week5_lecture

July 7, 2020

1 Modules

To carry out statistical tests in Python, we will be using an external module called SciPy, and to perform statistical modelling we will use the **ols** function from the external module statsmodels. We also need to import numpy and pandas, because we will be analysing data stored in pandas dataframes. Finally, we will import some plotnine modules to allow us to visualise the data that we are analysing.

[2]:
```
import scipy as sp
from statsmodels.formula.api import ols
from statsmodels.api import Logit
import numpy as np
import pandas as pd
from plotnine import ggplot, aes, theme, geom_histogram, geom_point,␣
↪geom_violin, geom_boxplot, element_text, facet_wrap
```

2 Data

To demonstrate the data analysis functionality of Python, we will use the metabric dataset. Some of the functions we will use do not handle missing data, so we will remove any rows for the dataset where data is missing. As we saw in week 3, we can use the **describe()** method to generate summary statistics for this dataset:

[4]:
```
metabric = pd.read_csv("../data/metabric_clinical_and_expression_data.csv").
↪dropna()
metabric.describe()
```

[4]:

	Cohort	Age_at_diagnosis	Survival_time	Tumour_size	\
count	1121.000000	1121.000000	1121.000000	1121.000000	
mean	2.207850	60.412721	126.239518	26.094112	
std	0.956449	13.012218	77.295543	15.102221	
min	1.000000	21.930000	0.100000	1.000000	
25%	1.000000	50.820000	60.133333	17.000000	
50%	2.000000	60.930000	116.433333	22.000000	
75%	3.000000	69.700000	188.733333	30.000000	
max	5.000000	96.290000	337.033333	180.000000	

```
       Tumour_stage  Neoplasm_histologic_grade  Lymph_nodes_examined_positive  \
count   1121.000000                1121.000000                    1121.000000
mean       1.756467                   2.445138                       1.873327
std        0.622865                   0.635888                       3.830332
min        1.000000                   1.000000                       0.000000
25%        1.000000                   2.000000                       0.000000
50%        2.000000                   3.000000                       0.000000
75%        2.000000                   3.000000                       2.000000
max        4.000000                   3.000000                      41.000000

       Lymph_node_status  Nottingham_prognostic_index  Mutation_count  \
count        1121.000000                  1121.000000     1121.000000
mean            1.626227                     4.123553        5.467440
std             0.739443                     1.059818        3.859249
min             1.000000                     2.002000        1.000000
25%             1.000000                     3.052000        3.000000
50%             1.000000                     4.046000        5.000000
75%             2.000000                     5.046000        7.000000
max             3.000000                     6.360000       46.000000

              ESR1        ERBB2          PGR         TP53       PIK3CA  \
count  1121.000000  1121.000000  1121.000000  1121.000000  1121.000000
mean      9.600854    10.770958     6.238728     6.191980     5.950108
std       2.093524     1.317631     1.020860     0.389334     0.310095
min       5.217238     7.281883     4.945672     5.225320     5.158697
25%       8.205776     9.981831     5.422349     5.936286     5.730861
50%      10.220349    10.532638     5.864217     6.176018     5.931565
75%      11.202333    11.149977     6.902124     6.439989     6.134401
max      13.265184    14.643900     9.932115     7.769900     8.708396

             GATA3        FOXA1         MLPH
count  1121.000000  1121.000000  1121.000000
mean      9.530585    10.839721    11.383495
std       1.468576     1.687979     1.630174
min       5.401414     5.289602     5.323652
25%       8.809316    10.878608    11.071585
50%       9.917441    11.365047    11.857401
75%      10.554370    11.749098    12.374549
max      12.812082    13.127682    14.432001
```

3 Statistical tests

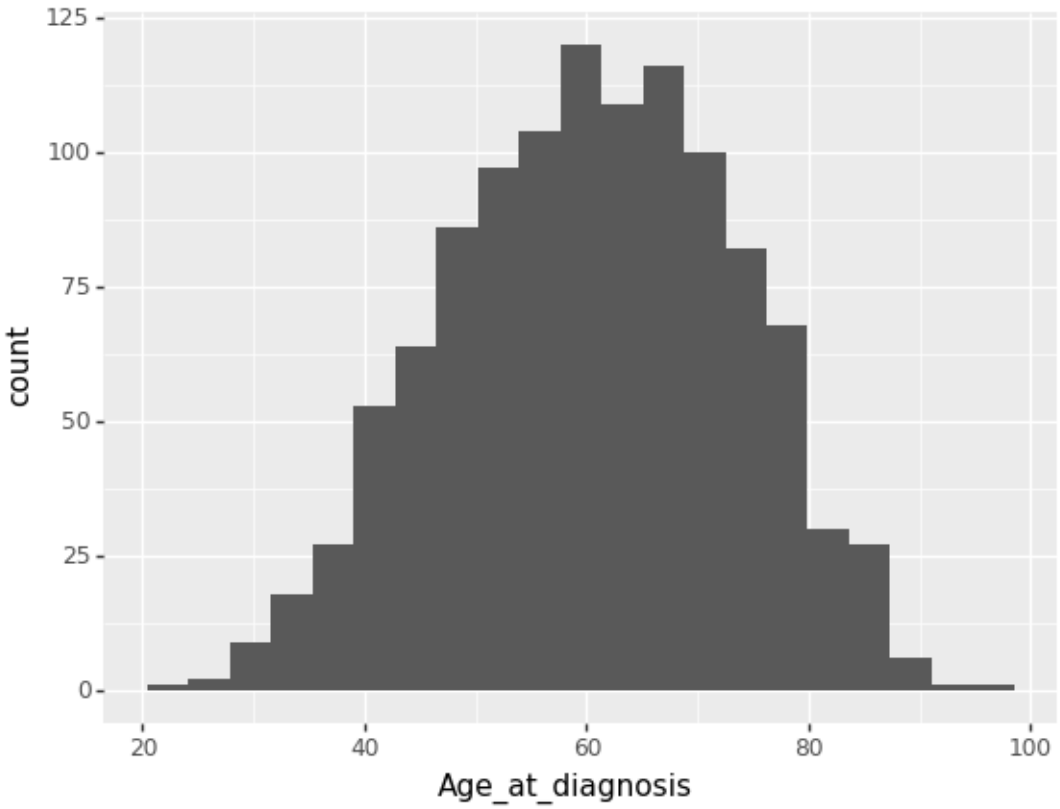
3.1 Tests for normality

When we are deciding which statistical test to use in our analysis, we often need to work out whether the data follows a normal distribution or not, as some tests (e.g. t-test) assume that our data are normally distributed. We can test whether a dataset follows a normal distribution by using

the Kolmogorov-Smirnov test. For example, the age at diagnosis looks like it could be normally distributed:

[5]:
```
(
ggplot(metabric, aes("Age_at_diagnosis"))
    + geom_histogram()
)
```

```
C:\ProgramData\Anaconda3\lib\site-packages\plotnine\stats\stat_bin.py:93:
PlotnineWarning: 'stat_bin()' using 'bins = 21'. Pick better value with
'binwidth'.
  warn(msg.format(params['bins']), PlotnineWarning)
```



[5]:
```
<ggplot: (-9223371879137589260)>
```

To run the Kolmogorov-Smirnov test, we use the **kstest()** function from the scipy stats module:

[53]:
```
sp.stats.kstest(metabric["Age_at_diagnosis"], "norm")
```

[53]:
```
KstestResult(statistic=1.0, pvalue=0.0)
```

The Kolmogorov-Smirnov test has a p value below 0.05, indicating that we can reject the null-

hypothesis that there is no difference between this distribution and a normal distribution. In other words, the distribution appears non-normal.

In SciPy, the results of most tests are returned as an object. When printed directly to screen this is not very pretty and hard to interpret, as we can see above. When running the test, we can assign the results object to a variable, and then access the attributes of the results object to print the results in a clearer format:

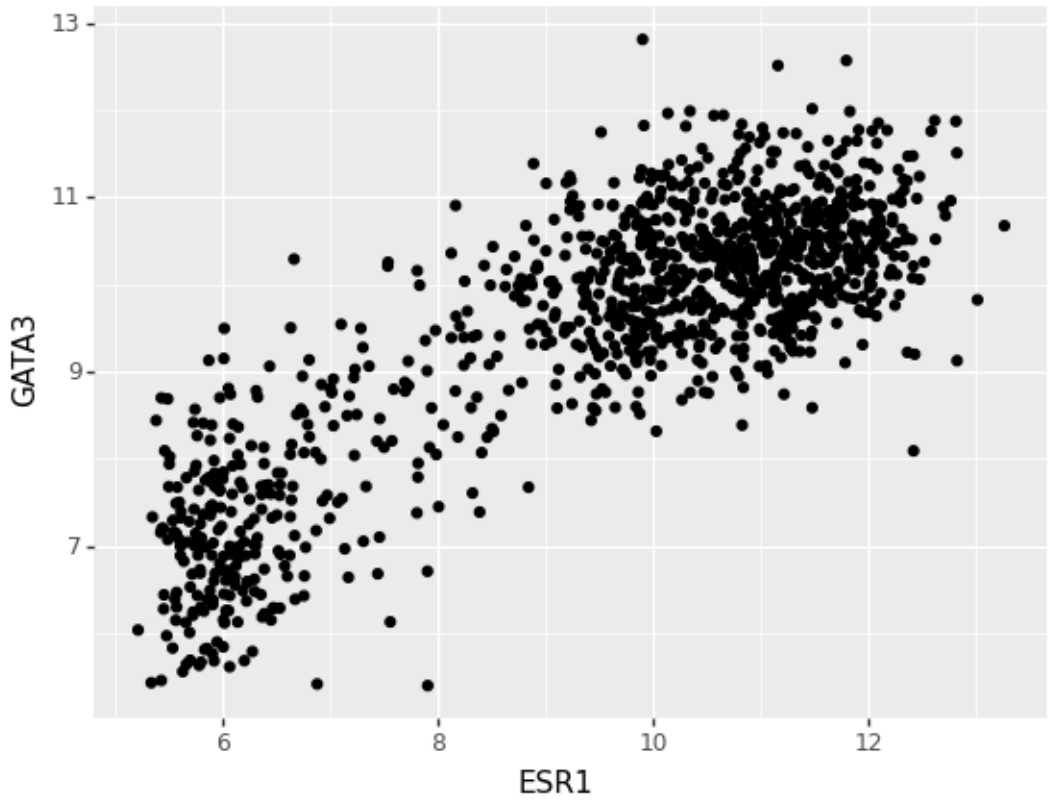
[59]:
```
# run the test and assign the result to a variable
age_diagnosis_ks = sp.stats.kstest(metabric["Age_at_diagnosis"], "norm")
# print the results by retrieving attributes from the result object
print("Age at diagnosis Kolmogorov-Smirnov test:")
print("p value = {}".format(age_diagnosis_ks.pvalue))
```

```
Age at diagnosis Kolmogorov-Smirnov test:
p value = 0.0
```

3.2 Correlation

We often want to test whether two continuous variables are related to each other, and we can do this by calculating a correlation. For example, there appears to be a relationship between the expression of the ESR1 gene and the GATA3 gene:

[3]:
```
(
ggplot(metabric, aes("ESR1", "GATA3"))
    + geom_point()
)
```



[3]: <ggplot: (-9223371866336140352)>

For normally distributed data, we can use calculate the Pearson's correlation using the **pearsonr()** function. **pearsonr()** returns the results as a tuple rather than an object, so we need to access the coefficient and p value using indexing:

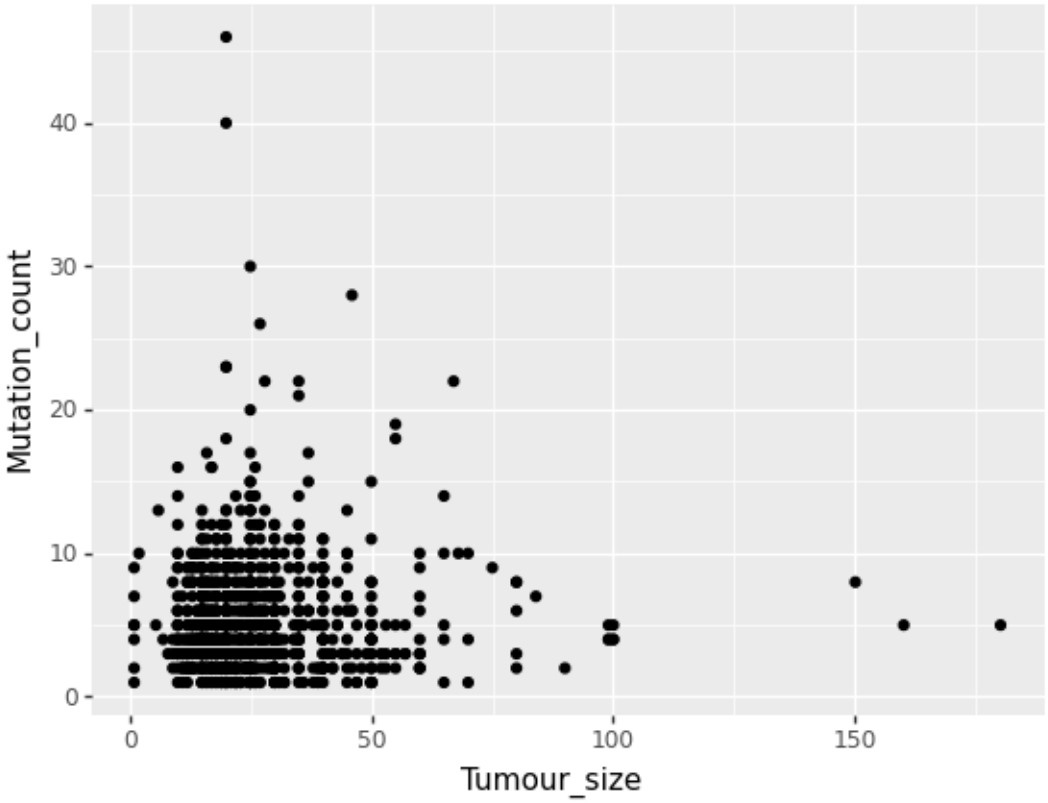
[57]:
```
ESR1_GATA3_pearson = sp.stats.pearsonr(metabric["ESR1"], metabric["GATA3"])
print("Pearson correlation between ESR1 & GATA3:")
print("coefficient = {}".format(ESR1_GATA3_pearson[0]))
print("p value = {}".format(ESR1_GATA3_pearson[1]))
```

```
Pearson correlation between ESR1 & GATA3:
coefficient = 0.8282016709899257
p value = 1.134210930536034e-283
```

For data that is not normally distributed, we can calculate the Spearman rank correlation. For example, a scatter plot of tumour size versus mutation count suggests that these are not normally distributed:

[5]:
```
(
ggplot(metabric, aes("Tumour_size", "Mutation_count"))
    + geom_point()
```

```
)
```



[5]: <ggplot: (-9223371894293519400)>

We can calculate the Spearman rank correlation using the **spearmanr()** function, again accessing the results using indexing:

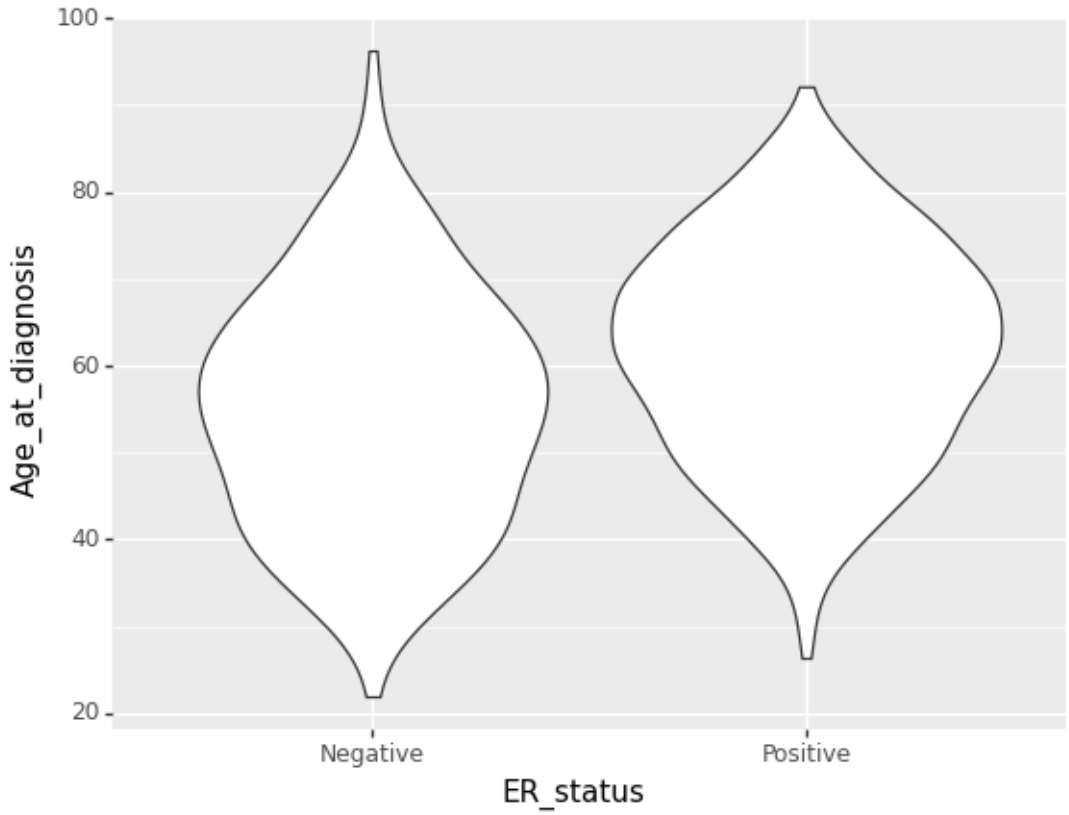
[58]:
```
size_mutation_spearman = sp.stats.spearmanr(metabric["Tumour_size"],␣
↪metabric["Mutation_count"])
print("Spearman rank correlation between tumour size and mutation count:")
print("coefficient = {}".format(size_mutation_spearman[0]))
print("p value = {}".format(size_mutation_spearman[1]))
```

```
Spearman rank correlation between tumour size and mutation count:
coefficient = 0.0070724805027381175
p value = 0.8130176386734044
```

3.3 T-test

To test whether the mean value of a continuous variable is significantly different between two different groups, we can use the t-test for normally distributed data. For example, age at diagnosis appears to be lower for ER-negative tumours compared with ER-positive tumours:

[42]:
```
(
ggplot(metabric, aes("ER_status", "Age_at_diagnosis"))
    + geom_violin()
)
```



[42]:
```
<ggplot: (-9223371879135783076)>
```

We can use the **ttest_ind()** function to carry out the t test, which confirms that we can reject the null hypothesis that age at diagnosis is not different between ER positive and negative tumours. Note that **ttest_ind()** takes two arguments, which are arrays containing the values of the two groups. Rather than extracting these values and assigning them to separate variables, we can do the data extraction within the function call:

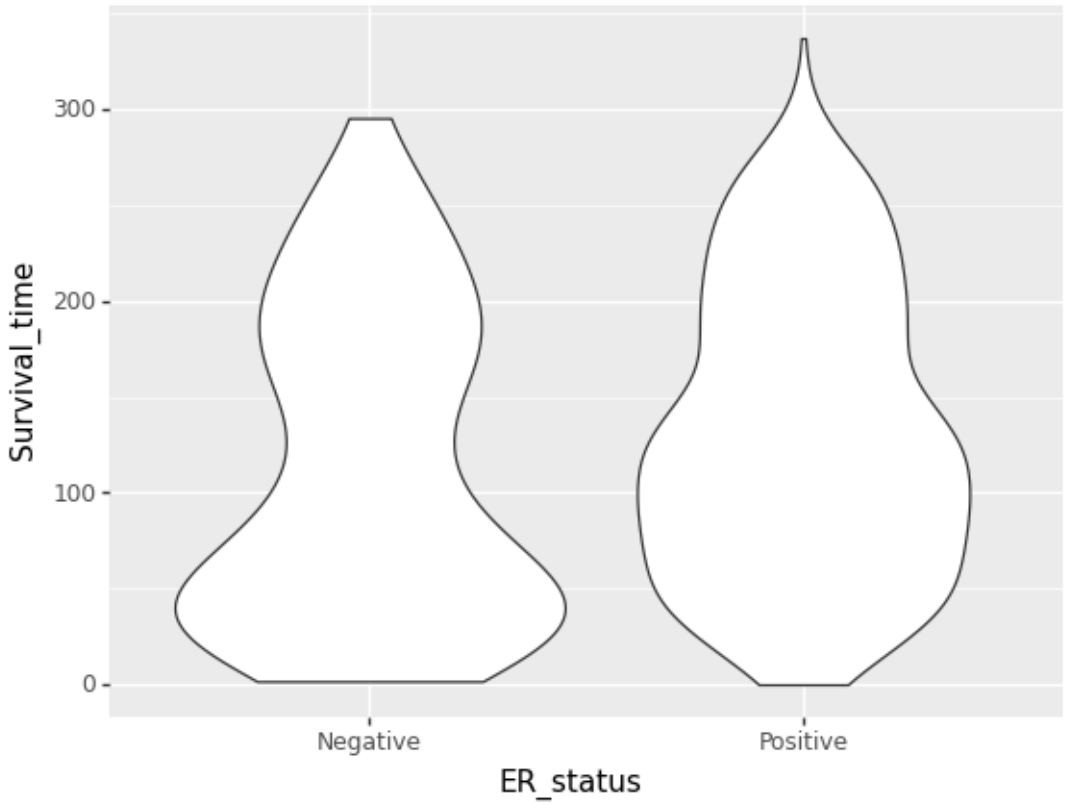
[60]:
```
ER_age_t = sp.stats.ttest_ind(
    # select samples with Negative ER_status and extract the Age_at_diagnosis␣
 ↪values
    metabric[metabric["ER_status"]=="Negative"]["Age_at_diagnosis"],
    # select samples with Positive ER_status and extract the Age_at_diagnosis␣
 ↪values
    metabric[metabric["ER_status"]=="Positive"]["Age_at_diagnosis"]
)
```

```
print("t test of age at diagnosis for ER_status Negative vs Positive:")
print("t = {}".format(ER_age_t.statistic))
print("p = {}".format(ER_age_t.pvalue))
```

```
t test of age at diagnosis for ER_status Negative vs Positive:
t = -7.543060668278905
p = 9.471433611351617e-14
```

If we have data that is not normally distributed we may want to use the Mann-Whitney U test, also known as the Wilcoxon rank-sum test, which is the non-parametric equivalent of the t test. For example, survival time does not follow a normal distribution, but it still appears to be different between ER positive and ER negative tumours:

[46]:
```
(
ggplot(metabric, aes("ER_status", "Survival_time"))
    + geom_violin()
)
```



[46]: <ggplot: (-9223371879137560864)>

We can use the **mannwhitneyu()** function to run the Mann-Whitney U test, which confirms that we can reject the null hypothesis that age at diagnosis is not different between ER positive and

negative tumours. Again, we are subsetting and selecting the data within the function call:

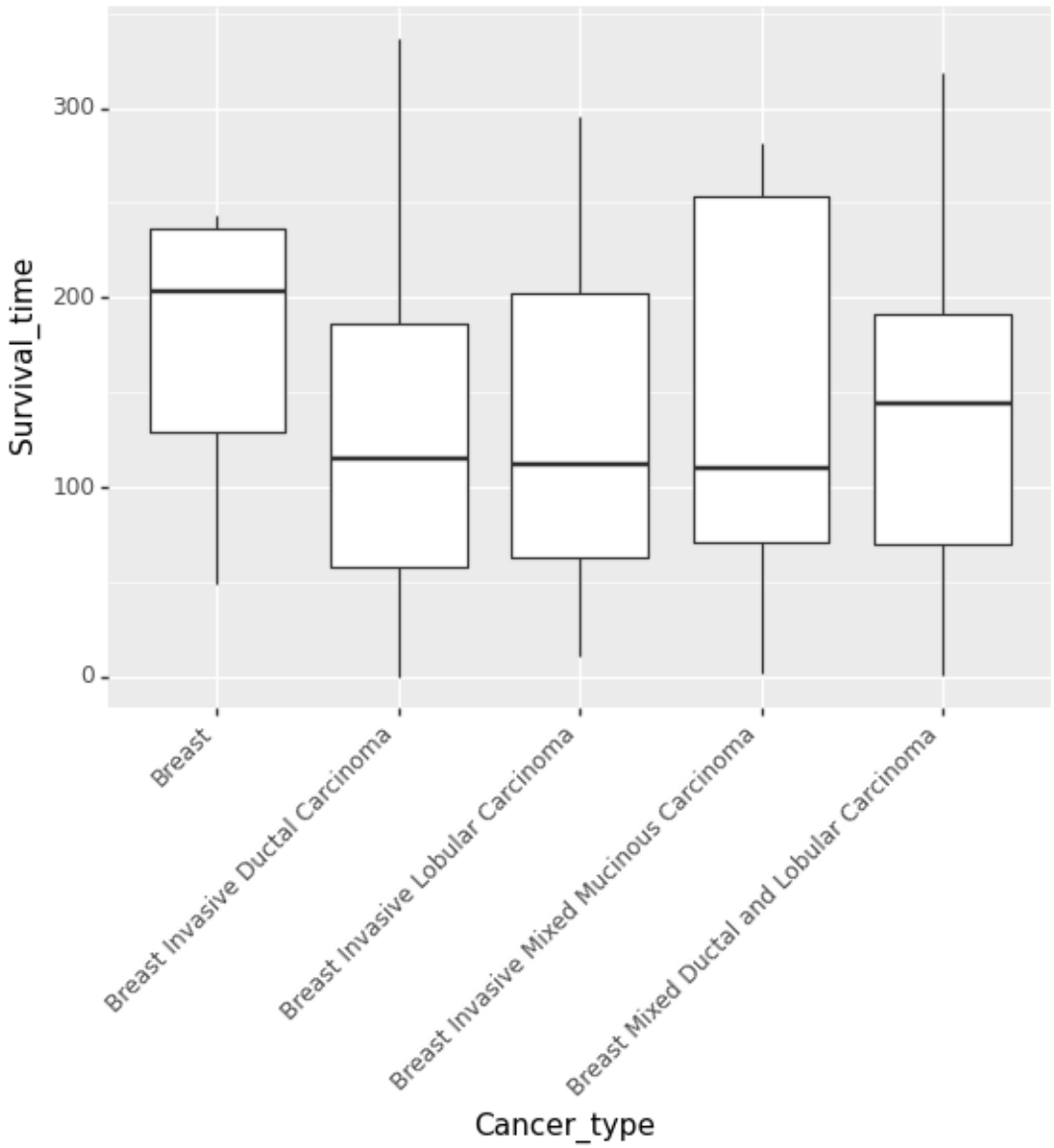
[61]:
```
ER_survival_MWU = sp.stats.mannwhitneyu(
    # select samples with Negative ER_status and extract the Age_at_diagnosis␣
 ↪values
    metabric[metabric["ER_status"]=="Negative"]["Age_at_diagnosis"],
    # select samples with Positive ER_status and extract the Age_at_diagnosis␣
 ↪values
    metabric[metabric["ER_status"]=="Positive"]["Age_at_diagnosis"]
)
print("Mann-Whitney U test of survival time for ER_status Negative vs Positive:
 ↪")
print("f = {}".format(ER_survival_MWU.statistic))
print("p = {}".format(ER_age_t.pvalue))
```

```
Mann-Whitney U test of survival time for ER_status Negative vs Positive:
f = 77360.0
p = 9.471433611351617e-14
```

3.4 ANOVA

If we want to test for a difference in the mean value of a continuous variable between >2 groups simultaneously, we can use the analysis of variance (ANOVA). For example, we may want to test for differences between survival times between different cancer types, which appear to be different:

[70]:
```
(
ggplot(metabric, aes("Cancer_type", "Survival_time"))
    + geom_boxplot()
    + theme(axis_text_x = element_text(angle=45, hjust=1))
)
```



[70]: <ggplot: (-9223371879136834176)>

We can use the **f_oneway()** function to run ANOVA, which shows that we cannot reject the null hypothesis that there is no difference in survival time between cancer types:

[78]:
```
type_survival_anova = sp.stats.f_oneway(
    # select samples with Breast cancer and extract the Survival_time values
    metabric[metabric["Cancer_type"]=="Breast"]["Survival_time"],
    # select samples with Breast cancer and extract the Survival_time values
    metabric[metabric["Cancer_type"]=="Breast Invasive Ductal␣
↪Carcinoma"]["Survival_time"],
```

```
    # select samples with Breast cancer and extract the Survival_time values
    metabric[metabric["Cancer_type"]=="Breast Invasive Lobular␣
 ↪Carcinoma"]["Survival_time"]
)
print("ANOVA of survival time for different cancer types:")
print("f = {}".format(type_survival_anova.statistic))
print("p = {}".format(type_survival_anova.pvalue))
```

```
ANOVA of survival time for different cancer types:
f = 1.3798530485937566
p = 0.25211254882000633
```

3.5 Chi-square

If we have two categorical variables of interest, and we want to test whether the status of one variable is linked to the status of the other, we can use the Chi-square test. For example, we may want to test whether the ER status of a tumour (Positive or Negative) is linked to the PR status (Positive or Negative). First, we need to format the data into a contingency table, containing counts of positive and negative values for ER and PR:

[104]:
```
# count the number of tumours for each ER-PR status combination
ERpos_PRpos = metabric[(metabric["ER_status"]=="Positive") &␣
 ↪(metabric["PR_status"]=="Positive")].shape[0]
ERpos_PRneg = metabric[(metabric["ER_status"]=="Positive") &␣
 ↪(metabric["PR_status"]=="Negative")].shape[0]
ERneg_PRpos = metabric[(metabric["ER_status"]=="Negative") &␣
 ↪(metabric["PR_status"]=="Positive")].shape[0]
ERneg_PRneg = metabric[(metabric["ER_status"]=="Negative") &␣
 ↪(metabric["PR_status"]=="Negative")].shape[0]
# make a contingency table for counts
ER_PR_contingency = [
    [ERpos_PRpos, ERpos_PRneg],
    [ERneg_PRpos, ERneg_PRneg]
]
ER_PR_contingency
```

[104]:
```
[[573, 296], [11, 241]]
```

Now, we use the **chi2_contingency()** function to run the Chi-square test, and assign the results to a variable. This shows that we can reject the null hypothesis that ER and PR status are independent. The results are returned as a tuple rather than an object, so we retrieve them by using indexing:

[105]:
```
ER_PR_chi = sp.stats.chi2_contingency(ER_PR_contingency)
print("Chi-square test for ER and PR status:")
print("Chi-square value = {}".format(ER_PR_chi[0]))
print("p value = {}".format(ER_PR_chi[1]))
```

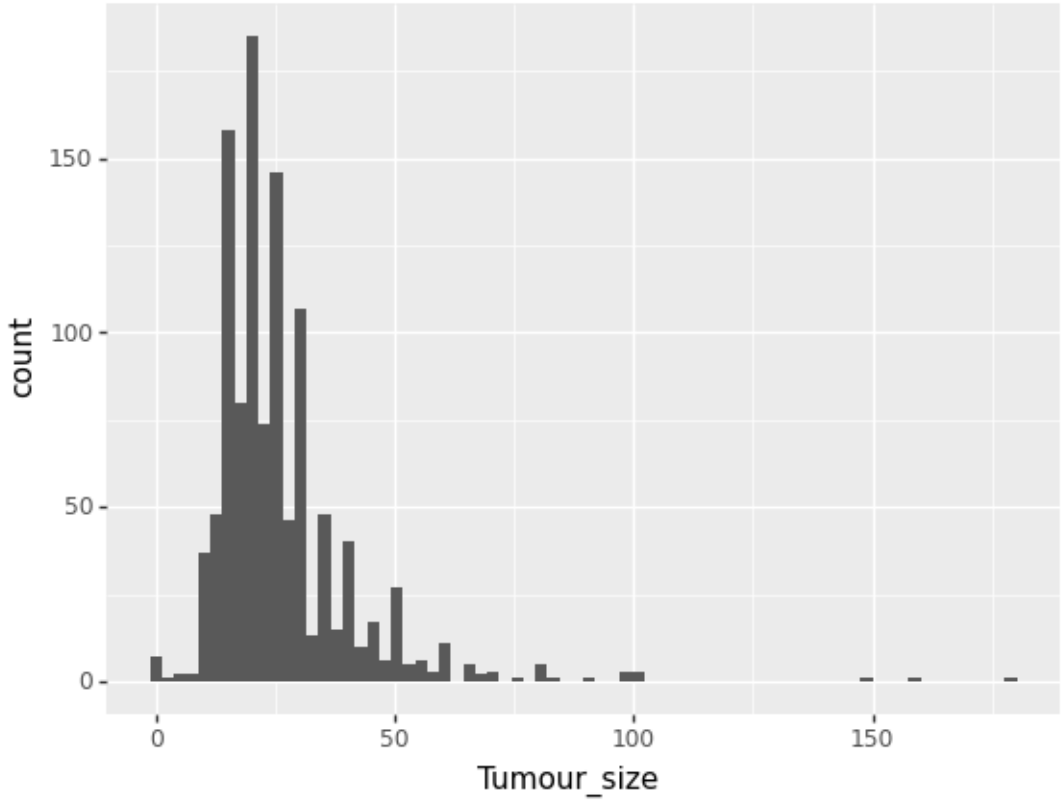
```
Chi-square test for ER and PR status:
Chi-square value = 294.3053708325672
p value = 5.73427299280106e-66
```

4 Data Transformation

When working with large datasets, we often have variables with very different ranges and distributions of values. For some analyses, particularly statistical modelling, it is helpful to be able to apply a mathematical transformation to a set of values, which rescales the values and makes their distribution and range more similar to other variables in the dataset. For example, in the Metabric dataset the distribution of tumour sizes is highly left-skewed, as most tumours are small but a few are very large:

[10]:
```
(
ggplot(metabric, aes("Tumour_size"))
    + geom_histogram()
)
```

```
C:\ProgramData\Anaconda3\lib\site-packages\plotnine\stats\stat_bin.py:93:
PlotnineWarning: 'stat_bin()' using 'bins = 72'. Pick better value with
'binwidth'.
  warn(msg.format(params['bins']), PlotnineWarning)
```



[10]:
```
<ggplot: (-9223371900156157056)>
```

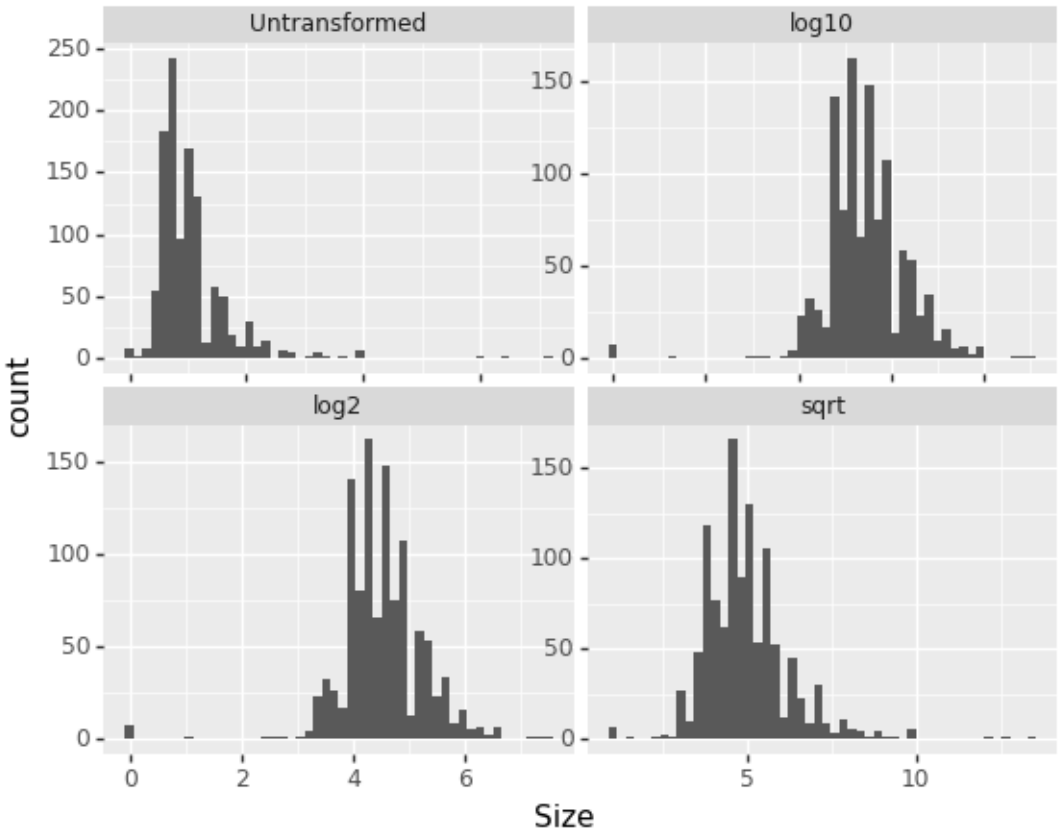
To perform transformations on this data, we can use some functions from numpy: - **sqrt()** = square-root transform - **log2()** = log-transform with base 2 - **log10()** = log-transform with base 10

All of these functions return a numpy array of transformed values. To retain the original (untransformed) data, we can add these transformed values to the metabric dataframe as a new column:

[36]:
```
metabric["Tumour_size_sqrt"] = np.sqrt(metabric["Tumour_size"])
metabric["Tumour_size_log2"] = np.log2(metabric["Tumour_size"])
metabric["Tumour_size_log10"] = np.log10(metabric["Tumour_size"])
```

After transformation, the tumour sizes look much closer to being normally distributed:

[36]:
```
# select the variables of interest
tumour_size_tranformations = metabric.loc[:,["Patient_ID", "Tumour_size",␣
 ↪"Tumour_size_sqrt", "Tumour_size_log2", "Tumour_size_log10"]]
# rename the columns for ease of plotting
tumour_size_tranformations.columns = ["Patient_ID", "Untransformed", "sqrt",␣
 ↪"log2", "log10"]
# reformat the untransformed and transformed data into three columns ahead of␣
 ↪plotting
tumour_size_tranformations = pd.melt(tumour_size_tranformations,␣
 ↪id_vars="Patient_ID", var_name="Transformation", value_name="Size")
# plot faceted histogram
(
ggplot(tumour_size_tranformations, aes("Size"))
    + facet_wrap("~Transformation", nrow=2, scales="free")
    + geom_histogram(bins=50)
)
```



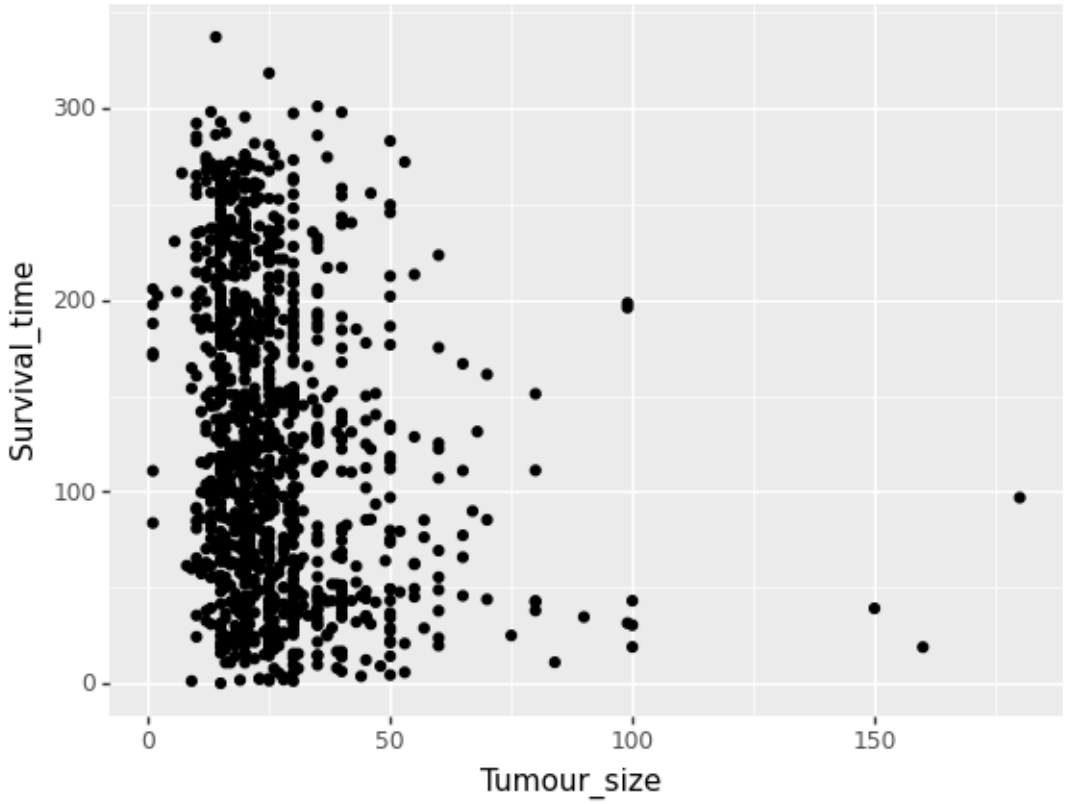
[36]: <ggplot: (-9223371900154646860)>

5 Modelling

5.1 Simple linear regression

If we have a continuous variable, and we want to model its relationship with another variable, we can use simple linear regression. In linear regression we call the variable of interest the **response**, and the other variable the **predictor**. The mathematical details of linear regression are beyond the scope of this course, but in the case of simple linear regression it basically amounts to fitting a line through the data that is closest to all of the points. For example, we may want to predict survival time based on tumour size, because survival time appears to differ across the range of tumour sizes:

[82]:
```
(
ggplot(metabric, aes("Tumour_size", "Survival_time"))
    + geom_point()
)
```



[82]: <ggplot: (-9223371917719325692)>

In Python, we can run simple linear regression using the **ols** function from the **statsmodels** package. There are three steps to completing this analysis: 1. Instantiate the model: create an object that holds the model specification and the input dataset. In the model specification, the response is to the left of the tilda (~) and the predictor is to the right 2. Fit the model: fit the specified model to the data and assign the results object to a variable 3. Display the results: use the **summary()** method of the results object to return a detailed breakdown of the model characteristic

[87]:
```
# instantiate model
simple_model = ols("Survival_time~Tumour_size", data=metabric)
# fit the model
simple_results = simple_model.fit()
# display the results
simple_results.summary()
```

[87]:
```
<class 'statsmodels.iolib.summary.Summary'>
"""
                            OLS Regression Results
==============================================================================
Dep. Variable:          Survival_time   R-squared:                       0.054
```

```
Model:                            OLS   Adj. R-squared:                  0.053
Method:                 Least Squares   F-statistic:                     63.87
Date:                Fri, 19 Jun 2020   Prob (F-statistic):           3.29e-15
Time:                        13:29:14   Log-Likelihood:                -6432.7
No. Observations:                1121   AIC:                         1.287e+04
Df Residuals:                    1119   BIC:                         1.288e+04
Df Model:                           1
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
Intercept    157.2724      4.486     35.057      0.000     148.470     166.075
Tumour_size   -1.1893      0.149     -7.992      0.000      -1.481      -0.897
==============================================================================
Omnibus:                      159.724   Durbin-Watson:                   1.741
Prob(Omnibus):                  0.000   Jarque-Bera (JB):               57.181
Skew:                           0.328   Prob(JB):                     3.83e-13
Kurtosis:                       2.109   Cond. No.                         60.3
==============================================================================

Warnings:
[1] Standard Errors assume that the covariance matrix of the errors is correctly
specified.
"""
```

The model summary contains a lot of detailed information, but we can create a more concise report of the results by extracting the results of interest e.g. the r2 value, the F-statistic and its p value:

[105]:
```
print("Simple linear regression: Survival_time~Tumour_size")
print("r2 = {}".format(simple_results.rsquared))
print("F-statistic = {}".format(simple_results.fvalue))
print("F-statistic p value= {}".format(simple_results.f_pvalue))
```

```
Simple linear regression: Survival_time~Tumour_size
r2 = 0.053992380252458894
F-statistic = 63.86573664028733
F-statistic p value= 3.294337572421415e-15
```

After fitting a linear regression model, we usually want to carry out some basic checks of the model characteristics. This is because linear regression makes some assumptions about the data and our model, and if the data that we have fitted our model to has violated these assumptions, then the predictions from the model may not be reliable. We will not cover these checks in this session as they are beyond the scope of the course, but if you want information on how to do this then please see the statsmodels documentation.

If we are happy with the checks of model characteristics, we can use the model to predict what the value of our response variable will be, given a certain value for the predictor variable. We do this using the **predict()** method of the results object, which takes the value of the predictor variable as the argument:

[106]:
```
simple_results.predict({"Tumour_size": 125})
```

[106]:
```
0    8.613872
dtype: float64
```

Our model predicts a survival time of 8.6 for a tumour size of 125; however, the low r2 value for this model (r2=0.053) indicates that it fits the data very poorly, so we may not be very confident in this prediction.

5.2 Multivariate linear regression

When we are analysing more complex processes, we often need to consider the influence of multiple predictors simultaneously. One way to do this is by using multivariate linear regression, which models the relationship between the response and two or more predictors. For example, we may wish to model the effect on survival time of tumour size, tumour stage, cancer type and ER status. To do this we repeat the simple regression process described above, but specify multiple predictors when instantiating the model. When viewing the results, we extract the *pvalues* attribute of the results object to print the p values associated with each predictor:

[11]:
```
# instantiate model
complex_model = ols("Survival_time~Tumour_size + Tumour_stage + Cancer_type +␣
 ↪ER_status", data=metabric)
# fit the model
complex_results = complex_model.fit()
# print the results of interest
print("Complex linear regression: Survival_time~Tumour_size + Tumour_stage +␣
 ↪Cancer_type + ER_status")
print("r2 = {}".format(complex_results.rsquared))
print("F-statistic = {}".format(complex_results.fvalue))
print("F-statistic p value= {}".format(complex_results.f_pvalue))
print("p values for each predictor:")
print(complex_results.pvalues)
```

```
Complex linear regression: Survival_time~Tumour_size + Tumour_stage +
Cancer_type + ER_status
r2 = 0.09815028798654712
F-statistic = 17.304319757467745
F-statistic p value= 7.505040904761116e-22
p values for each predictor:
Intercept                                                  2.696922e-13
Cancer_type[T.Breast Invasive Ductal Carcinoma]            7.754587e-02
Cancer_type[T.Breast Invasive Lobular Carcinoma]           1.418438e-01
Cancer_type[T.Breast Invasive Mixed Mucinous Carcinoma]    2.226865e-01
Cancer_type[T.Breast Mixed Ductal and Lobular Carcinoma]   1.512336e-01
ER_status[T.Positive]                                      3.584975e-02
Tumour_size                                                6.982111e-04
Tumour_stage                                               3.162942e-10
dtype: float64
```

Including these extra predictors has almost doubled the r2, but the model fit is still quite poor (r2=0.098). Given the complexity of breast cancer biology and the relative simplicity of our analysis, this isn't a big surprise!

6 Exercises

6.1 Exercise 1

t test

6.2 Exercise 2

correlation with and without transformation

6.3 Exercise 3

chi squared test