Case Study 18: Variable Selection for Linear Regression Models

In this Case Study, we will explore variable selection options that we have for linear regression models.

Specifically, we will analyze data for predicting breast tumor size using gene expression data.

Chanrion et al. (2008) reported results of a study of 155 patients treated for breast cancer with tamoxofen. The patients were followed for a period of time and diagnosed as having a recurrence of breast cancer (R), or being recurrence free (RF). Various clinical measurements were made including tumor size at the time of treatment. Gene expression was measured for a large number of gene sequences. Here we focus on a sample of 50 gene sequences and study the extent to which these might be predictive of tumor size.

The data for this example are in two different files:

- 'clinical_data.csv' contains the clinical measurements and status for the patients in the study
- 'gene_expr.csv' contains gene expression values for 50 genes

Preparing the Data

In [60]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns; sns.set()
```

In [2]:
```
clin = pd.read_csv("clinical_data.csv")
clin
```

Out[2]:

	Tumor sample	Status	Patient age (years)	Adjuvant therapy	Histological type	SBR Grade	Tumor size (mm)	pT (TNM)	Lymph node checked	N+	...	ER	PR	(m
0	EB5012	RF	75.6	X-ray + Tam	IDC	2.0	13.0	pT1	15	3.0	...	460	28	
1	EB5015	RF	55.3	X-ray + Tam	IDC	1.0	15.0	pT1	22	1.0	...	39	80	
2	EB5018	RF	64.4	X-ray + Tam	IDC	2.0	15.0	pT1	5	1.0	...	26	83	
3	EB5019	RF	55.4	X-ray + Tam	IDC	1.0	20.0	pT2	9	1.0	...	79	143	
4	EB5020	RF	51.1	X-ray + Tam	IDC	1.0	10.0	pT1	10	2.0	...	26	210	
...	...	...	...	...	...	...	...	...	...	...	...	...	...	
150	VB0488	R	58.9	X-ray+Tam	IDC	2.0	20.0	pT2	3	1.0	...	195	148	
151	VB0534	R	70.7	X-ray+Tam	ILC	2.0	30.0	pT2	13	0.0	...	29	19	

	Tumor sample	Status	Patient age (years)	Adjuvant therapy	Histological type	SBR Grade	Tumor size (mm)	pT (TNM)	Lymph node checked	N+	...	ER	PR	(m
152	VB1099	R	76.5	X-ray+Tam	IDC+ICC	2.0	15.0	pT1	10	9.0	...	253	41	
153	VB1165	R	59.9	X-ray+Tam	ILC	2.0	20.0	pT1	13	0.0	...	369	<10	
154	VB1218	R	92.4	X-ray+Tam	IDC	2.0	22.0	pT2	8	5.0	...	<10	13	

155 rows × 22 columns

In [3]:
```
gene = pd.read_csv("gene_expr.csv")
gene
```

Out[3]:

	X159	X960	X980	X986	X1023	X1028	X1064	X1092	X1103	X
0	0.739512	1.971036	-1.660842	2.777183	3.299062	-1.954834	2.784970	0.411848	3.974931	-0.97
1	2.037903	2.197854	-1.263034	4.082346	5.426886	-1.732520	3.085890	0.688056	4.503384	-1.18
2	2.218338	3.471559	-1.789433	2.829994	4.746466	-2.222392	2.977280	0.944858	4.021099	-1.82
3	0.972344	2.638734	-2.010999	3.913935	4.744161	-2.496426	3.139577	0.155651	4.632121	-1.67
4	2.412235	4.033491	-1.536501	4.239650	4.304348	-1.991067	3.700095	0.878536	4.295705	-2.14
...	...	...	...	...	...	...	...	...	...	...
150	1.037313	-1.084316	3.636530	0.764423	-0.166778	-0.379772	-1.300045	2.032701	0.473679	3.46
151	1.133431	-0.618910	4.335286	-0.191048	0.128324	0.363812	-0.671377	3.962470	0.406625	3.38
152	1.612861	-1.200427	5.036507	-0.074318	-0.353972	-0.393072	-1.172946	2.084975	0.809940	2.81
153	1.204820	-1.413874	4.426671	-0.174298	-0.629302	-0.935110	-2.120142	1.191116	0.567390	2.79
154	0.742063	-0.842899	4.180525	0.025311	-0.065291	-0.630354	-1.176323	4.576426	0.427749	3.52

155 rows × 50 columns

Merging Relevant Variables into One Dataframe

Let's merge the gene expression and tumor size dataframes into one dataframe and check for missing values.

In [4]:
```
# Merge features and target into one dataframe
#
df = gene
df['size'] = clin['Tumor size (mm)']
df.head()
```

Out[4]:

	X159	X960	X980	X986	X1023	X1028	X1064	X1092	X1103	X1109
0	0.739512	1.971036	-1.660842	2.777183	3.299062	-1.954834	2.784970	0.411848	3.974931	-0.979751
1	2.037903	2.197854	-1.263034	4.082346	5.426886	-1.732520	3.085890	0.688056	4.503384	-1.185032
2	2.218338	3.471559	-1.789433	2.829994	4.746466	-2.222392	2.977280	0.944858	4.021099	-1.825502

	X159	X960	X980	X986	X1023	X1028	X1064	X1092	X1103	X1109
3	0.972344	2.638734	-2.010999	3.913935	4.744161	-2.496426	3.139577	0.155651	4.632121	-1.671513
4	2.412235	4.033491	-1.536501	4.239650	4.304348	-1.991067	3.700095	0.878536	4.295705	-2.141092

5 rows × 51 columns

Identifying Missing Data

The code below displays whether each entry in the dataframe is NOT a missing value (True) or not (False).

In [5]:
```
# Check for missing values: display column name
# for any column with missing values
#
df.notna()
```

Out[5]:

	X159	X960	X980	X986	X1023	X1028	X1064	X1092	X1103	X1109	...	X1574	X1595	X1597
0	True	True	True	True	True	True	True	True	True	True	...	True	True	True
1	True	True	True	True	True	True	True	True	True	True	...	True	True	True
2	True	True	True	True	True	True	True	True	True	True	...	True	True	True
3	True	True	True	True	True	True	True	True	True	True	...	True	True	True
4	True	True	True	True	True	True	True	True	True	True	...	True	True	True
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
150	True	True	True	True	True	True	True	True	True	True	...	True	True	True
151	True	True	True	True	True	True	True	True	True	True	...	True	True	True
152	True	True	True	True	True	True	True	True	True	True	...	True	True	True
153	True	True	True	True	True	True	True	True	True	True	...	True	True	True
154	True	True	True	True	True	True	True	True	True	True	...	True	True	True

155 rows × 51 columns

The code below returns a true if it is the case that ALL entries in each column are not missing values.

In [6]:
```
df.notna().all()
```

Out[6]:
```
X159      True
X960      True
X980      True
X986      True
X1023     True
X1028     True
X1064     True
X1092     True
X1103     True
X1109     True
X1124     True
X1136     True
X1141     True
X1144     True
X1169     True
X1173     True
X1179     True
```

```
X1193     True
X1203     True
X1206     True
X1208     True
X1219     True
X1232     True
X1264     True
X1272     True
X1292     True
X1297     True
X1329     True
X1351     True
X1362     True
X1416     True
X1417     True
X1418     True
X1430     True
X1444     True
X1470     True
X1506     True
X1514     True
X1529     True
X1553     True
X1563     True
X1574     True
X1595     True
X1597     True
X1609     True
X1616     True
X1637     True
X1656     True
X1657     True
X1683     True
size     False
dtype: bool
```

Putting this all together, the code below displays only the columns that have at least one missing value.

In [7]:
```
df.columns[df.notna().all()==False]
```

Out[7]:
```
Index(['size'], dtype='object')
```

There were missing values in 'size'. Which observations were they?

In [8]:
```
df['size'][df['size'].isna()]
```

Out[8]:
```
23    NaN
63    NaN
91    NaN
132   NaN
142   NaN
Name: size, dtype: float64
```

We can then clean the data by dropping rows with missing tumor size (the intended label).

In [9]:
```
dfclean = df.dropna()

dfclean.shape
```

Out[9]:
```
(150, 51)
```

Descriptive Analytics

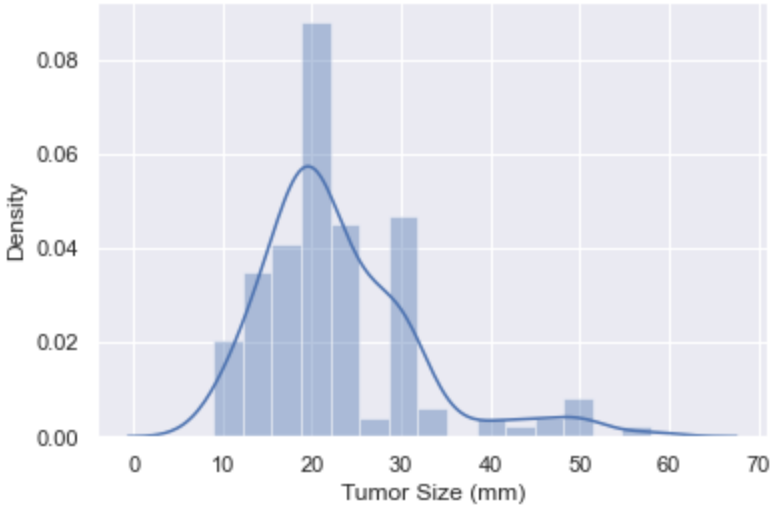
Before setting up our linear regression model, let's learn more about this dataset by performing some descriptive analytics.

Distribution of tumor size in the data

Because tumor size will be our response variable, let's first examine the distribution of tumor sizes in this dataset.

In [10]:
```
# raw tumor size
sns.distplot(dfclean['size'])
plt.xlabel("Tumor Size (mm)")
plt.show()
```

```
/Users/jdeeke/miniconda3/lib/python3.8/site-packages/seaborn/distributions.py:2619: FutureWarning: `distplot` is a deprecated function and will be removed in a future version. Please adapt your code to use either `displot` (a figure-level function with similar flexibility) or `histplot` (an axes-level function for histograms).
  warnings.warn(msg, FutureWarning)
```



Pretty skewed. Although that might be explained by the X variables.

Relationship between tumor size and gene expression levels

Scatterplots

Next, let's examine each of the individual relationships between the tumor size and the 50 gene expression levels. First, we will use the **sns.pairplot()** function to visualize the relationship bewteen the just size variable and the 5 gene expression variables.

If we do not want the **sns.pairplot()** function to visualize a scatterplot for every single pair of numerical variables in a dataframe, then we can use the **x_vars** and the **y_vars** parameters to specify which column names we do want to consider as shown below.

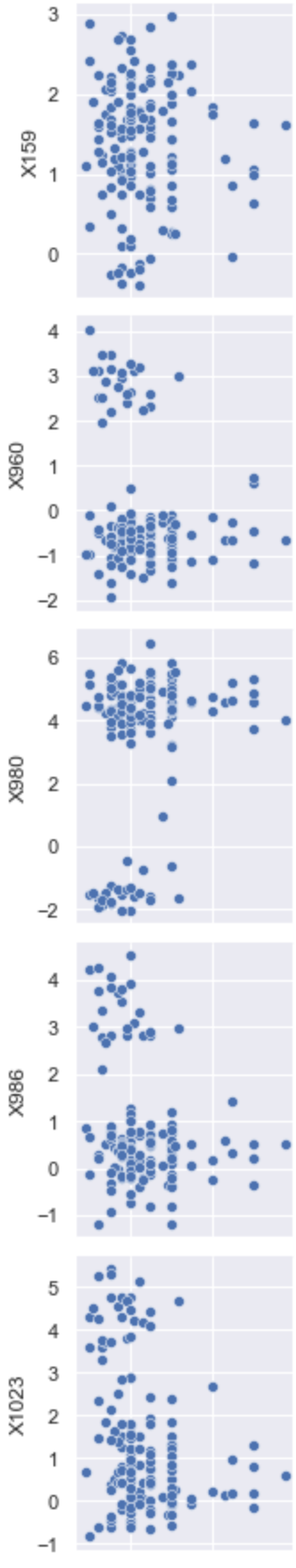
In [11]:
```
dfclean.columns[0:50]
```

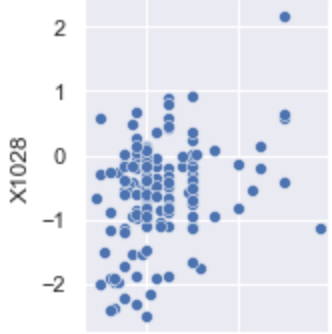
Out[11]:
```
Index(['X159', 'X960', 'X980', 'X986', 'X1023', 'X1028', 'X1064', 'X1092',
       'X1103', 'X1109', 'X1124', 'X1136', 'X1141', 'X1144', 'X1169', 'X1173',
       'X1179', 'X1193', 'X1203', 'X1206', 'X1208', 'X1219', 'X1232', 'X1264',
       'X1272', 'X1292', 'X1297', 'X1329', 'X1351', 'X1362', 'X1416', 'X1417',
       'X1418', 'X1430', 'X1444', 'X1470', 'X1506', 'X1514', 'X1529', 'X1553',
       'X1563', 'X1574', 'X1595', 'X1597', 'X1609', 'X1616', 'X1637', 'X1656',
```

```
       'X1657', 'X1683'],
      dtype='object')
```

In [12]:

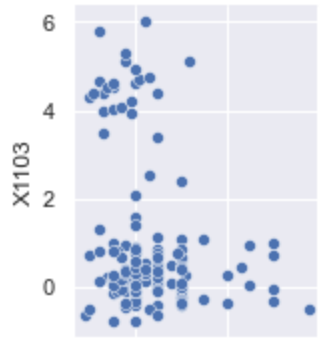
```
sns.pairplot(
    dfclean,
    x_vars=["size"],
    y_vars=dfclean.columns[0:50],
)
plt.show()
```

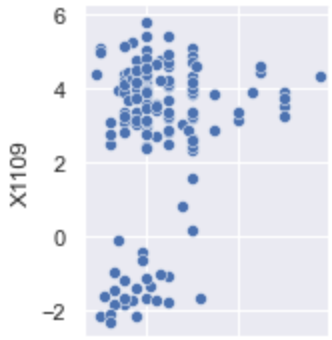



X1028
2
1
0
−1
−2

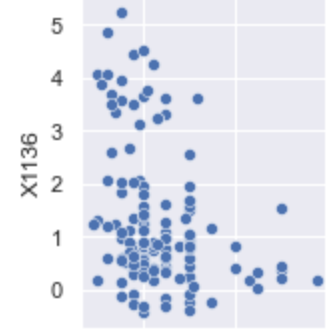

X1064
4
2
0
−2

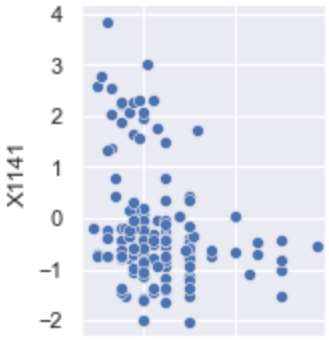

X1092
4
2
0

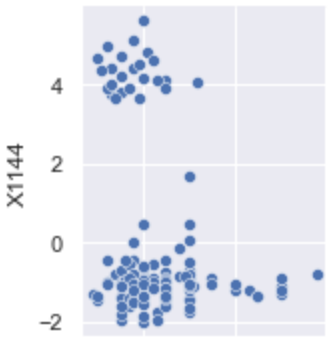

X1103
6
4
2
0


X1109
6
4
2
0
−2


X1124
6
4
2
0


X1136
5
4
3
2
1
0


X1141
4
3
2
1
0
−1
−2


X1144
4
2
0
−2

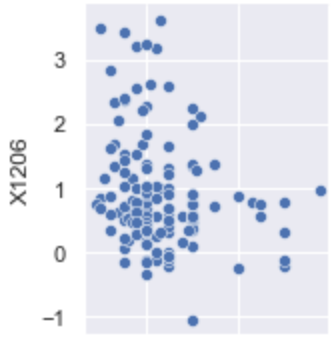

X1169
4
2
0
−2


X1173
3
2
1
0
−1

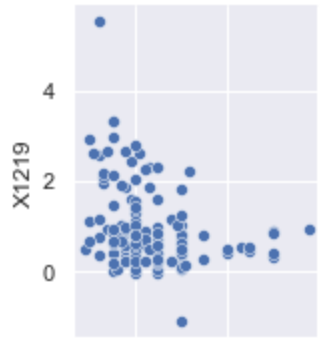

X1179
2
1
0
−1


X1193
4
2
0
−2

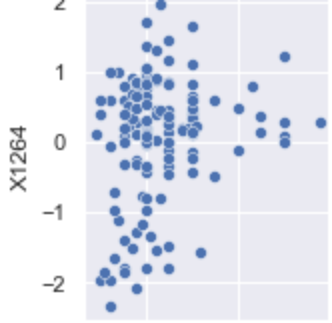

X1203
6
4
2
0


X1206
3
2
1
0
−1

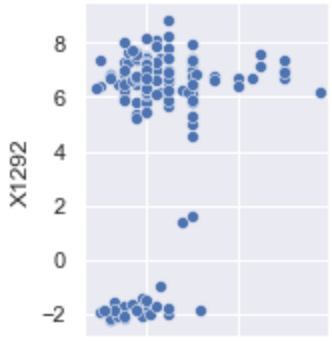

X1208
4
3
2
1
0
−1

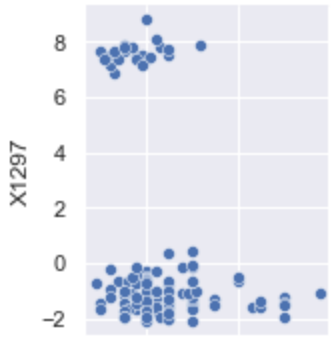

X1219
4
2
0

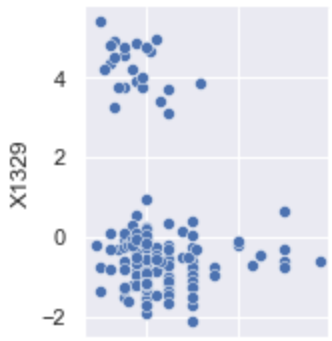

X1232
2
1
0
−1


X1264
2
1
0
−1
−2

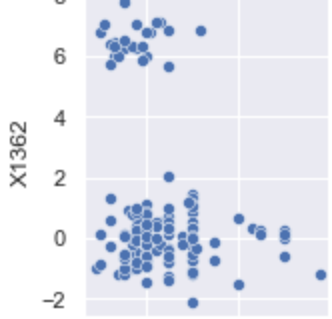

X1272
4
2
0
−2

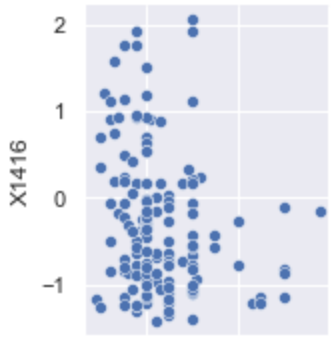

X1292
8
6
4
2
0
−2

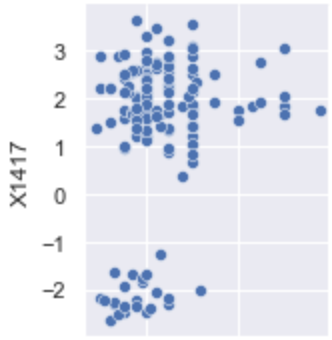

X1297
8
6
4
2
0
−2

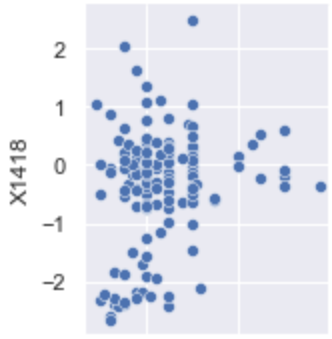

X1329
4
2
0
−2

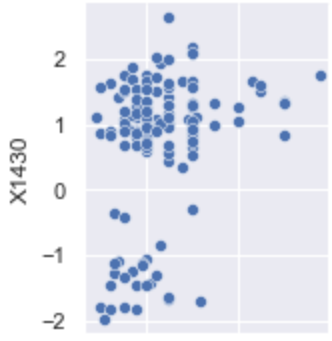

X1351
4
3
2
1
0
−1

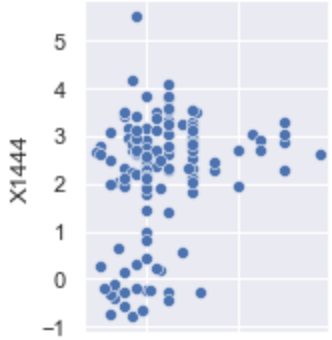

X1362
6
4
2
0
−2


X1416
2
1
0
−1

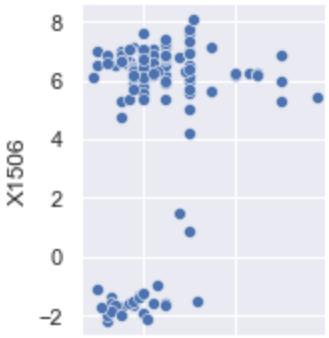

X1417
3
2
1
0
−1
−2

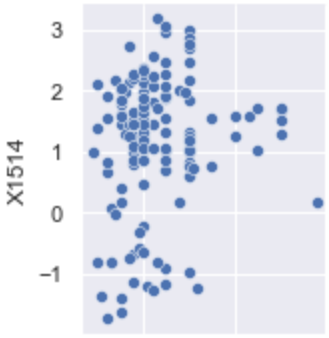

X1418
2
1
0
−1
−2

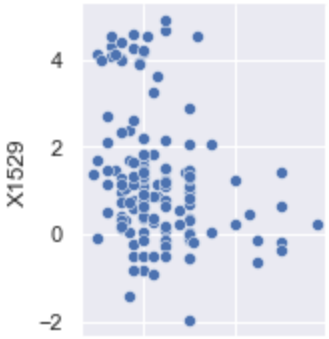

X1430
2
1
0
−1
−2

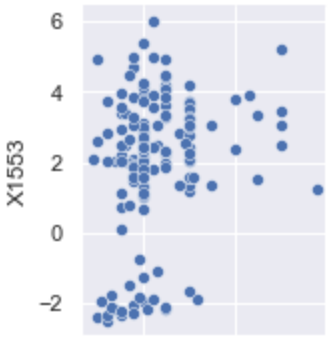

X1444
5
4
3
2
1
0
−1


X1470
8
6
4
2
0


X1506
8
6
4
2
0
−2

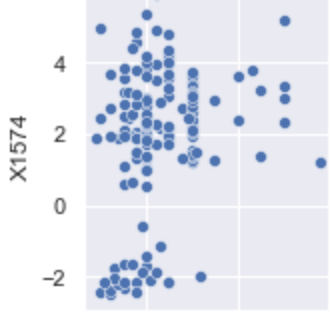

X1514
3
2
1
0
−1

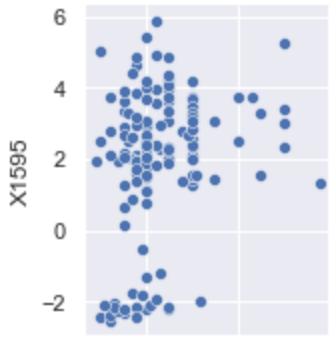

X1529
4
2
0
−2

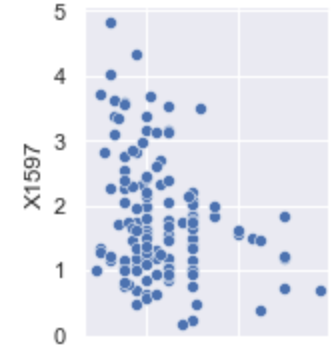

X1553
6
4
2
0
−2

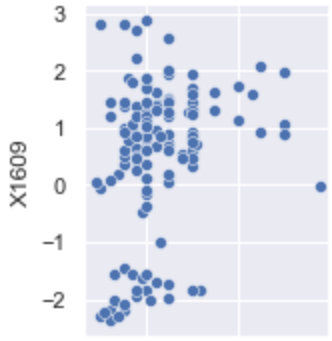

X1563
5
4
3
2
1
0
−1

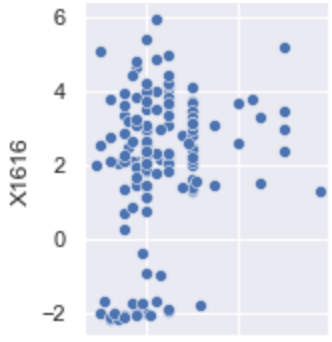
6

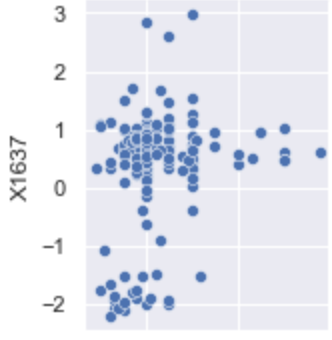

X1574
4
2
0
−2

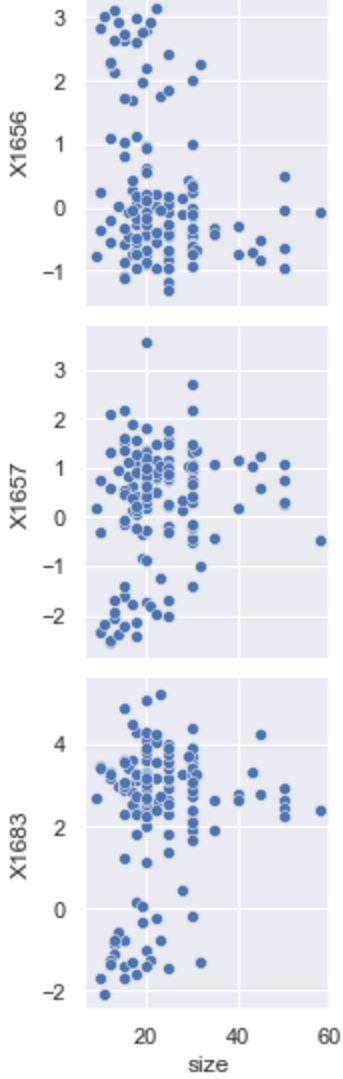

X1595
6
4
2
0
−2


X1597
5
4
3
2
1
0


X1609
3
2
1
0
−1
−2


X1616
6
4
2
0
−2


X1637
3
2
1
0
−1
−2



It looks like the relationship between the tumor size and some of these gene expression levels may not be completely linear. Therefore, when we calculate and interpret the correlations between each of these variables below, we should be cautious.

In addition, it looks like there may be some sort of grouping or **clustering** structure of the patients in this dataset. A **clustering algorithm** such as **k-means** might also be useful in exploring the hidden relationships of the gene expression levels and the tumor sizes in this dataset. (**STAT430-Unsupervised Learning** can be useful for learning more about this underlying clustering structure in dataset).

Correlations

Next, let's look at just the correlation between the tumor size and the gene expression levels.

In [13]:
```
dfclean.corr()
```

Out[13]:

	X159	X960	X980	X986	X1023	X1028	X1064	X1092	X1103
X159	1.000000	0.066745	-0.104831	0.025742	0.023293	-0.280436	-0.043324	-0.202610	0.055942
X960	0.066745	1.000000	-0.894160	0.925091	0.870388	-0.641563	0.935203	-0.470886	0.882168
X980	-0.104831	-0.894160	1.000000	-0.833771	-0.767237	0.770950	-0.880199	0.504594	-0.890721
X986	0.025742	0.925091	-0.833771	1.000000	0.815923	-0.625476	0.911853	-0.469611	0.845791
X1023	0.023293	0.870388	-0.767237	0.815923	1.000000	-0.609730	0.786784	-0.408869	0.749220
X1028	-0.280436	-0.641563	0.770950	-0.625476	-0.609730	1.000000	-0.610990	0.495474	-0.658699

	X159	X960	X980	X986	X1023	X1028	X1064	X1092	X1103
X1064	-0.043324	0.935203	-0.880199	0.911853	0.786784	-0.610990	1.000000	-0.490571	0.878715
X1092	-0.202610	-0.470886	0.504594	-0.469611	-0.408869	0.495474	-0.490571	1.000000	-0.458830
X1103	0.055942	0.882168	-0.890721	0.845791	0.749220	-0.658699	0.878715	-0.458830	1.000000
X1109	-0.129250	-0.871116	0.950441	-0.790289	-0.720030	0.730405	-0.855813	0.406759	-0.866261
X1124	0.017988	0.871704	-0.809285	0.850737	0.678425	-0.592205	0.915333	-0.447681	0.806964
X1136	0.175025	0.809158	-0.834491	0.780174	0.702090	-0.644116	0.767399	-0.474566	0.868931
X1141	0.002425	0.871253	-0.834486	0.830941	0.758415	-0.629569	0.865917	-0.476715	0.842050
X1144	0.046500	0.941457	-0.947119	0.900289	0.789929	-0.699407	0.951224	-0.479527	0.916472
X1169	-0.010883	0.926246	-0.897404	0.868840	0.788802	-0.669057	0.914425	-0.462668	0.885309
X1173	0.074232	0.883840	-0.893213	0.841747	0.768991	-0.700265	0.858960	-0.534545	0.868512
X1179	0.086510	0.744318	-0.747665	0.659681	0.662412	-0.622661	0.664611	-0.373440	0.736683
X1193	0.080197	0.937084	-0.898685	0.880916	0.811169	-0.646622	0.911706	-0.483284	0.894798
X1203	0.069778	0.915699	-0.859943	0.902524	0.779165	-0.644649	0.901203	-0.501649	0.919788
X1206	0.170467	0.762115	-0.701911	0.792253	0.669996	-0.601211	0.733109	-0.522741	0.739601
X1208	0.057921	0.908869	-0.911290	0.883716	0.773338	-0.666998	0.922524	-0.464204	0.900462
X1219	0.180408	0.779785	-0.767053	0.779501	0.644425	-0.606031	0.738619	-0.445959	0.778162
X1232	0.387639	0.348852	-0.329126	0.313909	0.214296	-0.385610	0.308042	-0.299802	0.257914
X1264	-0.063863	-0.785187	0.791860	-0.784906	-0.710885	0.652789	-0.804288	0.408886	-0.763568
X1272	-0.060304	-0.903459	0.944876	-0.861871	-0.760259	0.737613	-0.908371	0.489252	-0.865382
X1292	-0.097760	-0.914038	0.979569	-0.856286	-0.779023	0.745279	-0.903632	0.492162	-0.899345
X1297	0.024347	0.944898	-0.947926	0.900901	0.816155	-0.702602	0.942883	-0.483154	0.930711
X1329	0.003902	0.933209	-0.900720	0.890132	0.795104	-0.662516	0.913246	-0.458154	0.882063
X1351	0.071365	0.815941	-0.830614	0.751348	0.685504	-0.641621	0.798027	-0.468603	0.798251
X1362	0.025934	0.920544	-0.893161	0.877753	0.804857	-0.644734	0.902713	-0.444802	0.927801
X1416	0.219487	0.416013	-0.521002	0.386305	0.252880	-0.466091	0.417780	-0.502383	0.425317
X1417	-0.087216	-0.893396	0.940285	-0.834690	-0.783367	0.763985	-0.881190	0.457006	-0.855119
X1418	-0.091005	-0.780292	0.782619	-0.754446	-0.681776	0.623760	-0.783713	0.404107	-0.761407
X1430	-0.096587	-0.884711	0.907381	-0.817627	-0.777524	0.682521	-0.850185	0.480682	-0.863529
X1444	-0.150056	-0.817140	0.873892	-0.774423	-0.705041	0.718377	-0.799694	0.507306	-0.783583
X1470	0.042644	0.933201	-0.921284	0.908233	0.793085	-0.684112	0.930692	-0.471529	0.895721
X1506	-0.048830	-0.911708	0.974838	-0.862019	-0.783062	0.734861	-0.908696	0.487022	-0.903199
X1514	-0.088654	-0.802208	0.825526	-0.737415	-0.678616	0.682179	-0.819659	0.441564	-0.711679
X1529	0.272383	0.802861	-0.809486	0.739250	0.679274	-0.645365	0.749037	-0.468831	0.754183
X1553	-0.075158	-0.821128	0.874924	-0.778078	-0.771182	0.764840	-0.793098	0.564363	-0.763715
X1563	0.156287	0.896561	-0.874840	0.859510	0.776526	-0.700428	0.864322	-0.532883	0.873731
X1574	-0.072688	-0.848674	0.857792	-0.820038	-0.798989	0.753340	-0.822937	0.589041	-0.763501
X1595	-0.066504	-0.853107	0.864328	-0.815907	-0.797286	0.756401	-0.822922	0.587226	-0.767650
X1597	0.333983	0.740220	-0.728531	0.717987	0.608442	-0.649908	0.686648	-0.427521	0.782595

	X159	X960	X980	X986	X1023	X1028	X1064	X1092	X1103
X1609	-0.178051	-0.817977	0.881564	-0.767929	-0.697912	0.730739	-0.799453	0.504697	-0.804683
X1616	-0.064006	-0.844117	0.853613	-0.807926	-0.789011	0.749460	-0.817137	0.587326	-0.757206
X1637	-0.086649	-0.834558	0.840019	-0.801090	-0.695183	0.676955	-0.840555	0.448633	-0.788661
X1656	0.139922	0.807595	-0.849641	0.733349	0.653549	-0.653344	0.783910	-0.461038	0.787281
X1657	-0.080753	-0.774530	0.777174	-0.768140	-0.663290	0.651060	-0.789905	0.461783	-0.745999
X1683	-0.139577	-0.826996	0.893544	-0.784404	-0.667766	0.683719	-0.832483	0.426441	-0.801585
size	-0.044613	-0.212010	0.244669	-0.244490	-0.287543	0.327292	-0.182425	0.201470	-0.246617

51 rows × 51 columns

In [14]:
```
dfclean.corr()['size']
```

Out[14]:
```
X159    -0.044613
X960    -0.212010
X980     0.244669
X986    -0.244490
X1023   -0.287543
X1028    0.327292
X1064   -0.182425
X1092    0.201470
X1103   -0.246617
X1109    0.213882
X1124   -0.190937
X1136   -0.312942
X1141   -0.294343
X1144   -0.245640
X1169   -0.280702
X1173   -0.233817
X1179   -0.242076
X1193   -0.266954
X1203   -0.265788
X1206   -0.217645
X1208   -0.231898
X1219   -0.297885
X1232   -0.084374
X1264    0.200336
X1272    0.226608
X1292    0.245428
X1297   -0.266409
X1329   -0.259232
X1351   -0.299613
X1362   -0.252061
X1416   -0.232698
X1417    0.243210
X1418    0.180415
X1430    0.278647
X1444    0.247366
X1470   -0.263179
X1506    0.230032
X1514    0.136756
X1529   -0.290717
X1553    0.239752
X1563   -0.293857
X1574    0.256957
X1595    0.254229
X1597   -0.273540
X1609    0.261237
```

```
X1616    0.248713
X1637    0.229905
X1656   -0.293103
X1657    0.186372
X1683    0.197666
size     1.000000
Name: size, dtype: float64
```

When the number of feature variables considered is large, it can be useful to screen variables in advance by computing the individual slopes of each X variable on y or vice versa. There are other possibilities, for example, plotting coefficient t test statistics or sample correlations. Let's plot the correlations. Then we can visualize the relations by graphing slopes versus coefficient number.

In [15]:
```
# Compute correlation matrix for screening purposes.
# Extract first column and remove correlation of size with itself.
corrxy = dfclean.corr()['size'][:len(dfclean.columns)-1]
corrxy
```

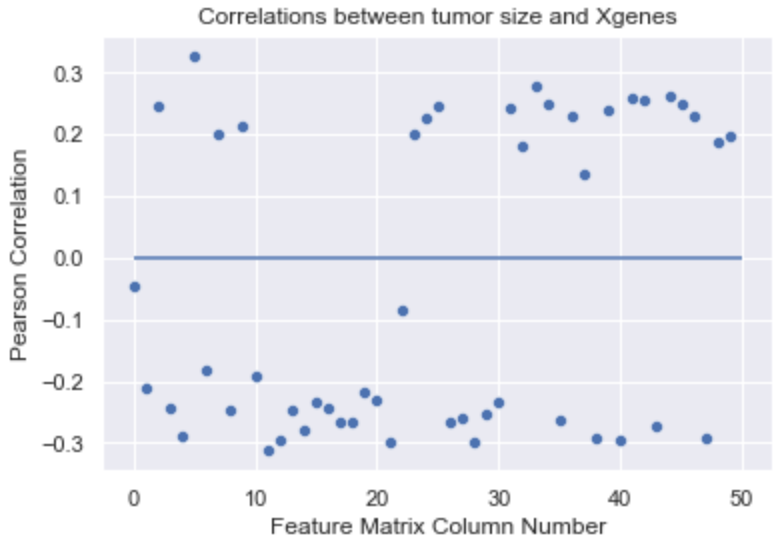
Out[15]:
```
X159    -0.044613
X960    -0.212010
X980     0.244669
X986    -0.244490
X1023   -0.287543
X1028    0.327292
X1064   -0.182425
X1092    0.201470
X1103   -0.246617
X1109    0.213882
X1124   -0.190937
X1136   -0.312942
X1141   -0.294343
X1144   -0.245640
X1169   -0.280702
X1173   -0.233817
X1179   -0.242076
X1193   -0.266954
X1203   -0.265788
X1206   -0.217645
X1208   -0.231898
X1219   -0.297885
X1232   -0.084374
X1264    0.200336
X1272    0.226608
X1292    0.245428
X1297   -0.266409
X1329   -0.259232
X1351   -0.299613
X1362   -0.252061
X1416   -0.232698
X1417    0.243210
X1418    0.180415
X1430    0.278647
X1444    0.247366
X1470   -0.263179
X1506    0.230032
X1514    0.136756
X1529   -0.290717
X1553    0.239752
X1563   -0.293857
X1574    0.256957
X1595    0.254229
X1597   -0.273540
X1609    0.261237
X1616    0.248713
```

```
X1637    0.229905
X1656   -0.293103
X1657    0.186372
X1683    0.197666
Name: size, dtype: float64
```

In [16]:

```
sns.scatterplot(np.arange(np.size(corrxy)), corrxy)
plt.ylabel('Pearson Correlation')
plt.xlabel('Feature Matrix Column Number')
plt.title("Correlations between tumor size and Xgenes")
plt.hlines(y=0, xmin=0, xmax=50)
plt.show()
```

```
/Users/jdeeke/miniconda3/lib/python3.8/site-packages/seaborn/_decorators.py:36: FutureWarn
ing: Pass the following variables as keyword args: x, y. From version 0.12, the only valid
positional argument will be `data`, and passing other arguments without an explicit keywor
d will result in an error or misinterpretation.
  warnings.warn(
```



A Variable Selection Preprocessing Step

For ultra high dimensional regression problems, one technique is to screen out variables whose correlations with y fall below a threshold. Here, there might be two variables out of the 50 screened in this way, but otherwise nothing jumps out here.

The common correlation magnitudes across many genes suggests that they may all be moving more or less together, with no particular genes among this group is dominant in having a strong association with tumor size. Possibly the optimal model would selects a subset of variables that represents the relationship about entire groups of genes.

Feature and Target Matrices

Using the language of machine learning, the dataset of explanatory variables is commonly called the **feature matrix** and the dataset of the response variable is commonly called the **target array**. Let's extract these two dataframes below first.

In [17]:

```
# Extract target and feature matrix from dfclean:
y = dfclean['size']
X = dfclean.drop(columns='size')
```

More Descriptive Analytics - Quickly Summarizing the Relationship Between the Response Variable and Each of the Explanatory Variables

Next, it might be useful if we could quickly summarize the relationship between each of the gene expression levels and the tumor size in just one plot. Below is one way that we could do this.

Let's divide up the values of our tumor sizes into bins, as we would for a histogram, and compare each of the gene expression means across those bins.

Let's bin the tumor sizes into 5 groups based on percentiles of the sample distribution, working with the target,

```
y = tumor size
```

and comparing values in the feature matrix

```
X = matrix of gene expression values
```

Determine the Four Sizes Thresholds that Separate:

We can use the pandas **.quantile()** function to bin the corresponding tumor sizes that are:

- Tumors that are in the bottom 20% of sizes.
- Tumors that are in the 20%-40% of tumor sizes.
- Tumors that are in the 40%-60% of tumor sizes.
- Tumors that are in the 60%-80% of tumor sizes.
- Tumors that are in the top 20% of tumor sizes.

In [18]:
```
# Compute selected percentiles of log2 tumor size
quants = y.quantile(q=[0.2, 0.40, 0.60, 0.80])
quants
```

Out[18]:
```
0.2    17.0
0.4    20.0
0.6    22.4
0.8    30.0
Name: size, dtype: float64
```

Now let's create a new categorical variable ylevel that labels:

- Tumors that are in the bottom 20% of sizes as 0
- Tumors that are in the 20%-40% of tumor sizes as 1
- Tumors that are in the 40%-60% of tumor sizes as 2
- Tumors that are in the 60%-80% of tumor sizes as 3
- Tumors that are in the top 20% of tumor sizes as 4.

To do this we will first create a pandas series 'ylevel' that is comprised of all 0's and has the same dimensions as our pandas series y.

In [19]:
```
ylevel = y*0
ylevel
```

Out[19]:
```
0      0.0
1      0.0
2      0.0
3      0.0
4      0.0
      ...
150    0.0
151    0.0
152    0.0
153    0.0
154    0.0
Name: size, Length: 150, dtype: float64
```

In [20]:
```
level = 0
for cut in quants:
    print('Cut Threshold: ',cut)
    level = level + 1

    #For each response variable value that is greater than or equal to that particular cut
    ylevel[y >= cut] = level

df_size_levels=pd.DataFrame({'y': y, 'ylevel': ylevel})
df_size_levels
```

```
Cut Threshold:  17.0
Cut Threshold:  20.0
Cut Threshold:  22.39999999999999
Cut Threshold:  30.0
```

Out[20]:

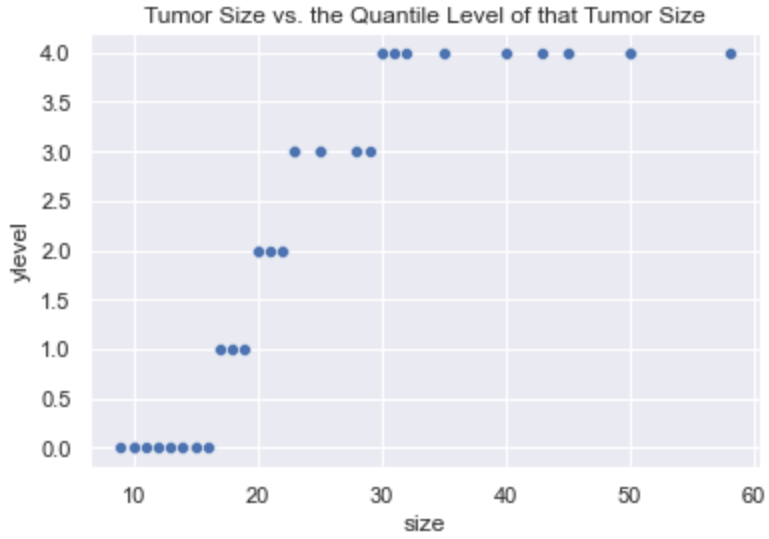
	y	ylevel
0	13.0	0.0
1	15.0	0.0
2	15.0	0.0
3	20.0	2.0
4	10.0	0.0
...	...	...
150	20.0	2.0
151	30.0	4.0
152	15.0	0.0
153	20.0	2.0
154	22.0	2.0

150 rows × 2 columns

In [21]:
```
sns.scatterplot(y, ylevel)
plt.ylabel('ylevel')
plt.title('Tumor Size vs. the Quantile Level of that Tumor Size')
plt.show()
```

```
/Users/jdeeke/miniconda3/lib/python3.8/site-packages/seaborn/_decorators.py:36: FutureWarn
ing: Pass the following variables as keyword args: x, y. From version 0.12, the only valid
positional argument will be `data`, and passing other arguments without an explicit keywor
```

```
d will result in an error or misinterpretation.
  warnings.warn(
```



Find the average feature size for each tumor size category.

In [22]:
```
df['ylevel']=ylevel

df.head()
```

Out[22]:

	X159	X960	X980	X986	X1023	X1028	X1064	X1092	X1103	X1109
0	0.739512	1.971036	-1.660842	2.777183	3.299062	-1.954834	2.784970	0.411848	3.974931	-0.979751
1	2.037903	2.197854	-1.263034	4.082346	5.426886	-1.732520	3.085890	0.688056	4.503384	-1.185032
2	2.218338	3.471559	-1.789433	2.829994	4.746466	-2.222392	2.977280	0.944858	4.021099	-1.825502
3	0.972344	2.638734	-2.010999	3.913935	4.744161	-2.496426	3.139577	0.155651	4.632121	-1.671513
4	2.412235	4.033491	-1.536501	4.239650	4.304348	-1.991067	3.700095	0.878536	4.295705	-2.141092

5 rows × 52 columns

In [23]:
```
mean_levels=df.drop(['size'], axis=1).groupby(['ylevel']).mean()
mean_levels
```

Out[23]:

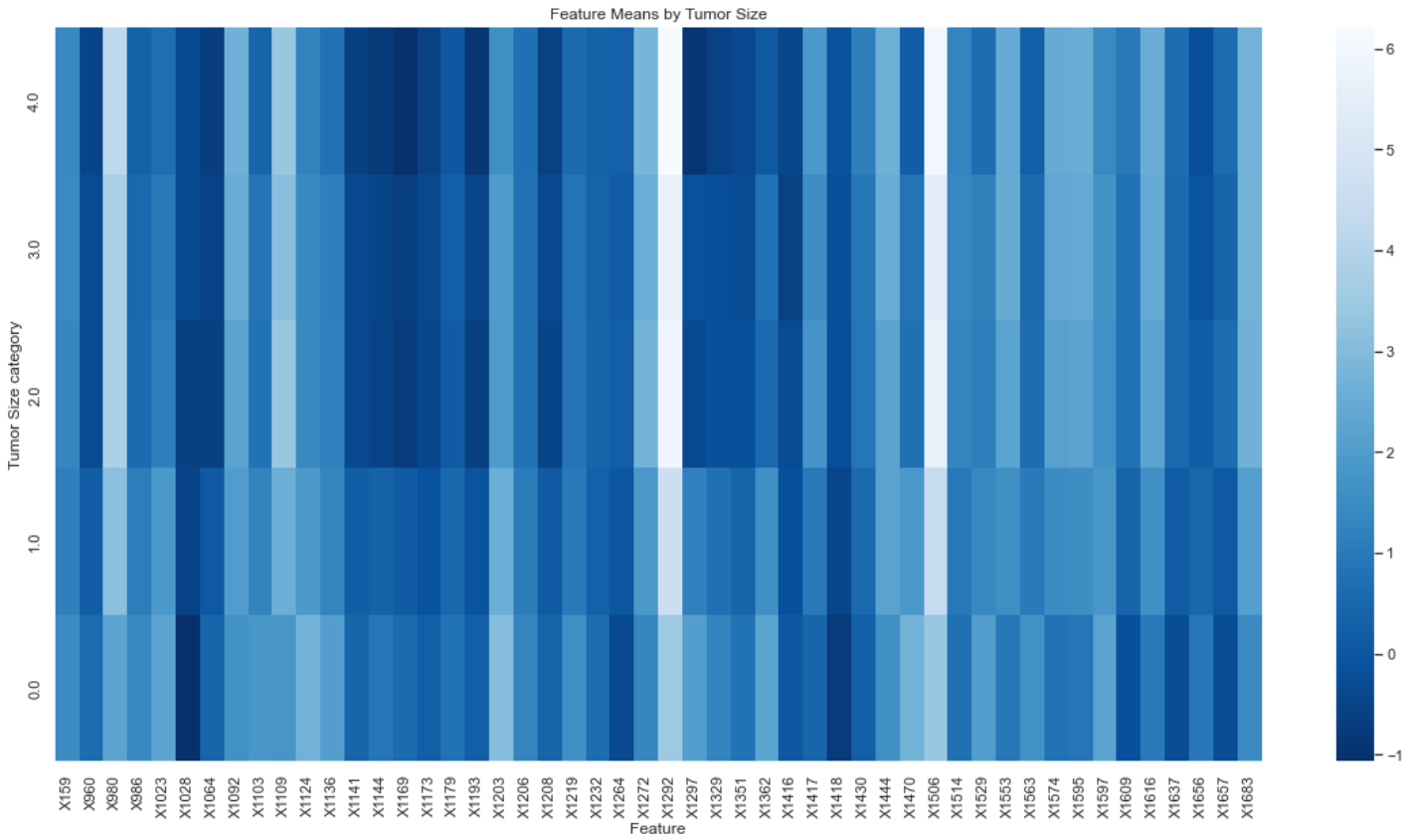
ylevel	X159	X960	X980	X986	X1023	X1028	X1064	X1092	X1103	X1
0.0	1.504755	0.612521	2.233018	1.404236	2.239300	-1.066753	0.391509	1.660575	1.830215	1.801
1.0	1.126742	0.203442	3.126544	1.076623	1.872226	-0.551960	0.072738	1.989218	1.249245	2.593
2.0	1.344909	-0.311217	3.787881	0.595172	1.138927	-0.618690	-0.639796	2.202338	0.805685	3.369
3.0	1.448158	-0.314860	3.774694	0.509667	1.002144	-0.410417	-0.542063	2.542014	0.839154	3.112
4.0	1.397943	-0.515673	4.197393	0.350807	0.689921	-0.337765	-0.682360	2.675173	0.427064	3.387

5 rows × 50 columns

Plot in a binned heatmap.

In [24]:
```
plt.figure(figsize=(20,10))
```

```
fig = sns.heatmap(mean_levels, annot=False, linewidths=0,
            square = False, cmap = 'Blues_r');
fig.set_ylim([0,5]);
plt.ylabel('Tumor Size category');
plt.xlabel('Feature');
all_sample_title = 'Feature Means by Tumor Size'
plt.title(all_sample_title, size = 12);
plt.show(fig)
```



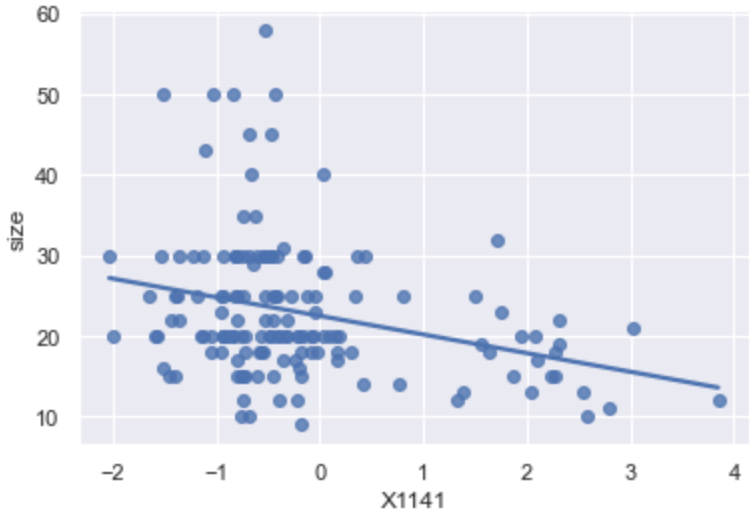
In the heat map we look for trends in intensity of the colors across different tumor size categories.

Pick out a few features and visualize the relationship between this feature and the numerical response variable (size).

Below is the scatterplots of y versus a couple of the X variables (columns 12 and 49). The least squares lines are included in the plots.

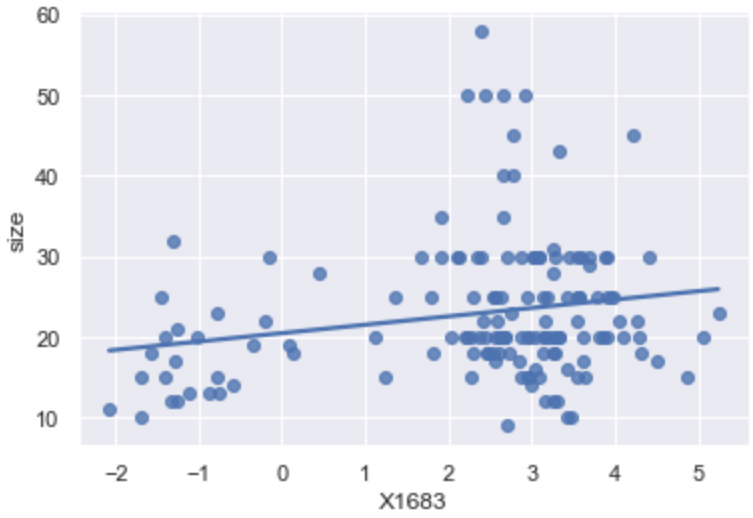
In [25]:

```
sns.regplot(x=X.iloc[:,12], y=y, ci=False)
plt.show()
```



In [26]:

```
sns.regplot(x=X.iloc[:,49], y=y, ci=False)
plt.show()
```



Comparing Non-Regularized Linear Regression and Lasso Linear Regression using a single Training and Test Dataset

Next, we would like to fit two linear regression models. Each of these linear regression models will predict the tumor size, using each of the 50 gene expression levels with a **training dataset**. Let's set up and compare the slopes for the following linear regression models.

1. Non-regularized linear regression
2. LASSO linear regression

Creating a Training Feature Matrix, a Training Target Array, a Test Feature Matrix, and a Test Target Array

We will use the **sklearn.linear_model** package along with the **LinearRegression()** and **Lasso()** functions to model our non-regularized linear regression models and lasso linear regression models respectively. Like with the **LogisticRegression()** function that we used from this package, these **LinearRegression()** and **Lasso()** functions requires a feature matrix and a target array as the input separately.

In addition, we would also like to train each of these linear regression models, not with the full dataset this time, but with a training dataset that is comprised of a random sample of 90% of the observations. Thus,

we can use the **train_test_split()** function from the **sklearn.linear_model** package to create these corresponding dataframes:

- training feature matrix
- training target array
- test feature matrix
- test target array.

In [27]:
```
from sklearn.model_selection import train_test_split
from sklearn.linear_model import Lasso, LinearRegression
```

In [28]:
```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.10, random_state=42)
```

In [29]:
```
X_train
```

Out[29]:

	X159	X960	X980	X986	X1023	X1028	X1064	X1092	X1103	
57	0.967334	-0.267347	4.485619	0.007160	2.906412	-0.869692	-1.036342	2.170722	0.143364	4.3
107	-0.227177	-0.746552	4.718697	0.426931	1.472402	0.145300	-0.565979	3.800514	0.229201	4.8
71	1.830426	-1.098734	4.763800	-0.250737	0.208695	-0.825715	-0.972202	3.058445	-0.369000	3.1
56	2.032421	-1.137504	4.639839	0.053018	0.045563	0.067815	-0.765535	2.680721	1.078814	2.8
136	1.845373	-0.787271	4.511494	0.557741	1.081900	-0.401617	-1.171611	1.955501	0.230077	4.7
...	...	...	...	...	...	...	...	...	...	
73	1.902578	-0.365649	4.126204	0.351763	1.409644	-0.171271	-0.818162	3.208871	-0.032666	3.1
109	0.687676	-0.525628	4.826888	0.306427	1.059334	-0.404066	-1.150780	4.276053	0.021859	4.1
14	1.471072	3.086622	-2.028855	3.811939	4.755416	-2.300659	3.171115	0.661461	5.287507	-2.1
95	1.813748	-0.432586	4.392673	0.539400	-0.076757	-0.133025	-1.383004	3.989450	-0.022601	3.0
105	0.316945	-0.390375	4.743326	0.551410	1.817976	0.048910	-0.698110	1.448081	0.164387	4.4

135 rows × 50 columns

In [30]:
```
X_test
```

Out[30]:

	X159	X960	X980	X986	X1023	X1028	X1064	X1092	X1103	
75	1.652077	-0.514573	4.991603	0.089886	0.734903	-0.321928	-0.621488	2.797013	0.842261	2.9
18	2.074001	2.862090	-1.460336	2.664106	3.716482	-1.967820	2.086825	0.631218	4.535928	-0.0
121	1.783376	-0.730813	4.756599	-0.378512	0.072150	-0.490986	-1.612977	3.449307	0.307055	2.7
80	1.587740	-1.205810	4.054216	-0.140007	-0.653269	-0.457348	-0.975197	2.792988	0.286471	3.2
78	1.032180	3.275634	-1.300395	4.508867	3.856110	-1.470320	2.306661	2.641685	4.944718	-1.1
32	1.210634	-0.643515	4.474013	0.810474	0.506353	-0.713477	-0.276210	0.259387	0.595851	3.8
66	1.062931	0.612270	5.307150	0.213174	1.297836	0.594123	-0.196589	3.449372	-0.294621	3.5
146	1.267480	-0.075407	4.334232	-0.185032	-0.279154	0.096862	-0.872244	3.607604	0.488286	3
70	1.524568	-1.071161	4.779535	-0.318076	0.273285	-0.546894	-1.243114	2.547183	0.082462	3.0

	X159	X960	X980	X986	X1023	X1028	X1064	X1092	X1103	
84	0.703225	-1.351222	6.454067	-0.791135	0.772766	0.413224	-1.097466	3.007232	0.471900	4.1
113	0.093867	-0.519778	5.161481	0.531447	0.436099	0.254929	-0.704993	3.717129	0.213707	4.5
12	2.428405	3.126210	-1.588805	3.074312	4.232906	-2.145198	3.682621	0.379102	4.715560	-1.3
37	-0.171046	-0.623558	3.537520	0.637750	0.821368	-0.036296	-0.359373	0.219830	0.954485	3.1
9	1.101824	2.743575	-1.366782	3.725140	4.569959	-1.527247	3.469656	0.828234	4.091210	-1.7
19	1.154648	2.511039	-1.709291	2.115477	3.774525	-1.893715	2.405909	-0.013297	3.481324	-1.4

15 rows × 50 columns

In [31]:
```
y_train
```

Out[31]:
```
57     20.0
107    20.0
71     40.0
56     35.0
136    25.0
       ...
73     17.0
109    30.0
14     18.0
95     20.0
105    18.0
Name: size, Length: 135, dtype: float64
```

In [32]:
```
y_test
```

Out[32]:
```
75     25.0
18     14.0
121    22.0
80     25.0
78     20.0
32     30.0
66     50.0
146    20.0
70     15.0
84     25.0
113    18.0
12     21.0
37     18.0
9      17.0
19     13.0
Name: size, dtype: float64
```

Train the Non-Regularized Linear Regression

Next, let's train the non-regularized linear regression model with just the training data. To perform non-regularized linear regression, we use the **LinearRegression()** function. No additional parameters are required for this function.

In [33]:
```
clf0 = LinearRegression()
clf0.fit(X_train, y_train)
```

Out[33]:
```
LinearRegression()
```

Here are all of the feature coefficient estimates:

In [34]:
```
pd.DataFrame(clf0.coef_.T, columns = ['non_reg_linear'], index=X_train.columns)
```

Out[34]:

	non_reg_linear
X159	1.860688
X960	7.197889
X980	-5.483316
X986	-0.996936
X1023	-2.768050
X1028	5.217861
X1064	1.734583
X1092	-0.336899
X1103	1.511997
X1109	0.293528
X1124	-1.070527
X1136	-0.832631
X1141	-3.810395
X1144	-1.634424
X1169	-2.272834
X1173	2.939901
X1179	0.255691
X1193	-1.943169
X1203	-1.285699
X1206	0.481157
X1208	0.659692
X1219	-0.794412
X1232	1.912179
X1264	-2.688033
X1272	2.000918
X1292	1.828815
X1297	2.590561
X1329	2.776266
X1351	1.696442
X1362	1.323481
X1416	-1.393758
X1417	-1.385110
X1418	-0.166155
X1430	11.202687
X1444	0.310800

	non_reg_linear
X1470	-2.449365
X1506	-0.038284
X1514	-4.044828
X1529	-1.410526
X1553	-0.437429
X1563	-3.958899
X1574	1.202832
X1595	-14.667350
X1597	1.154521
X1609	2.253494
X1616	13.448853
X1637	-2.940066
X1656	-1.568248
X1657	0.253881
X1683	0.668972

... and the intercept:

In [35]:
```
clf0.intercept_
```

Out[35]:
```
35.304938233261986
```

Train the LASSO Linear Regression

Next, let's train the LASSO linear regression model with just the training data. To perform LASSO linear regression, we use the **Lasso()** function. No additional parameters are required for this function.

- $\lambda = 0.3$
  - Note: With this Lasso() function, the α parameter actually equals the λ parameter that we defined in the slides
- max_iter=1000 = maximum number of iterations that the solver (Coordinate Descent Algorithm) will use if it doesn't converge.

In [36]:
```
clf1 = Lasso(alpha=0.3, max_iter=1000)
clf1.fit(X_train, y_train)
```

Out[36]:
```
Lasso(alpha=0.3)
```

In [37]:
```
pd.DataFrame(clf1.coef_.T, columns = ['lasso_linear'], index=X_train.columns)
```

Out[37]:

	lasso_linear
X159	0.000000
X960	0.901501

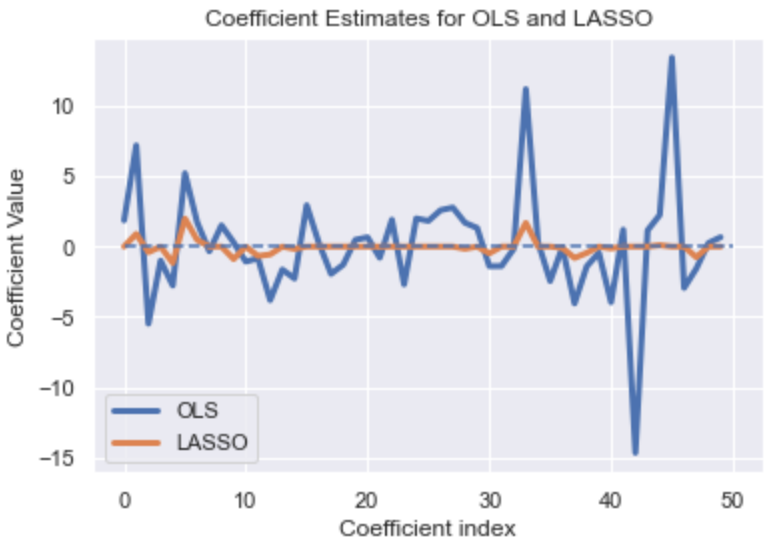
	lasso_linear
X980	-0.401145
X986	-0.000000
X1023	-1.167591
X1028	1.992447
X1064	0.476193
X1092	0.000000
X1103	0.000000
X1109	-0.887398
X1124	-0.000000
X1136	-0.672314
X1141	-0.559411
X1144	0.000000
X1169	-0.163310
X1173	0.000000
X1179	-0.000000
X1193	-0.000000
X1203	-0.000000
X1206	0.000000
X1208	0.000000
X1219	-0.000000
X1232	0.000000
X1264	0.000000
X1272	-0.000000
X1292	-0.000000
X1297	-0.000000
X1329	-0.000000
X1351	-0.155889
X1362	0.000000
X1416	-0.482222
X1417	-0.000000
X1418	0.000000
X1430	1.686456
X1444	0.000000
X1470	-0.000000
X1506	-0.095976
X1514	-0.809395
X1529	-0.453423
X1553	-0.000000

	lasso_linear
X1563	-0.124961
X1574	0.000000
X1595	-0.000000
X1597	-0.000000
X1609	0.102858
X1616	-0.000000
X1637	0.000000
X1656	-0.757361
X1657	0.000000
X1683	0.000000

Let's Compare the Coefficients from the Two Models

In [38]:

```
lw=3
plt.plot(np.arange(np.size(clf0.coef_)), clf0.coef_, lw=lw)
plt.plot(np.arange(np.size(clf1.coef_)), clf1.coef_, lw=lw)
#plt.xticks(np.arange(0,20,1))
plt.xlabel('Coefficient index')
plt.ylabel('Coefficient Value')
plt.title('Coefficient Estimates for OLS and LASSO')
plt.legend(['OLS','LASSO'], loc='lower left')
plt.hlines(y=0, xmin=0, xmax=50, linestyles='--')
plt.show()
```



In [39]:

```
pd.DataFrame({'non_reg_linear':clf0.coef_, 'lasso_linear': clf1.coef_}, index=X_train.colu
```

Out[39]:

	non_reg_linear	lasso_linear
X159	1.860688	0.000000
X960	7.197889	0.901501
X980	-5.483316	-0.401145
X986	-0.996936	-0.000000

	non_reg_linear	lasso_linear
X1023	-2.768050	-1.167591
X1028	5.217861	1.992447
X1064	1.734583	0.476193
X1092	-0.336899	0.000000
X1103	1.511997	0.000000
X1109	0.293528	-0.887398
X1124	-1.070527	-0.000000
X1136	-0.832631	-0.672314
X1141	-3.810395	-0.559411
X1144	-1.634424	0.000000
X1169	-2.272834	-0.163310
X1173	2.939901	0.000000
X1179	0.255691	-0.000000
X1193	-1.943169	-0.000000
X1203	-1.285699	-0.000000
X1206	0.481157	0.000000
X1208	0.659692	0.000000
X1219	-0.794412	-0.000000
X1232	1.912179	0.000000
X1264	-2.688033	0.000000
X1272	2.000918	-0.000000
X1292	1.828815	-0.000000
X1297	2.590561	-0.000000
X1329	2.776266	-0.000000
X1351	1.696442	-0.155889
X1362	1.323481	0.000000
X1416	-1.393758	-0.482222
X1417	-1.385110	-0.000000
X1418	-0.166155	0.000000
X1430	11.202687	1.686456
X1444	0.310800	0.000000
X1470	-2.449365	-0.000000
X1506	-0.038284	-0.095976
X1514	-4.044828	-0.809395
X1529	-1.410526	-0.453423
X1553	-0.437429	-0.000000
X1563	-3.958899	-0.124961
X1574	1.202832	0.000000

	non_reg_linear	lasso_linear
X1595	-14.667350	-0.000000
X1597	1.154521	-0.000000
X1609	2.253494	0.102858
X1616	13.448853	-0.000000
X1637	-2.940066	0.000000
X1656	-1.568248	-0.757361
X1657	0.253881	0.000000
X1683	0.668972	0.000000

We see that the non-regularized linear regression coefficient estimates are highly variable compared to the Lasso estimates. In addiition, the Lasso estimator zeroes out some of the coefficients. In other words, it has a variable selection property that can reduce the complexity of the trained model compared to ordinary least squares regression.

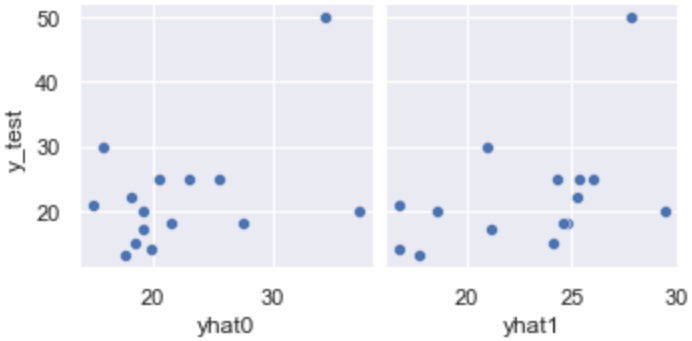
Next, let's test data predictions and performance scores of the two models

Having trained both model classes, we evaluate the relative performance using the .predict and .score methods. First, we copute the test data predictions based on the trained model.

In [40]:
```
yhat0 = clf0.predict(X_test)
yhat1 = clf1.predict(X_test)
```

How do these predictions *each individually* compare to the actual test values for y? Let's look at the scatter plots of actual versus predicted.

In [41]:
```
sns.pairplot(data=
             pd.DataFrame({'y_test': y_test,
                           'yhat0': yhat0,
                           'yhat1': yhat1}),
             y_vars='y_test', x_vars=['yhat0', 'yhat1'])
plt.show()
```



Compare R^2

Performance scores based on test data R-squared (proportion of variance explained) are computed using the .score(X,y) method. For further details see the documentation of sklearn.linear_model modules LinearRegression and Lasso.

In [42]:
```
print("Test R-square for OLS: ", clf0.score(X_test, y_test))
```

```
print("Test R-square for Lasso: ", clf1.score(X_test, y_test))
```

```
Test R-square for OLS:  0.1204157335536663
Test R-square for Lasso:  0.19483080965860788
```

Both are pretty low, but we see that the Lasso has substantially outperformed OLS based on the test R-square scores.

Importance of using an independent test set: a cautionary tale

It is instructive to compare the R-square values we would have gotten based on the training data.

"Naive" R^2 for OLS (ie. Using the Training Dataset R^2 for Evaluating Predictive Power)

In [43]:
```
round(clf0.score(X_train, y_train), 3)
```

Out[43]:
```
0.416
```

"Naive" R^2 for Lasso (ie. Using the Training Dataset R^2 for Evaluating Predictive Power)

In [44]:
```
round(clf1.score(X_train, y_train), 3)
```

Out[44]:
```
0.219
```

Conclusion: If we had relied on these in-sample evaluations alone, then OLS would have *appeared* to be much better than the Lasso, and *seemed* to explain over 41% of the variance in tumor size. Thus, we would have fooled ourselves badly!

Cross-Validation Analysis

In order to give every observation the chance to be in a test dataset at least once, we will use k-fold cross validation to compare the performance of the non-regularized linear regression model and the LASSO linear regression model. Specifically, we will use $k = 5$ folds in this analysis.

The **cross_val_score()** function from the **sklearn.model_selection** package will automatically do the following for us:

1. Split the dataset into k approximately equally sized folds.
2. Trains the model with each of the k training datasets created.
3. Tests the model with each of the k corresponding test datasets created.
4. After all the k fitted models has been tested, the function returns the **negative mean squared error** for each of the k corresponding test datasets.

$$-MSE_{test} = -\frac{1}{n_{test}} \sum_{i=1}^{n_{test}} (y_{test,i} - \hat{y}_{test,i})^2 \tag{1}$$

With this result we can take the average of these k $-MSE_{test}$ values to calculate the average performance of the given model.

Non-Regularized Linear Regression

Below we find that the average MSE of the $k = 5$ test dataset for the non-regularized linear regression model was 127.91.

In [45]:
```
from sklearn.model_selection import cross_val_score
modclass0 = LinearRegression()
scores0 = cross_val_score(modclass0, X, y, cv=5,
                          scoring="neg_mean_squared_error")
print("CV MSE scores:", -scores0)
print("Average MSE:", round(-scores0.mean(), 2))
print("Std. error:", round(scores0.std()/np.sqrt(len(scores0)), 2))
```

```
CV MSE scores: [224.87305851 125.45118196 117.67760614  79.61891434  91.90508918]
Average MSE: 127.91
Std. error: 22.93
```

LASSO Linear Regression

Below we find that the average MSE of the $k = 5$ test dataset for the LASSO linear regression model was 80.98.

In [46]:
```
modclass1 = Lasso(alpha=0.3, max_iter=1000)
scores1 = cross_val_score(modclass1, X, y, cv=5,
                          scoring="neg_mean_squared_error")
print("CV MSE scores:", -scores1)
print("Average MSE:", round(-scores1.mean(), 2))
print("Std. error:", round(scores1.std()/np.sqrt(len(scores1)), 2))
```

```
CV MSE scores: [ 52.00450159 105.81602909 104.74790139  72.53890058  69.78054464]
Average MSE: 80.98
Std. error: 9.42
```

Conclusion

We see that LASSO LINEAR regression has much better predictive accuracy in this example than the non-regularized linear regression model.

Estimated predictive accuracy

The average test MSE gives us a rough idea of how close the model predictions are expected to be to the actual values. For the Lasso, the estimated MSE was 81.0, which translates to a standard deviation for prediction of 9.0. We would therefore predict with approximate 95% confidence that the actual tumor size measurement would be within +/- 18.0 mm of the predicted value based on the gene expression measurements.

Comparing to the marginal distribution of tunor sizes above, this seems only a small gain in precision using these 50 gene expression values. It is unclear that tumor size is associated with these gene expressions.

Selecting the Value of λ in LASSO

To use a regularization method like the Lasso, we need to set the value of the tuning parameter that determines how much weight to give to the penalty function in fitting the model. how much impact does this have?

Example: Let's experiment with different values for λ in the breast cancer example, using gene expression features to predict tumor size.

Fit the models

First, let's fit multiple LASSO models using different values of λ.

In [47]:
```
# Several settings for alpha
mod000 = LinearRegression()
mod025 = Lasso(alpha=0.25, max_iter=1000)
mod050 = Lasso(alpha=0.50, max_iter=1000)
mod075 = Lasso(alpha=0.75, max_iter=1000)
mod100 = Lasso(alpha=1.00, max_iter=1000)
mod300 = Lasso(alpha=3.00, max_iter=1000)
```

In [48]:
```
# Fit the models
mod000.fit(X, y)
mod025.fit(X, y)
mod050.fit(X, y)
mod075.fit(X, y)
mod100.fit(X, y)
mod300.fit(X, y)
```

Out[48]:
```
Lasso(alpha=3.0)
```

Comparison

Let's compare the Number of Non-Zero Coefficients for each of these fitted models.

In [49]:
```
# compare numbers of nonzero coefficients
tol = 10**(-6)
perf_table = pd.DataFrame({
    'alpha': [0.00, 0.25, 0.50, 0.75, 1.00, 3.00],
    'nonzero_coefs': [
        np.sum(np.abs(mod000.coef_) > tol),
        np.sum(np.abs(mod025.coef_) > tol),
        np.sum(np.abs(mod050.coef_) > tol),
        np.sum(np.abs(mod075.coef_) > tol),
        np.sum(np.abs(mod100.coef_) > tol),
        np.sum(np.abs(mod300.coef_) > tol)]
    })
perf_table
```

Out[49]:

	alpha	nonzero_coefs
0	0.00	50
1	0.25	18
2	0.50	12
3	0.75	6
4	1.00	7
5	3.00	1

Is this what we would expect?

Perform 5-fold Cross Validation of Each Model and Compare

Let's perform 5-fold cross validation of each of these models and compare the average MSE of the 5 test dataset from each of these models.

In [50]:
```
# Compare cv scores
k=5
perf_table['MSE'] = [
```

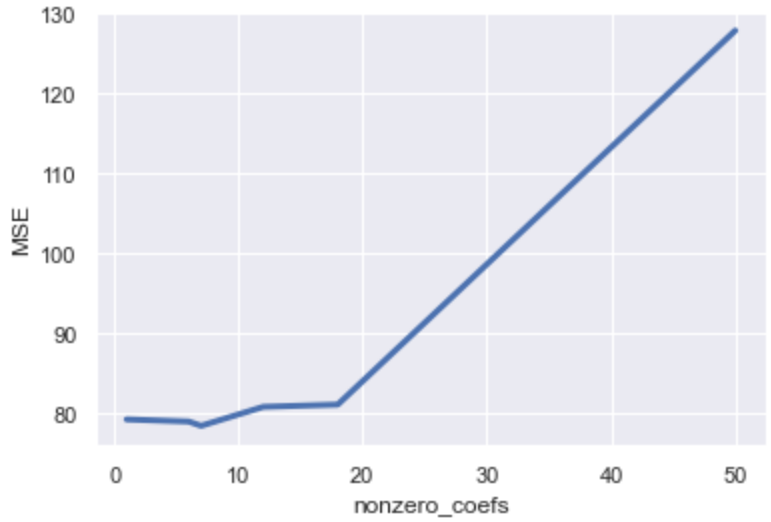
```
    -cross_val_score(mod000, X, y, cv=k, scoring="neg_mean_squared_error").mean(),
    -cross_val_score(mod025, X, y, cv=k, scoring="neg_mean_squared_error").mean(),
    -cross_val_score(mod050, X, y, cv=k, scoring="neg_mean_squared_error").mean(),
    -cross_val_score(mod075, X, y, cv=k, scoring="neg_mean_squared_error").mean(),
    -cross_val_score(mod100, X, y, cv=k, scoring="neg_mean_squared_error").mean(),
    -cross_val_score(mod300, X, y, cv=k, scoring="neg_mean_squared_error").mean()]
perf_table['RMSE'] = np.sqrt(np.abs(perf_table['MSE']))
perf_table
```

Out[50]:

	alpha	nonzero_coefs	MSE	RMSE
0	0.00	50	127.905170	11.309517
1	0.25	18	81.034117	9.001895
2	0.50	12	80.747123	8.985940
3	0.75	6	78.908112	8.883024
4	1.00	7	78.369937	8.852680
5	3.00	1	79.173781	8.897965

In [51]:

```
sns.lineplot(x='nonzero_coefs', y='MSE', data=perf_table, lw=3)
plt.show()
```



What does this mean?

Remarks:

- The penalty tuning parameter alpha has a big effect on the sparsity of the model; larger values of alpha place greater constraints on how many coefficients can be nonzero.
- Among these choices, alpha = 1.00, with 7 nonzero variables gives the smallest value for cross-validated mean-square-error.
- The differences on MSE are relatively small, as the root-mean-square error values range only from 8.9 to 9.0. Even the one-variable model does almost as well as the 18 variable model.
- All of the sparse Lasso fits improve on the OLS fit with 50 variables.

Automatic Parameter Tuning for LASSO

The **LassoCV()** function provides a tool for using cross-validation to compare performance across a range of values for the tuning parameter γ. The code here is adapted from code in the documentation for LassoCV.

Using this function below, we automatically try out the following values of γ in the LASSO linear regression model: 0.25, 0.50, 0.75, 1, 2, 3, 5, and 10, using 5-fold cross validation.

In [52]:
```
from sklearn.linear_model import LassoCV
model = LassoCV(cv=5, alphas=[0.25, 0.50, 0.75, 1.00, 2.00, 3.00, 5, 10]).fit(X, y)

model
```

Out[52]:
```
LassoCV(alphas=[0.25, 0.5, 0.75, 1.0, 2.0, 3.0, 5, 10], cv=5)
```

In [53]:
```
model.alphas_
```

Out[53]:
```
array([10.  ,  5.  ,  3.  ,  2.  ,  1.  ,  0.75,  0.5 ,  0.25])
```

In [54]:
```
pd.DataFrame(model.mse_path_, index=model.alphas_, columns=['test_fold_%s_avg_MSE'%(str(i)
```

Out[54]:

	test_fold_1_avg_MSE	test_fold_2_avg_MSE	test_fold_3_avg_MSE	test_fold_4_avg_MSE	test_fold_5_a
10.00	67.365625	82.228958	117.933333	74.850000	6(
5.00	67.365625	82.003717	114.438824	74.434986	5
3.00	67.365625	82.141190	111.999843	74.529572	5
2.00	67.365625	83.651007	110.887078	74.824178	6
1.00	57.687258	89.637655	108.114968	74.811516	6
0.75	60.193665	90.998073	106.865448	73.685921	6
0.50	66.513379	94.457836	103.851655	73.327269	6
0.25	48.226059	109.744121	105.332595	70.760787	7

Let's display the cross-validation average MSE for each of these LASSO linear regression models below.

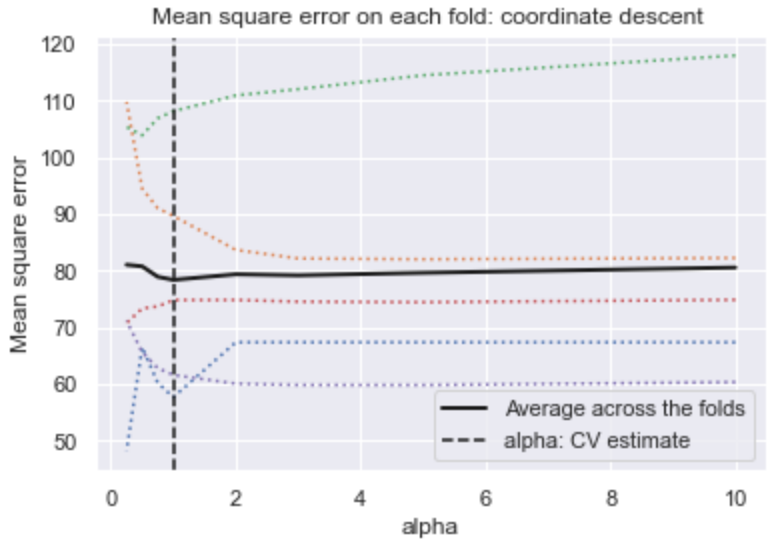
In [55]:
```
# Display results
plt.figure()

#Plot the MSE for each value of lambda (there are five "folds" (ie. pairs of training/tes
plt.plot(model.alphas_, model.mse_path_, ':')

#Plot the average MSE for each value of lambda (across all five folds)
plt.plot(model.alphas_, model.mse_path_.mean(axis=-1), 'k',
         label='Average across the folds', linewidth=2)

#Plot a vertical line for the alpha that produces the minimum average MSE (across all the
plt.axvline(model.alpha_, linestyle='--', color='k',
            label='alpha: CV estimate')
plt.legend()
plt.xlabel('alpha')
plt.ylabel('Mean square error')
plt.title('Mean square error on each fold: coordinate descent ')
plt.axis('tight')
```

Out[55]:
```
(-0.23750000000000004, 10.4875, 44.74069525897193, 121.41869705116007)
```



What do you observe?

Conclusion: The optimum value is estimated to be near $\gamma = \alpha = 1$. We can also see that the train/test MSE paths for each "fold", that is for each of the 5 train/test splits of the data. The large variation between optimal alpha values between different folds suggests that the best alpha is highly variable. We can't put too much confidence in the best value for these data. Conversely, this means that the choice of alpha is not critical within the range considered.

Displaying the results of the final model

Given that this value of $\alpha = \gamma = 1$ had the lowest average MSE, the model chooses LASSO linear regression with $\alpha = \gamma = 1$ as the **final model**.

In [56]:
```
model.alpha_
```

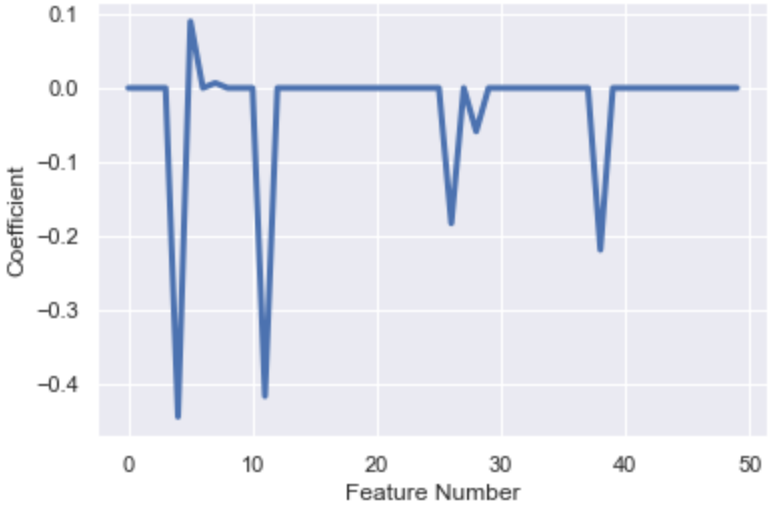
Out[56]:
```
1.0
```

We can extract the slopes of the final LASSO model using the **.coef_** attribute as shown below.

In [57]:
```
model.coef_
```

Out[57]:
```
array([-0.        ,  0.        ,  0.        , -0.        , -0.4449253 ,
        0.08993896,  0.        ,  0.00678522, -0.        , -0.        ,
        0.        , -0.41695898, -0.        , -0.        , -0.        ,
       -0.        , -0.        , -0.        , -0.        , -0.        ,
       -0.        , -0.        , -0.        ,  0.        ,  0.        ,
        0.        , -0.18317027, -0.        , -0.05873826, -0.        ,
       -0.        ,  0.        ,  0.        ,  0.        ,  0.        ,
       -0.        ,  0.        , -0.        , -0.21900174,  0.        ,
       -0.        ,  0.        ,  0.        , -0.        ,  0.        ,
        0.        ,  0.        , -0.        ,  0.        ,  0.        ])
```

In [58]:
```
plt.plot(np.arange(np.size(model.coef_)), model.coef_, lw=lw)
plt.xlabel('Feature Number')
plt.ylabel('Coefficient')
plt.show()
```



In [59]:
```
# Which gene expressions had nonzero coefficients?
tol = 10**(-8)
X.columns[abs(model.coef_)>tol]
```

Out[59]:
```
Index(['X1023', 'X1028', 'X1092', 'X1136', 'X1297', 'X1351', 'X1529'], dtype='object')
```

References

Chanrion, M. et al. "A Gene Expression Signature that Can Predict the Recurrence of Tamoxifen-Treated Primary Breast Cancer." *Clin. Cancer. Res.* 2008;14(6)March15, 2008

STAT 207, Julie Deeke, Victoria Ellison, and Douglas Simpson, University of Illinois at Urbana-Champaign