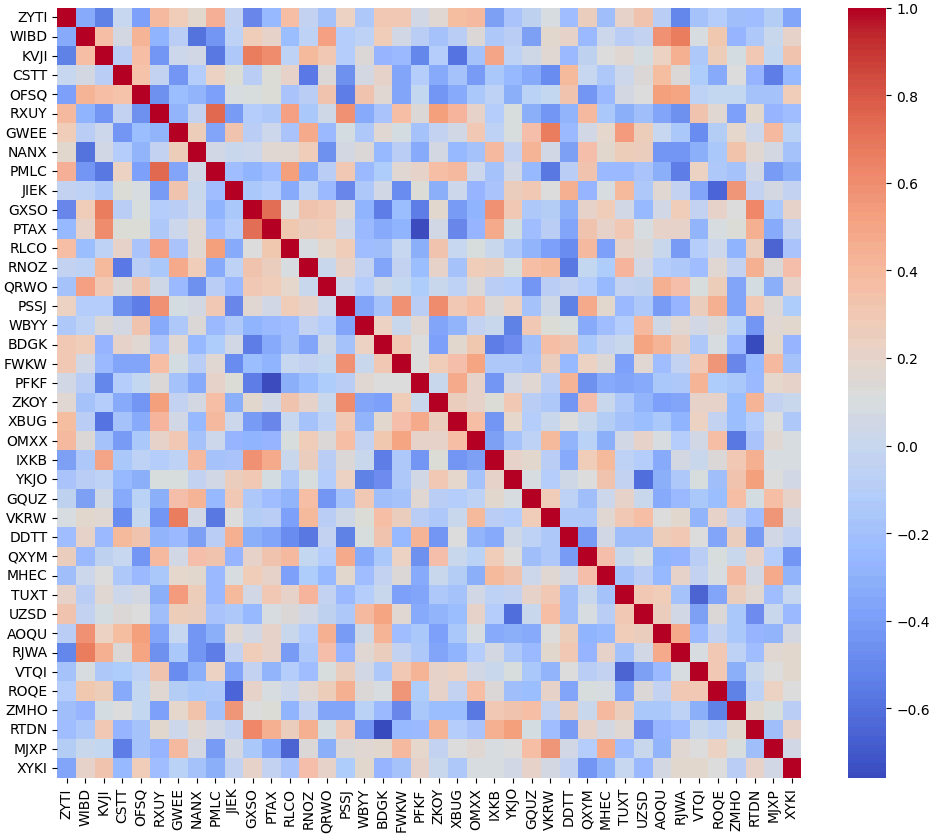
Going through each graph we find that for

- `RESP__0` : `BDGK` 161.15, `MHEC` 131.82 and `ZYTI` 131.82 versus the next highest of VKRW of 5.25
- `RESP__1` : `NANX` 218.88, `WIBD` 191.41 and `PFKF` 188.92 versus the next highest of 6.48
- `RESP__2` : `KVJI` 95.31, `GQUZ` 64.73 and `RLCO` 52.87 versus the next highest of 2.73
- `RESP__3` : `RXUY` 191.41, `ROQE` 161.49 and `XBUG` 82.92 versus the next highest of 5.38
- `RESP__4` : `WBYY` 200.83, `UZSD` 176.14 and `OFSQ` 117.62 versus the next highest of 4.56

In all cases only three variables seem to matter very much and interesting no variable seems to matter to two different response variables.

In [117…

```
corr_matrix = X.corr()
plt.figure(figsize=(12, 10))
sns.heatmap(corr_matrix, annot=False, cmap='coolwarm')
plt.show()
```



When interpreting coefficients however we should be careful and particularly wary of multicollinearity which is defined as the situation where the `X` variables are correlated with each other [27]. So far multicollinearity has not been a concern as we have only focused on the overall predictive power of the model as a whole. However, it can be problematic when we are trying to interpret individual coefficients and their meaning as we are doing here. The reason is that if `x1` and `x2` are correlated and both effect the response variable how should the model assign coefficients? A large coefficient to `x1`, and small to `x2`, or vice-versa or maybe medium sized coefficients to both? The regression has no way to know which to choose.

In order to determine whether multicollinearity is a problem we plot a heatmap where red colours indicate positive correlation and blue colours negative correlation. What we see is that along the diagonal is a strong red colour because there is, unsurprisingly perfect correlation between a variable and itself. Outside of this there are no obvious red correlations, although there are some notable darker blues indicate a strong negative correlation. For example, `BDGK` has a strong negative correlation with `RTDN` which is interesting as `BDGK` is the highest coefficient for `RESP__0`. Similarly for `PFKF` and `PTAX` where `PFKF` which is the third most significant for `RESP__1`

[27] https://en.wikipedia.org/wiki/Multicollinearity

In [118…

```
from statsmodels.stats.outliers_influence import variance_inflation_factor

df_vif = pd.DataFrame()
df_vif["feature"] = X.columns
df_vif["VIF"] = [variance_inflation_factor(X.values, i) for i in range(X.shape[1])]
print(df_vif)
```

	feature	VIF
0	ZYTI	71.65
1	WIBD	58.06
2	KVJI	114.83
3	CSTT	74.98
4	OFSQ	76.43
5	RXUY	71.27
6	GWEE	73.37
7	NANX	82.44
8	PMLC	74.90
9	JIEK	88.54
10	GXSO	81.20
11	PTAX	92.10
12	RLCO	70.60
13	RNOZ	68.59
14	QRWO	56.84
15	PSSJ	80.89
16	WBYY	60.84
17	BDGK	90.20
18	FWKW	78.42
19	PFKF	68.51
20	ZKOY	109.87
21	XBUG	67.33
22	OMXX	95.29
23	IXKB	93.14
24	YKJO	76.65
25	GQUZ	65.33
26	VKRW	79.50
27	DDTT	81.84
28	QXYM	87.60
29	MHEC	60.19
30	TUXT	79.26
31	UZSD	58.91
32	AOQU	72.78
33	RJWA	73.98
34	VTQI	70.22
35	ROQE	67.32
36	ZMHO	70.31
37	RTDN	68.10
38	MJXP	91.14
39	XYKI	91.28

Although, helpful for an initial impression ideally we would like a more quantitative approach to determine how bad our multicollinearity might be. We use the variance inflation factor (VIF) which is a ratio between the variance in the parameter estimate when fitting a full model versus the variance when fitting only a single parameter on its own [28]. What we find is that the VIFs are extremely large, notably `KVJI` with a ratio of 114.83 which means that the standard error is 114x larger then it would have been if the variable had no correlation to other variables. Typically, VIFs above 5-10 are considered large.

[28] https://en.wikipedia.org/wiki/Variance_inflation_factor

In [124…

```
from sklearn.linear_model import LassoCV

lasso_cv = MultiOutputRegressor(LassoCV(cv=5))
lasso_cv.fit(X_train, Y_train)

Y_train_predict = lasso_cv.predict(X_train)
Y_test_predict = lasso_cv.predict(X_test)

rmse_train = np.sqrt(mean_squared_error(Y_train, Y_train_predict))
rmse_test = np.sqrt(mean_squared_error(Y_test, Y_test_predict))

print(f"RMSE: Train = {rmse_train:.2f}, Test = {rmse_test:.2f}")

ar2_train = adjusted_r2(Y_train, Y_train_predict, X_train)
ar2_test = adjusted_r2(Y_test, Y_test_predict, X_test)

print(f"Adjusted R^2: Train = {ar2_train:.2f}, Test = {ar2_test:.2f}")

for i, estimator in enumerate(lasso_cv.estimators_):
    print(f"Optimal lambda for RESP__{i}: {estimator.alpha_:.4f}")

for i, estimator in enumerate(lasso_cv.estimators_):
    print(f"\nNon-zero coefficients for RESP__{i}:")
    non_zero = [(feature, coef) for feature, coef in zip(X_train.columns, estimator.coef_) if coef != 0]
    for feature, coef in non_zero:
        print(f"{feature}: {coef:.4f}")
```

```
RMSE: Train = 858979154.78, Test = 819671227.40
Adjusted R^2: Train = 0.95, Test = 0.95
Optimal lambda for RESP__0: 211072329325221.4375
Optimal lambda for RESP__1: 143528779147763.8125
Optimal lambda for RESP__2: 115274539055589.2656
Optimal lambda for RESP__3: 2317337663590 50.7500
Optimal lambda for RESP__4: 281556664206038.6250

Non-zero coefficients for RESP__0:
ZYTI: 97.5460
GXSO: -1.4158
RNOZ: -2.5652
WBYY: -1.1718
BDGK: 162.9124
FWKW: 0.3984
PFKF: -0.1909
IXKB: -0.8350
GQUZ: -0.0359
VKRW: 3.3633
QXYM: 1.0138
MHEC: 131.8632
XYKI: -1.4913

Non-zero coefficients for RESP__1:
ZYTI: -3.9190
WIBD: 189.1183
RXUY: -1.7726
GWEE: 2.1716
NANX: 215.7693
JIEK: -2.9410
GXSO: -3.5772
RLCO: -0.7370
RNOZ: 0.2055
BDGK: -0.4563
PFKF: 189.3562
XBUG: 1.4155
IXKB: -0.5115
VKRW: 0.0171
QXYM: -0.2934
MHEC: -0.5729
UZSD: -0.5481
VTQI: 0.1789
ROQE: -0.3620

Non-zero coefficients for RESP__2:
KVJI: 95.2080
NANX: 1.4413
PTAX: 0.3451
RLCO: 51.7411
RNOZ: 0.1362
WBYY: 0.1605
BDGK: -1.6324
FWKW: -0.1884
XBUG: -1.1690
IXKB: 0.4914
YKJO: -0.0921
GQUZ: 64.6574
DDTT: -0.7156
TUXT: 1.1777
XYKI: 1.4485

Non-zero coefficients for RESP__3:
ZYTI: 1.6113
RXUY: 192.6434
NANX: -1.2326
PMLC: 2.2817
JIEK: -3.3281
QRWO: 2.4221
PSSJ: 1.8637
FWKW: 3.3602
ZKOY: 3.8594
XBUG: 81.9394
OMXX: 0.0284
GQUZ: -1.0103
TUXT: -0.0756
ROQE: 161.2673
ZMHO: -0.8405

Non-zero coefficients for RESP__4:
KVJI: 0.7826
OFSQ: 114.8221
NANX: 0.6402
PMLC: -2.6259
WBYY: 202.2977
YKJO: -4.5653
VKRW: 0.4209
MHEC: -1.3507
UZSD: 176.9233
RTDN: -2.6667
```

Ideally, we would like to be able to select which variables matter based upon some domain knowledge. For example, perhaps one feature is expected to have a causal relationship with `RESP__0` etc. and another feature is not. Unfortunately, we don't even know what the response variables represent so this is not possible.

Instead, we shall try to use statistical tools. Least absolute shrinkage and selection operator (LASSO) [29] is a regression and rularization technique that tries to mitigate the multicollinearity problem. It does this by calculating the loss function as the residual sum of the squares (RSS) as does OLS but adds a penalty term

$$Loss = RSS + \lambda * \sum |\beta_i|$$

which is the sum of the absolute values of the coefficients β_i scaled by some λ regularization parameter. The effect is that the model gets penalized for having lots of variables with large coefficients and is instead incentivised to choose a smaller subset of them. Lasso works particularly well for sparse datasets where most variables have no effect, but this is already what we have seen in our OLS analysis. This is known as coefficient shrinkage and effectively selects features for us.

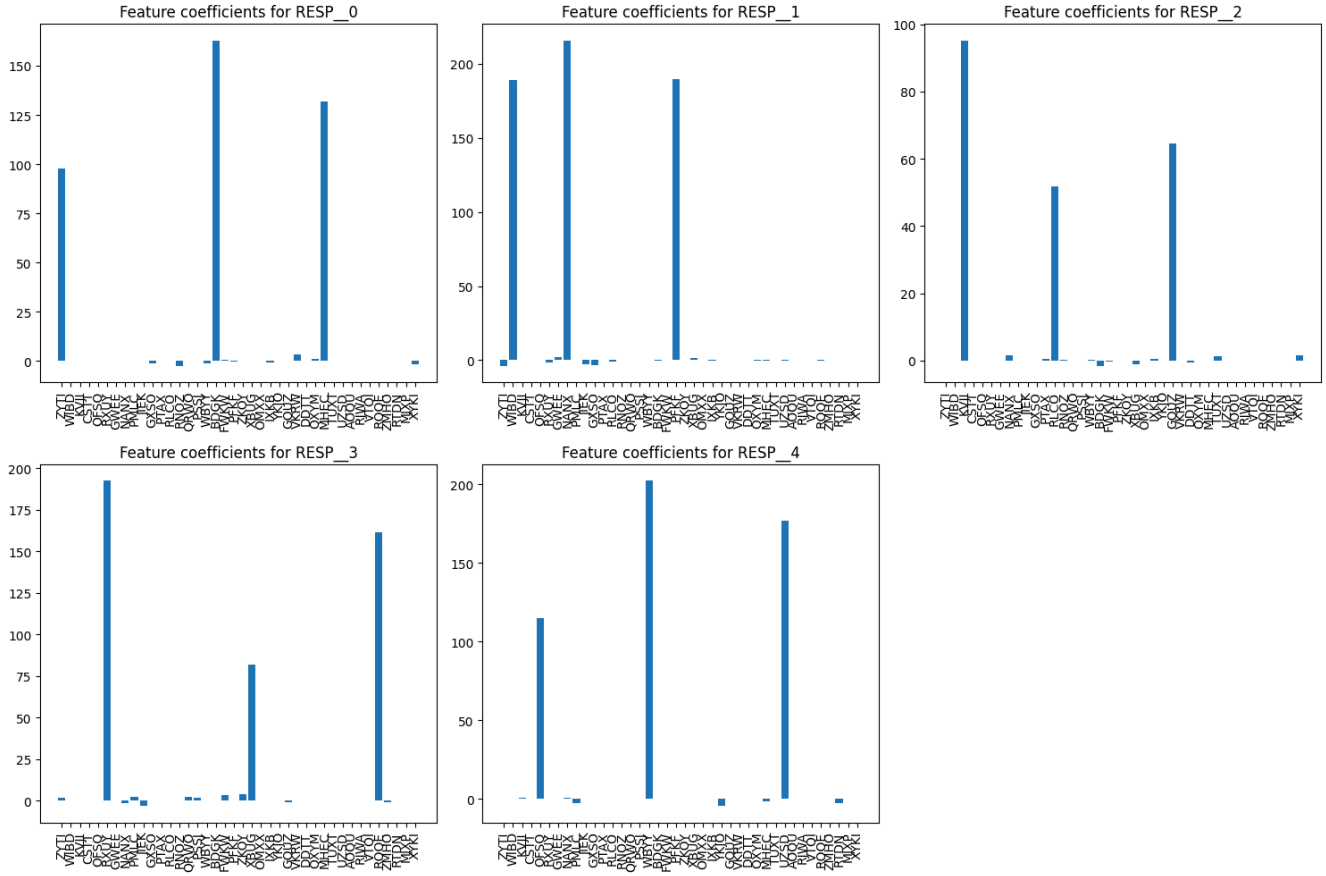
In our analysis we use LassoCV where the CV part stands for cross-validation wnich automatically sets the regularisation strength parameter λ for us by testing different values and comparing which performs best.

We follow a similar process as before and find that the `R^2` remains high at 0.95, which makes sense as this is about coefficient estimation not overall model prediction power. We then print the optimal λ that the cross-validation process found. And finally we print the non-zero coefficients where we can see that rather that 40 coefficients for `RESP__0` we only have 12 non-zero coefficients, and for `RESP__4` it is even less with just 10.

[29] https://en.wikipedia.org/wiki/Lasso_(statistics)

In [125…

```
plt.figure(figsize=(15, 10))
for i, response in enumerate(Y.columns):
    coefficients = lasso_cv.estimators_[i].coef_
    plt.subplot(2, 3, i+1)
    plt.bar(X.columns, coefficients)
    plt.title(f'Feature coefficients for {response}')
    plt.xticks(rotation=90)
plt.tight_layout()
plt.show()

for i, response in enumerate(Y.columns):
    coefficients = lasso_cv.estimators_[i].coef_
    feature_importance = pd.DataFrame({'feature': X.columns, 'importance': abs(coefficients)})
    feature_importance = feature_importance.sort_values('importance', ascending=False)

    print(f"\nTop 5 features for {response}:")
    print(feature_importance.head(5))
```




```
Top 5 features for RESP__0:
   feature  importance
17    BDGK      162.91
29    MHEC      131.86
0     ZYTI       97.55
26    VKRW        3.36
13    RNOZ        2.57

Top 5 features for RESP__1:
   feature  importance
7     NANX      215.77
19    PFKF      189.36
1     WIBD      189.12
0     ZYTI        3.92
10    GXSO        3.58

Top 5 features for RESP__2:
   feature  importance
2     KVJI       95.21
25    GQUZ       64.66
12    RLCO       51.74
17    BDGK        1.63
39    XYKI        1.45

Top 5 features for RESP__3:
   feature  importance
5     RXUY      192.64
35    ROQE      161.27
21    XBUG       81.94
20    ZKOY        3.86
18    FWKW        3.36

Top 5 features for RESP__4:
   feature  importance
16    WBYY      202.30
31    UZSD      176.92
4     OFSQ      114.82
24    YKJO        4.57
37    RTDN        2.67
```

We then do exactly the same process as before to find the key coefficients and reassuringly find the same three variables for each `RESP` driving all of the behaviour. The main difference is that many of the other variables now have zero coefficients.

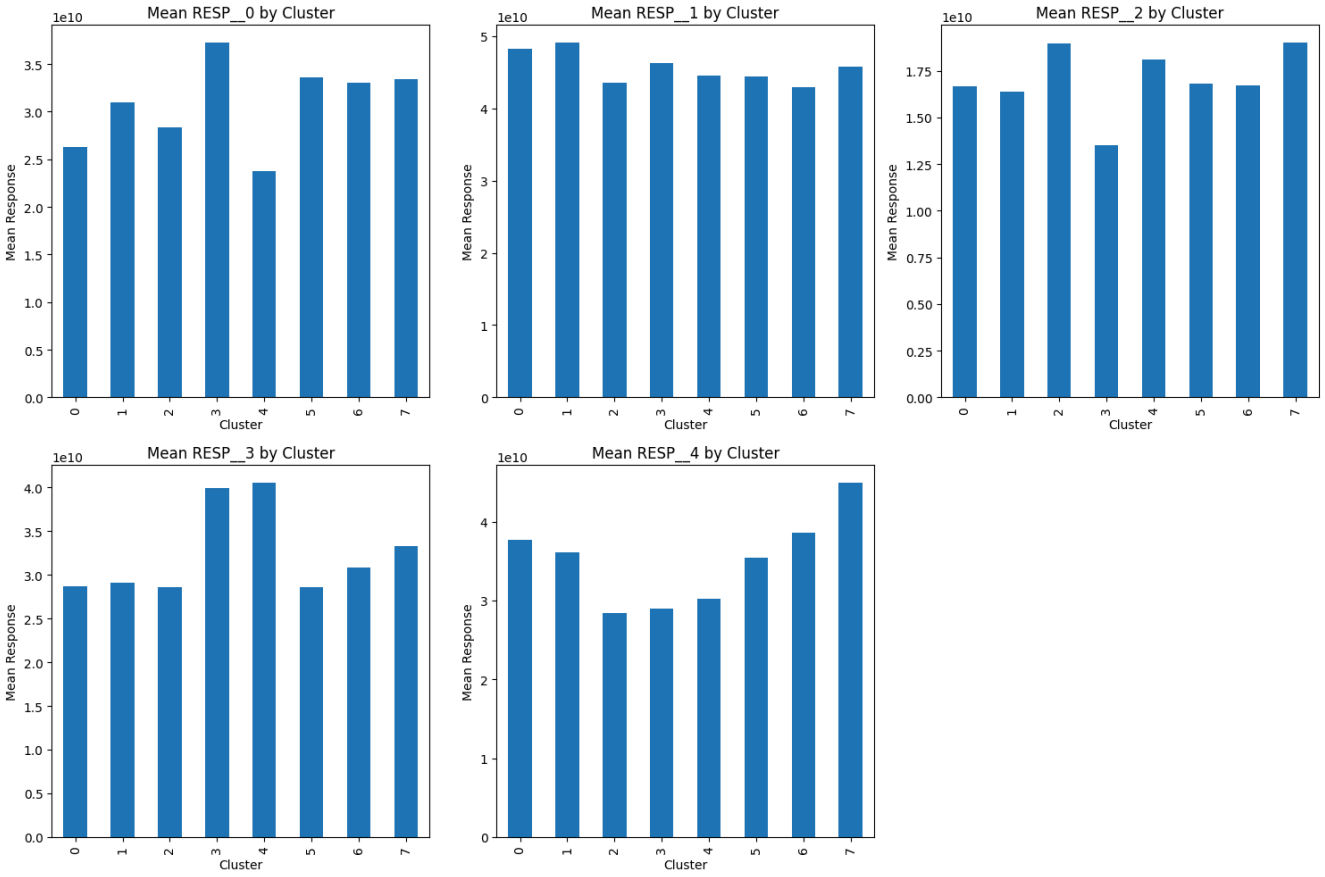
Furthermore, we have found the exact same three variables as before and almost the same coefficients

- `RESP__0` : `BDGK` 162.91 vs 161.15, `MHEC` 131.86 vs 131.82 and `ZYTI` 97.55 vs 131.82
- `RESP__1` : `NANX` 215.77 vs 218.88, `PFKF` 189.36 vs 188.92 and `WIBD` 189.12 vs 191.41
- `RESP__2` : `KVJI` 95.21 vs 95.31, `GQUZ` 64.66 vs 64.73 and `RLCO` 51.74 vs 52.87 versus
- `RESP__3` : `RXUY` 192.64 vs 191.41, `ROQE` 161.27 vs 161.49 and `XBUG` 81.94 vs 82.92
- `RESP__4` : `WBYY` 202.30 vs 200.83, `UZSD` 176.92 vs 176.14 and `OFSQ` 114.82 vs 117.62

Thus the only coefficient that has significantly changed is `ZYTI` but even then not by that much relative to the next highest.

6. If there are different types of wheat present in the dataset, then which type of wheat has the greatest impact on each response variable?

In [132…
```
pca = PCA(n_components=7)
np_pca = pca.fit_transform(df_x)
df_pca = pd.DataFrame(np_pca, columns=[f"PC{i}" for i in range(1, 8)])

kmeans = KMeans(n_clusters=8)
cluster_predict = kmeans.fit_predict(np_pca)

df_clustered = df_no_outliers.copy()
df_clustered['cluster'] = cluster_predict.astype(str)
cluster_means = df_clustered.groupby('cluster')[Y.columns].mean()

plt.figure(figsize=(15, 10))
for i, response in enumerate(Y.columns):
    plt.subplot(2, 3, i+1)
    cluster_means[response].plot(kind='bar')
    plt.title(f'Mean {response} by Cluster')
    plt.xlabel('Cluster')
    plt.ylabel('Mean Response')
plt.tight_layout()
plt.show()
```



We repeat the process we followed previously but now we see the average response for each cluster `cluster_means` however this does not seem that insightful. For example, we can see that for `RESP__3` the mean responses are higher for `PC3` and `PC4` but for `RESP_1` they are all the same.

In [139…
```
cluster_means_all = df_clustered.groupby('cluster').mean()

cluster_means_all['cluster'] = cluster_means_all.index

X_cluster = cluster_means_all.iloc[:, :40]
Y_cluster = cluster_means_all.iloc[:, 40:45]

print(X_cluster.shape, Y_cluster.shape)
```

```
(8, 40) (8, 5)
```

We using the 8 clusters we found and for each cluster calculate the `mean` value for each of the 40 features and 5 response variables and the we separate them into `X_cluster` and `Y_cluster` dataframes. We can see from their shape that they have the expected 8 clusters with 40 and 5 variables each. Note we drop the final column which is the cluster.

In [140…
```
model = LinearRegression()
model.fit(X_cluster, Y_cluster)
```

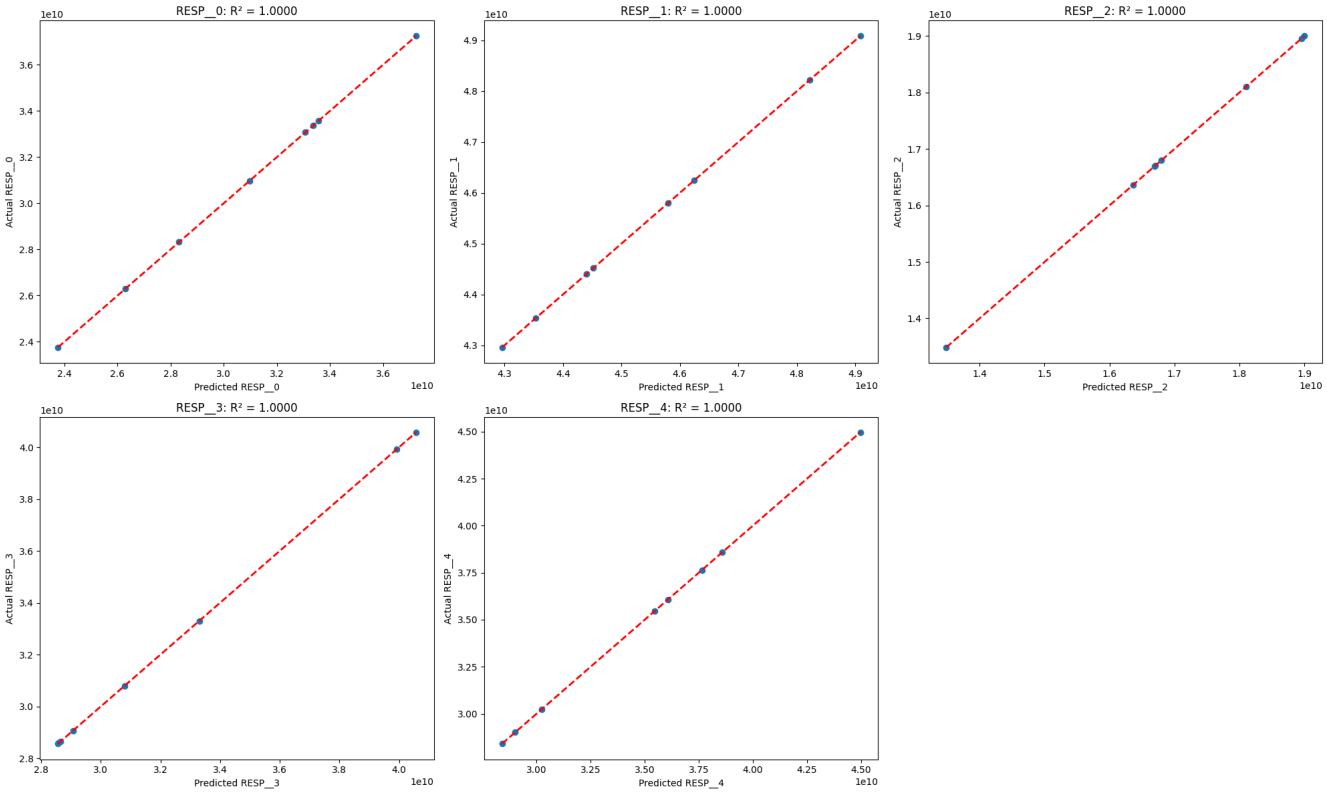
```
Y_pred = model.predict(X_cluster)

r2_scores = r2_score(Y_cluster, Y_pred, multioutput='raw_values')

for i, response in enumerate(Y_cluster.columns):
    print(f"R-squared for {response}: {r2_scores[i]:.4f}")
```

```
R-squared for RESP__0: 1.0000
R-squared for RESP__1: 1.0000
R-squared for RESP__2: 1.0000
R-squared for RESP__3: 1.0000
R-squared for RESP__4: 1.0000
```

We then do linear regression on the cluster centres effectively and find perfect `R^2` of 1 for all the response variables.

In [143…

```
fig, axes = plt.subplots(2, 3, figsize=(20, 12))
axes = axes.flatten()

for i, response in enumerate(Y_cluster.columns):
    axes[i].scatter(Y_pred[:, i], Y_cluster[response])
    axes[i].plot([Y_pred[:, i].min(), Y_pred[:, i].max()],
                 [Y_pred[:, i].min(), Y_pred[:, i].max()],
                 'r--', lw=2)
    axes[i].set_xlabel(f'Predicted {response}')
    axes[i].set_ylabel(f'Actual {response}')
    axes[i].set_title(f'{response}: R² = {r2_scores[i]:.4f}')

fig.delaxes(axes[5])
plt.tight_layout()
plt.show()
```



We plot the 8 clusters comparing the predicted responses on the x-axis and the actual responses on the y-axis and see why the cluster centres perfectly predict because they all fall on a straight-line. Therefore, if given a type of wheat if we can allocate it to the correct cluster we can accurately estimate its response variable.

Libraries Used

In the order they are used:

```
pandas
missingno
numpy
decimal
seaborn
matplotlib
sklearn
statsmodels
```

References

[1] https://pandas.pydata.org/pandas-docs/stable/user_guide/basics.html#basics-dtypes

[2] https://pandas.pydata.org/pandas-docs/stable/user_guide/io.html#type-inference

[3] https://en.wikipedia.org/wiki/Imputation_(statistics)

[4] https://en.wikipedia.org/wiki/Missing_data

[5] https://en.wikipedia.org/wiki/Kernel_density_estimation

[6] https://en.wikipedia.org/wiki/List_of_logarithmic_identities

[7] https://scikit-learn.org/stable/modules/impute.html#multivariate-feature-imputation

[8] https://www.geeksforgeeks.org/how-to-fix-runtimewarning-invalid-value-encountered-in-double_scalars/

[9] https://en.wikipedia.org/wiki/Standard_score

[10] https://en.wikipedia.org/wiki/Median_absolute_deviation

[11] https://en.wikipedia.org/wiki/Winsorizing

[12] https://en.wikipedia.org/wiki/Principal_component_analysis

[13] https://en.wikipedia.org/wiki/Orthonormal_basis

[14] https://en.wikipedia.org/wiki/Eigendecomposition_of_a_matrix

[15] https://scikit-learn.org/stable/modules/preprocessing.html

[16] https://en.wikipedia.org/wiki/Hierarchical_clustering

[17] https://en.wikipedia.org/wiki/DBSCAN

[18] https://en.wikipedia.org/wiki/Mixture_model

[19] https://en.wikipedia.org/wiki/K-means_clustering

[20] https://en.wikipedia.org/wiki/Silhouette_(clustering)

[21] https://en.wikipedia.org/wiki/Curse_of_dimensionality

[22] https://stackoverflow.com/questions/47370795/pca-on-sklearn-how-to-interpret-pca-components

[23] https://towardsdatascience.com/interpretable-k-means-clusters-feature-importances-7e516eeb8d3c

[24] https://en.wikipedia.org/wiki/Ordinary_least_squares

[25] https://en.wikipedia.org/wiki/Coefficient_of_determination

[26] https://en.wikipedia.org/wiki/Polynomial_regression

[27] https://en.wikipedia.org/wiki/Multicollinearity

[28] https://en.wikipedia.org/wiki/Variance_inflation_factor

[29] https://en.wikipedia.org/wiki/Lasso_(statistics)