logistic-regression

August 14, 2020

1 Logistic Regression

Hello again! You are going to implement a logistic regression classifer in this Jupyter notebook using scikit-learn and predict using it. We will also see a technique which is useful for visualizing the data.

1.1 Before you start

- In order for the notebooks to function as intended, modify only between lines marked "### begin your code here (__ lines)." and "### end your code here.".
- The line count is a suggestion of how many lines of code you need to accomplish what is asked.
- You should execute the cells (the boxes that a notebook is composed of) in order.
- You can execute a cell by pressing Shift and Enter (or Return) simultaneously.
- You should have completed the previous Jupyter notebooks before attempting this one as the concepts covered there are not repeated, for the sake of brevity.

1.2 Loading the appropriate packages

Nothing new here. We will import logistic regression class along with some helpers from scikit-learn.

[1]:
```
import numpy as np
from sklearn.linear_model import LogisticRegression
from sklearn.decomposition import PCA
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
from sklearn.preprocessing import LabelEncoder
import pandas as pd
import plotly.express as px
import plotly.graph_objs as go
```

Let's turn off the scientific notation for floating point numbers.

[2]:
```
np.set_printoptions(suppress=True)
```

1.3 Loading and examining the data

We will load our data from a CSV file and put it in a pandas an object of the `DataFrame` class.

This dataset is the breast cancer Wisconsin (diagnostic) dataset which contains 30 different features computed from a images of a fine needle aspirate (FNA) of breast masses for 569 patients with each example labeled as being a *benign* or *malignant* mass.

- This was taken and modified from the Machine Learning dataset repository of School of Information and Computer Science of University of California Irvine (UCI):

  Dua, D. and Graff, C. (2019). UCI Machine Learning Repository [http://archive.ics.uci.edu/ml]. Irvine, CA: University of California, School of Information and Computer Science.

[3]:
```
df_30 = pd.read_csv('data_logistic_regression.csv')
```

Let's take a look at the data:

[4]:
```
df_30
```

[4]:
```
    mean radius  mean texture  mean perimeter  mean area  mean smoothness  \
0        17.990         10.38          122.80     1001.0          0.11840
1        20.570         17.77          132.90     1326.0          0.08474
2        19.690         21.25          130.00     1203.0          0.10960
3        11.420         20.38           77.58      386.1          0.14250
4        20.290         14.34          135.10     1297.0          0.10030
5        12.450         15.70           82.57      477.1          0.12780
6        18.250         19.98          119.60     1040.0          0.09463
7        13.710         20.83           90.20      577.9          0.11890
8        13.000         21.82           87.50      519.8          0.12730
9        12.460         24.04           83.97      475.9          0.11860
10       16.020         23.24          102.70      797.8          0.08206
11       15.780         17.89          103.60      781.0          0.09710
12       19.170         24.80          132.40     1123.0          0.09740
13       15.850         23.95          103.70      782.7          0.08401
14       13.730         22.61           93.60      578.3          0.11310
15       14.540         27.54           96.73      658.8          0.11390
16       14.680         20.13           94.74      684.5          0.09867
17       16.130         20.68          108.10      798.8          0.11700
18       19.810         22.15          130.00     1260.0          0.09831
19       13.540         14.36           87.46      566.3          0.09779
20       13.080         15.71           85.63      520.0          0.10750
21        9.504         12.44           60.34      273.9          0.10240
22       15.340         14.26          102.50      704.4          0.10730
23       21.160         23.04          137.20     1404.0          0.09428
24       16.650         21.38          110.00      904.6          0.11210
25       17.140         16.40          116.00      912.7          0.11860
26       14.580         21.53           97.41      644.8          0.10540
27       18.610         20.25          122.10     1094.0          0.09440
28       15.300         25.27          102.40      732.4          0.10820
29       17.570         15.05          115.00      955.1          0.09847
```

..	...	...	...	...	...
539	7.691	25.44	48.34	170.4	0.08668
540	11.540	14.44	74.65	402.9	0.09984
541	14.470	24.99	95.81	656.4	0.08837
542	14.740	25.42	94.70	668.6	0.08275
543	13.210	28.06	84.88	538.4	0.08671
544	13.870	20.70	89.77	584.8	0.09578
545	13.620	23.23	87.19	573.2	0.09246
546	10.320	16.35	65.31	324.9	0.09434
547	10.260	16.58	65.85	320.8	0.08877
548	9.683	19.34	61.05	285.7	0.08491
549	10.820	24.21	68.89	361.6	0.08192
550	10.860	21.48	68.51	360.5	0.07431
551	11.130	22.44	71.49	378.4	0.09566
552	12.770	29.43	81.35	507.9	0.08276
553	9.333	21.94	59.01	264.0	0.09240
554	12.880	28.92	82.50	514.3	0.08123
555	10.290	27.61	65.67	321.4	0.09030
556	10.160	19.59	64.73	311.7	0.10030
557	9.423	27.88	59.26	271.3	0.08123
558	14.590	22.68	96.39	657.1	0.08473
559	11.510	23.93	74.52	403.5	0.09261
560	14.050	27.15	91.38	600.4	0.09929
561	11.200	29.37	70.67	386.0	0.07449
562	15.220	30.62	103.40	716.9	0.10480
563	20.920	25.09	143.00	1347.0	0.10990
564	21.560	22.39	142.00	1479.0	0.11100
565	20.130	28.25	131.20	1261.0	0.09780
566	16.600	28.08	108.30	858.1	0.08455
567	20.600	29.33	140.10	1265.0	0.11780
568	7.760	24.54	47.92	181.0	0.05263

	mean compactness	mean concavity	mean concave points	mean symmetry \
0	0.27760	0.300100	0.147100	0.2419
1	0.07864	0.086900	0.070170	0.1812
2	0.15990	0.197400	0.127900	0.2069
3	0.28390	0.241400	0.105200	0.2597
4	0.13280	0.198000	0.104300	0.1809
5	0.17000	0.157800	0.080890	0.2087
6	0.10900	0.112700	0.074000	0.1794
7	0.16450	0.093660	0.059850	0.2196
8	0.19320	0.185900	0.093530	0.2350
9	0.23960	0.227300	0.085430	0.2030
10	0.06669	0.032990	0.033230	0.1528
11	0.12920	0.099540	0.066060	0.1842
12	0.24580	0.206500	0.111800	0.2397
13	0.10020	0.099380	0.053640	0.1847

14	0.22930	0.212800	0.080250	0.2069
15	0.15950	0.163900	0.073640	0.2303
16	0.07200	0.073950	0.052590	0.1586
17	0.20220	0.172200	0.102800	0.2164
18	0.10270	0.147900	0.094980	0.1582
19	0.08129	0.066640	0.047810	0.1885
20	0.12700	0.045680	0.031100	0.1967
21	0.06492	0.029560	0.020760	0.1815
22	0.21350	0.207700	0.097560	0.2521
23	0.10220	0.109700	0.086320	0.1769
24	0.14570	0.152500	0.091700	0.1995
25	0.22760	0.222900	0.140100	0.3040
26	0.18680	0.142500	0.087830	0.2252
27	0.10660	0.149000	0.077310	0.1697
28	0.16970	0.168300	0.087510	0.1926
29	0.11570	0.098750	0.079530	0.1739
..	...	...	...	...
539	0.11990	0.092520	0.013640	0.2037
540	0.11200	0.067370	0.025940	0.1818
541	0.12300	0.100900	0.038900	0.1872
542	0.07214	0.041050	0.030270	0.1840
543	0.06877	0.029870	0.032750	0.1628
544	0.10180	0.036880	0.023690	0.1620
545	0.06747	0.029740	0.024430	0.1664
546	0.04994	0.010120	0.005495	0.1885
547	0.08066	0.043580	0.024380	0.1669
548	0.05030	0.023370	0.009615	0.1580
549	0.06602	0.015480	0.008160	0.1976
550	0.04227	0.000000	0.000000	0.1661
551	0.08194	0.048240	0.022570	0.2030
552	0.04234	0.019970	0.014990	0.1539
553	0.05605	0.039960	0.012820	0.1692
554	0.05824	0.061950	0.023430	0.1566
555	0.07658	0.059990	0.027380	0.1593
556	0.07504	0.005025	0.011160	0.1791
557	0.04971	0.000000	0.000000	0.1742
558	0.13300	0.102900	0.037360	0.1454
559	0.10210	0.111200	0.041050	0.1388
560	0.11260	0.044620	0.043040	0.1537
561	0.03558	0.000000	0.000000	0.1060
562	0.20870	0.255000	0.094290	0.2128
563	0.22360	0.317400	0.147400	0.2149
564	0.11590	0.243900	0.138900	0.1726
565	0.10340	0.144000	0.097910	0.1752
566	0.10230	0.092510	0.053020	0.1590
567	0.27700	0.351400	0.152000	0.2397
568	0.04362	0.000000	0.000000	0.1587

	mean fractal dimension	...	worst texture	worst perimeter	worst area	\
0	0.07871	...	17.33	184.60	2019.0	
1	0.05667	...	23.41	158.80	1956.0	
2	0.05999	...	25.53	152.50	1709.0	
3	0.09744	...	26.50	98.87	567.7	
4	0.05883	...	16.67	152.20	1575.0	
5	0.07613	...	23.75	103.40	741.6	
6	0.05742	...	27.66	153.20	1606.0	
7	0.07451	...	28.14	110.60	897.0	
8	0.07389	...	30.73	106.20	739.3	
9	0.08243	...	40.68	97.65	711.4	
10	0.05697	...	33.88	123.80	1150.0	
11	0.06082	...	27.28	136.50	1299.0	
12	0.07800	...	29.94	151.70	1332.0	
13	0.05338	...	27.66	112.00	876.5	
14	0.07682	...	32.01	108.80	697.7	
15	0.07077	...	37.13	124.10	943.2	
16	0.05922	...	30.88	123.40	1138.0	
17	0.07356	...	31.48	136.80	1315.0	
18	0.05395	...	30.88	186.80	2398.0	
19	0.05766	...	19.26	99.70	711.2	
20	0.06811	...	20.49	96.09	630.5	
21	0.06905	...	15.66	65.13	314.9	
22	0.07032	...	19.08	125.10	980.9	
23	0.05278	...	35.59	188.00	2615.0	
24	0.06330	...	31.56	177.00	2215.0	
25	0.07413	...	21.40	152.40	1461.0	
26	0.06924	...	33.21	122.40	896.9	
27	0.05699	...	27.26	139.90	1403.0	
28	0.06540	...	36.71	149.30	1269.0	
29	0.06149	...	19.52	134.90	1227.0	
..	...	...	...	...	...	
539	0.07751	...	31.89	54.49	223.6	
540	0.06782	...	19.68	78.78	457.8	
541	0.06341	...	31.73	113.50	808.9	
542	0.05680	...	32.29	107.40	826.4	
543	0.05781	...	37.17	92.48	629.6	
544	0.06688	...	24.75	99.17	688.6	
545	0.05801	...	29.09	97.58	729.8	
546	0.06201	...	21.77	71.12	384.9	
547	0.06714	...	22.04	71.08	357.4	
548	0.06235	...	25.59	69.10	364.2	
549	0.06328	...	31.45	83.90	505.6	
550	0.05948	...	24.77	74.08	412.3	
551	0.06552	...	28.26	77.80	436.6	
552	0.05637	...	36.00	88.10	594.7	

553	0.06576	...	25.05	62.86	295.8
554	0.05708	...	35.74	88.84	595.7
555	0.06127	...	34.91	69.57	357.6
556	0.06331	...	22.88	67.88	347.3
557	0.06059	...	34.24	66.50	330.6
558	0.06147	...	27.27	105.90	733.5
559	0.06570	...	37.16	82.28	474.2
560	0.06171	...	33.17	100.20	706.7
561	0.05502	...	38.30	75.19	439.6
562	0.07152	...	42.79	128.70	915.0
563	0.06879	...	29.41	179.10	1819.0
564	0.05623	...	26.40	166.10	2027.0
565	0.05533	...	38.25	155.00	1731.0
566	0.05648	...	34.12	126.70	1124.0
567	0.07016	...	39.42	184.60	1821.0
568	0.05884	...	30.37	59.16	268.6

	worst smoothness	worst compactness	worst concavity	\
0	0.16220	0.66560	0.71190	
1	0.12380	0.18660	0.24160	
2	0.14440	0.42450	0.45040	
3	0.20980	0.86630	0.68690	
4	0.13740	0.20500	0.40000	
5	0.17910	0.52490	0.53550	
6	0.14420	0.25760	0.37840	
7	0.16540	0.36820	0.26780	
8	0.17030	0.54010	0.53900	
9	0.18530	1.05800	1.10500	
10	0.11810	0.15510	0.14590	
11	0.13960	0.56090	0.39650	
12	0.10370	0.39030	0.36390	
13	0.11310	0.19240	0.23220	
14	0.16510	0.77250	0.69430	
15	0.16780	0.65770	0.70260	
16	0.14640	0.18710	0.29140	
17	0.17890	0.42330	0.47840	
18	0.15120	0.31500	0.53720	
19	0.14400	0.17730	0.23900	
20	0.13120	0.27760	0.18900	
21	0.13240	0.11480	0.08867	
22	0.13900	0.59540	0.63050	
23	0.14010	0.26000	0.31550	
24	0.18050	0.35780	0.46950	
25	0.15450	0.39490	0.38530	
26	0.15250	0.66430	0.55390	
27	0.13380	0.21170	0.34460	
28	0.16410	0.61100	0.63350	

29	0.12550	0.28120	0.24890
..	...	...	...
539	0.15960	0.30640	0.33930
540	0.13450	0.21180	0.17970
541	0.13400	0.42020	0.40400
542	0.10600	0.13760	0.16110
543	0.10720	0.13810	0.10620
544	0.12640	0.20370	0.13770
545	0.12160	0.15170	0.10490
546	0.12850	0.08842	0.04384
547	0.14610	0.22460	0.17830
548	0.11990	0.09546	0.09350
549	0.12040	0.16330	0.06194
550	0.10010	0.07348	0.00000
551	0.10870	0.17820	0.15640
552	0.12340	0.10640	0.08653
553	0.11030	0.08298	0.07993
554	0.12270	0.16200	0.24390
555	0.13840	0.17100	0.20000
556	0.12650	0.12000	0.01005
557	0.10730	0.07158	0.00000
558	0.10260	0.31710	0.36620
559	0.12980	0.25170	0.36300
560	0.12410	0.22640	0.13260
561	0.09267	0.05494	0.00000
562	0.14170	0.79170	1.17000
563	0.14070	0.41860	0.65990
564	0.14100	0.21130	0.41070
565	0.11660	0.19220	0.32150
566	0.11390	0.30940	0.34030
567	0.16500	0.86810	0.93870
568	0.08996	0.06444	0.00000

	worst concave points	worst symmetry	worst fractal dimension	type
0	0.26540	0.4601	0.11890	malignant
1	0.18600	0.2750	0.08902	malignant
2	0.24300	0.3613	0.08758	malignant
3	0.25750	0.6638	0.17300	malignant
4	0.16250	0.2364	0.07678	malignant
5	0.17410	0.3985	0.12440	malignant
6	0.19320	0.3063	0.08368	malignant
7	0.15560	0.3196	0.11510	malignant
8	0.20600	0.4378	0.10720	malignant
9	0.22100	0.4366	0.20750	malignant
10	0.09975	0.2948	0.08452	malignant
11	0.18100	0.3792	0.10480	malignant
12	0.17670	0.3176	0.10230	malignant

13	0.11190	0.2809	0.06287	malignant
14	0.22080	0.3596	0.14310	malignant
15	0.17120	0.4218	0.13410	malignant
16	0.16090	0.3029	0.08216	malignant
17	0.20730	0.3706	0.11420	malignant
18	0.23880	0.2768	0.07615	malignant
19	0.12880	0.2977	0.07259	benign
20	0.07283	0.3184	0.08183	benign
21	0.06227	0.2450	0.07773	benign
22	0.23930	0.4667	0.09946	malignant
23	0.20090	0.2822	0.07526	malignant
24	0.20950	0.3613	0.09564	malignant
25	0.25500	0.4066	0.10590	malignant
26	0.27010	0.4264	0.12750	malignant
27	0.14900	0.2341	0.07421	malignant
28	0.20240	0.4027	0.09876	malignant
29	0.14560	0.2756	0.07919	malignant
..	...	...	...	...
539	0.05000	0.2790	0.10660	benign
540	0.06918	0.2329	0.08134	benign
541	0.12050	0.3187	0.10230	benign
542	0.10950	0.2722	0.06956	benign
543	0.07958	0.2473	0.06443	benign
544	0.06845	0.2249	0.08492	benign
545	0.07174	0.2642	0.06953	benign
546	0.02381	0.2681	0.07399	benign
547	0.08333	0.2691	0.09479	benign
548	0.03846	0.2552	0.07920	benign
549	0.03264	0.3059	0.07626	benign
550	0.00000	0.2458	0.06592	benign
551	0.06413	0.3169	0.08032	benign
552	0.06498	0.2407	0.06484	benign
553	0.02564	0.2435	0.07393	benign
554	0.06493	0.2372	0.07242	benign
555	0.09127	0.2226	0.08283	benign
556	0.02232	0.2262	0.06742	benign
557	0.00000	0.2475	0.06969	benign
558	0.11050	0.2258	0.08004	benign
559	0.09653	0.2112	0.08732	benign
560	0.10480	0.2250	0.08321	benign
561	0.00000	0.1566	0.05905	benign
562	0.23560	0.4089	0.14090	malignant
563	0.25420	0.2929	0.09873	malignant
564	0.22160	0.2060	0.07115	malignant
565	0.16280	0.2572	0.06637	malignant
566	0.14180	0.2218	0.07820	malignant
567	0.26500	0.4087	0.12400	malignant

```
568                    0.00000          0.2871                  0.07039     benign

[569 rows x 31 columns]
```

For this example to be educational, we need to be able to visualize our data, so our data has to be 2-dimensional. However, our data here is 30 dimensional. Let us use a trick (that we can use for many things including visualizations) to get 2-dimensional data out of this dataset.

Remember we talked about *unsupervised learning* in course 1. We said that *representation learning*, the methods use to create representations of the data (which are hopefully helping us to dod machine learning more efficiently) are a subclass of unsupervised learning methods. Specifically, we said that *dimensionality reduction* are a set of representation learning algorithms aimed at, as the name suggests, reducing the dimensionality of our data. We are going to use a very popular dimensionality reduction technique, called the *Principal Components Analysis* (*PCA*) to reduce the dimensioanlity of our feature space down to 2, so we can visualize our data in 3D plots.

Note that we can not only expand our feature space by adding features, for exmample, non-linear feature expansions, but also transform features and get new ones and we are doing exactly that with PCA. We are taking all of the features and constructing the two features that are *a.* a linear combination of our features; and *b.* are most informative in spreading out the data. In other words, with PCA, we construct two features from our original features where in these new features, the data points are most spread out and varied, among all features we can construct out of linearly combining our original features.

To do that we first need to extract our data, from the dataframe, in NumPy arrays:

[5]:
```
X_30 = df_30.drop('type', axis=1).to_numpy()
y_text = df_30['type'].to_numpy()
```

As a sanity check, let's check `X_30`:

[6]:
```
X_30
```

[6]:
```
array([[ 17.99    ,  10.38    , 122.8     , ...,   0.2654 ,   0.4601 ,
          0.1189 ],
       [ 20.57    ,  17.77    , 132.9     , ...,   0.186  ,   0.275  ,
          0.08902],
       [ 19.69    ,  21.25    , 130.      , ...,   0.243  ,   0.3613 ,
          0.08758],
       ...,
       [ 16.6     ,  28.08    , 108.3     , ...,   0.1418 ,   0.2218 ,
          0.0782 ],
       [ 20.6     ,  29.33    , 140.1     , ...,   0.265  ,   0.4087 ,
          0.124  ],
       [  7.76    ,  24.54    ,  47.92    , ...,   0.     ,   0.2871 ,
          0.07039]])
```

...and the size:

[7]:
```
X_30.shape
```

[7]:
```
(569, 30)
```

Let's do the same thing for `y_text`:

[8]:
```
y_text
```

[8]:
```
array(['malignant', 'malignant', 'malignant', 'malignant', 'malignant',
       'malignant', 'malignant', 'malignant', 'malignant', 'malignant',
       'malignant', 'malignant', 'malignant', 'malignant', 'malignant',
       'malignant', 'malignant', 'malignant', 'malignant', 'benign',
       'benign', 'benign', 'malignant', 'malignant', 'malignant',
       'malignant', 'malignant', 'malignant', 'malignant', 'malignant',
       'malignant', 'malignant', 'malignant', 'malignant', 'malignant',
       'malignant', 'malignant', 'benign', 'malignant', 'malignant',
       'malignant', 'malignant', 'malignant', 'malignant', 'malignant',
       'malignant', 'benign', 'malignant', 'benign', 'benign', 'benign',
       'benign', 'benign', 'malignant', 'malignant', 'benign',
       'malignant', 'malignant', 'benign', 'benign', 'benign', 'benign',
       'malignant', 'benign', 'malignant', 'malignant', 'benign',
       'benign', 'benign', 'benign', 'malignant', 'benign', 'malignant',
       'malignant', 'benign', 'malignant', 'benign', 'malignant',
       'malignant', 'benign', 'benign', 'benign', 'malignant',
       'malignant', 'benign', 'malignant', 'malignant', 'malignant',
       'benign', 'benign', 'benign', 'malignant', 'benign', 'benign',
       'malignant', 'malignant', 'benign', 'benign', 'benign',
       'malignant', 'malignant', 'benign', 'benign', 'benign', 'benign',
       'malignant', 'benign', 'benign', 'malignant', 'benign', 'benign',
       'benign', 'benign', 'benign', 'benign', 'benign', 'benign',
       'malignant', 'malignant', 'malignant', 'benign', 'malignant',
       'malignant', 'benign', 'benign', 'benign', 'malignant',
       'malignant', 'benign', 'malignant', 'benign', 'malignant',
       'malignant', 'benign', 'malignant', 'malignant', 'benign',
       'benign', 'malignant', 'benign', 'benign', 'malignant', 'benign',
       'benign', 'benign', 'benign', 'malignant', 'benign', 'benign',
       'benign', 'benign', 'benign', 'benign', 'benign', 'benign',
       'benign', 'malignant', 'benign', 'benign', 'benign', 'benign',
       'malignant', 'malignant', 'benign', 'malignant', 'benign',
       'benign', 'malignant', 'malignant', 'benign', 'benign',
       'malignant', 'malignant', 'benign', 'benign', 'benign', 'benign',
       'malignant', 'benign', 'benign', 'malignant', 'malignant',
       'malignant', 'benign', 'malignant', 'benign', 'malignant',
       'benign', 'benign', 'benign', 'malignant', 'benign', 'benign',
       'malignant', 'malignant', 'benign', 'malignant', 'malignant',
       'malignant', 'malignant', 'benign', 'malignant', 'malignant',
       'malignant', 'benign', 'malignant', 'benign', 'malignant',
       'benign', 'benign', 'malignant', 'benign', 'malignant',
       'malignant', 'malignant', 'malignant', 'benign', 'benign',
       'malignant', 'malignant', 'benign', 'benign', 'benign',
       'malignant', 'benign', 'benign', 'benign', 'benign', 'benign',
       'malignant', 'malignant', 'benign', 'benign', 'malignant',
       'benign', 'benign', 'malignant', 'malignant', 'benign',
       'malignant', 'benign', 'benign', 'benign', 'benign', 'malignant',
       'benign', 'benign', 'benign', 'benign', 'benign', 'malignant',
```

```
'benign', 'malignant', 'malignant', 'malignant', 'malignant',
'malignant', 'malignant', 'malignant', 'malignant', 'malignant',
'malignant', 'malignant', 'malignant', 'malignant', 'malignant',
'benign', 'benign', 'benign', 'benign', 'benign', 'benign',
'malignant', 'benign', 'malignant', 'benign', 'benign',
'malignant', 'benign', 'benign', 'malignant', 'benign',
'malignant', 'malignant', 'benign', 'benign', 'benign', 'benign',
'benign', 'benign', 'benign', 'benign', 'benign', 'benign',
'benign', 'benign', 'benign', 'malignant', 'benign', 'benign',
'malignant', 'benign', 'malignant', 'benign', 'benign', 'benign',
'benign', 'benign', 'benign', 'benign', 'benign', 'benign',
'benign', 'benign', 'benign', 'benign', 'benign', 'malignant',
'benign', 'benign', 'benign', 'malignant', 'benign', 'malignant',
'benign', 'benign', 'benign', 'benign', 'malignant', 'malignant',
'malignant', 'benign', 'benign', 'benign', 'benign', 'malignant',
'benign', 'malignant', 'benign', 'malignant', 'benign', 'benign',
'benign', 'malignant', 'benign', 'benign', 'benign', 'benign',
'benign', 'benign', 'benign', 'malignant', 'malignant',
'malignant', 'benign', 'benign', 'benign', 'benign', 'benign',
'benign', 'benign', 'benign', 'benign', 'benign', 'benign',
'malignant', 'malignant', 'benign', 'malignant', 'malignant',
'malignant', 'benign', 'malignant', 'malignant', 'benign',
'benign', 'benign', 'benign', 'benign', 'malignant', 'benign',
'benign', 'benign', 'benign', 'benign', 'malignant', 'benign',
'benign', 'benign', 'malignant', 'benign', 'benign', 'malignant',
'malignant', 'benign', 'benign', 'benign', 'benign', 'benign',
'benign', 'malignant', 'benign', 'benign', 'benign', 'benign',
'benign', 'benign', 'benign', 'malignant', 'benign', 'benign',
'benign', 'benign', 'benign', 'malignant', 'benign', 'benign',
'malignant', 'benign', 'benign', 'benign', 'benign', 'benign',
'benign', 'benign', 'benign', 'benign', 'benign', 'benign',
'benign', 'malignant', 'benign', 'malignant', 'malignant',
'benign', 'malignant', 'benign', 'benign', 'benign', 'benign',
'benign', 'malignant', 'benign', 'benign', 'malignant', 'benign',
'malignant', 'benign', 'benign', 'malignant', 'benign',
'malignant', 'benign', 'benign', 'benign', 'benign', 'benign',
'benign', 'benign', 'benign', 'malignant', 'malignant', 'benign',
'benign', 'benign', 'benign', 'benign', 'benign', 'malignant',
'benign', 'benign', 'benign', 'benign', 'benign', 'benign',
'benign', 'benign', 'benign', 'benign', 'malignant', 'benign',
'benign', 'benign', 'benign', 'benign', 'benign', 'benign',
'malignant', 'benign', 'malignant', 'benign', 'benign',
'malignant', 'benign', 'benign', 'benign', 'benign', 'benign',
'malignant', 'malignant', 'benign', 'malignant', 'benign',
'malignant', 'benign', 'benign', 'benign', 'benign', 'benign',
'malignant', 'benign', 'benign', 'malignant', 'benign',
'malignant', 'benign', 'malignant', 'malignant', 'benign',
```

```
'benign', 'benign', 'malignant', 'benign', 'benign', 'benign',
'benign', 'benign', 'benign', 'benign', 'benign', 'benign',
'benign', 'benign', 'malignant', 'benign', 'malignant',
'malignant', 'benign', 'benign', 'benign', 'benign', 'benign',
'benign', 'benign', 'benign', 'benign', 'benign', 'benign',
'benign', 'benign', 'benign', 'benign', 'benign', 'benign',
'benign', 'benign', 'benign', 'benign', 'benign', 'benign',
'benign', 'benign', 'malignant', 'malignant', 'malignant',
'malignant', 'malignant', 'malignant', 'benign'], dtype=object)
```

…and for shape of `y_text`:

[9]:
```
y_text.shape
```

[9]:
```
(569,)
```

1.3.1 Reducing dimensionality

[10]:
```
pca = PCA(n_components=2)
pca.fit(X_30)
X = pca.transform(X_30)
```

See how now we can find the proper transformation from the `X_30` and specify that we want our transformation to produce data with 2 features for us in the output by letting `n_components=2`? Also, see how PCA does not get the labels, in `fit`? It's unsupervised learning after all and it does not use the labels!

Let's check this new `X`:

[11]:
```
X
```

[11]:
```
array([[1160.1425737 , -293.91754364],
       [1269.12244319,   15.63018184],
       [ 995.79388896,   39.15674324],
       ...,
       [ 314.50175618,   47.55352518],
       [1124.85811531,   34.12922497],
       [-771.52762188,  -88.64310636]])
```

…and its shape:

[12]:
```
X.shape
```

[12]:
```
(569, 2)
```

Now we can generate a data frame from this two dimesional data `X` that we generated:

[13]:
```
df = pd.DataFrame(data=np.c_[X, y_text], columns=['Feature 1', 'Feature 2',␣
 ↪'Label'])
```

Let's take a look at our new 2-dimensional data as a table. We have to construct a data frame from our new 2-dimensional data as well as our labels:

[14]:
```
df
```

[14]:

	Feature 1	Feature 2	Label
0	1160.14	-293.918	malignant
1	1269.12	15.6302	malignant
2	995.794	39.1567	malignant
3	-407.181	-67.3803	malignant
4	930.341	189.341	malignant
5	-211.591	-79.8774	malignant
6	821.211	-47.1497	malignant
7	-25.09	-74.186	malignant
8	-191.293	-42.1265	malignant
9	-238.293	-65.3865	malignant
10	304.688	-17.7251	malignant
11	424.361	-109.22	malignant
12	634.514	167.205	malignant
13	63.0427	111.678	malignant
14	-196.441	29.6579	malignant
15	56.0047	-29.1483	malignant
16	235.858	-108.426	malignant
17	447.393	-102.152	malignant
18	1615.09	-270.333	malignant
19	-191.621	12.2592	benign
20	-285.051	14.5574	benign
21	-683.584	-32.5761	benign
22	112.56	-9.16055	malignant
23	1873.73	-260.196	malignant
24	1273.73	-479.321	malignant
25	634.88	-79.9317	malignant
26	8.59184	-16.9844	malignant
27	677.785	104.825	malignant
28	373.72	-135.335	malignant
29	453.678	77.4	malignant
..	...	...	...
539	-815.98	-74.3364	benign
540	-493.648	3.93097	benign
541	-60.7825	38.5607	benign
542	-39.6621	39.8148	benign
543	-276.258	30.4219	benign
544	-201.254	39.8182	benign
545	-171.841	8.35606	benign
546	-597.225	-25.3103	benign
547	-623.076	-14.5232	benign
548	-635.064	-48.2346	benign
549	-473.473	-56.5534	benign
550	-554.764	-9.01713	benign
551	-524.66	-6.2808	benign
552	-322.054	22.3741	benign
553	-704.967	-31.3012	benign

```
554  -317.926   27.3694     benign
555  -622.215  -14.2743     benign
556  -636.046  -17.0436     benign
557  -670.677   -43.207     benign
558  -125.245   78.4234     benign
559  -479.336  -4.37787     benign
560  -177.243   43.7231     benign
561  -518.012  -1.53085     benign
562   61.9408   35.2648  malignant
563   1167.14   105.597  malignant
564   1414.13   110.222  malignant
565   1045.02   77.0576  malignant
566   314.502   47.5535  malignant
567   1124.86   34.1292  malignant
568  -771.528  -88.6431     benign

[569 rows x 3 columns]
```

Let's also do a scatter plot of our data:

[15]:
```
fig = px.scatter(df, x='Feature 1', y='Feature 2', color='Label')
fig.show()
```

We can also create $\{-1, +1\}$ labels for our data from `y_text` and assign it to (vector) variable `y`. We use `LabelEncoder` from scikit-learn again to transform labels into -1s or +1s:

[16]:
```
y = (2 * LabelEncoder().fit_transform(y_text)) - 1
```

As usual let's check our `y`:

[17]:
```
y
```

[17]:
```
array([ 1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,
        1,  1, -1, -1, -1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,
        1,  1,  1, -1,  1,  1,  1,  1,  1,  1,  1,  1, -1,  1, -1, -1, -1,
       -1, -1,  1,  1, -1,  1,  1, -1, -1, -1, -1,  1, -1,  1,  1, -1, -1,
       -1, -1,  1, -1,  1,  1, -1,  1, -1,  1,  1, -1, -1, -1,  1,  1, -1,
        1,  1,  1, -1, -1, -1,  1, -1, -1,  1,  1, -1, -1, -1,  1,  1, -1,
       -1, -1, -1,  1, -1, -1,  1, -1, -1, -1, -1, -1, -1, -1, -1,  1,  1,
        1, -1,  1,  1, -1, -1, -1,  1,  1, -1,  1, -1,  1,  1, -1,  1,  1,
       -1, -1,  1, -1, -1,  1, -1, -1, -1, -1,  1, -1, -1, -1, -1, -1, -1,
       -1, -1, -1,  1, -1, -1, -1, -1,  1,  1, -1,  1, -1, -1,  1,  1, -1,
       -1,  1,  1, -1, -1, -1, -1,  1, -1, -1,  1,  1,  1, -1,  1, -1,  1,
       -1, -1, -1,  1, -1, -1,  1,  1, -1,  1,  1,  1,  1, -1,  1,  1,  1,
       -1,  1, -1,  1, -1, -1,  1, -1,  1,  1,  1,  1, -1, -1,  1,  1, -1,
       -1, -1,  1, -1, -1, -1, -1, -1,  1,  1, -1, -1,  1, -1, -1,  1,  1,
       -1,  1, -1, -1, -1, -1,  1, -1, -1, -1, -1, -1,  1, -1,  1,  1,  1,
        1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1, -1, -1, -1, -1, -1, -1,
        1, -1,  1, -1, -1,  1, -1, -1,  1, -1,  1,  1, -1, -1, -1, -1, -1,
       -1, -1, -1, -1, -1, -1, -1, -1,  1, -1, -1,  1, -1,  1, -1, -1, -1,
       -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,  1, -1, -1, -1,  1, -1,
```

```
 1, -1, -1, -1, -1,  1,  1,  1, -1, -1, -1, -1,  1, -1,  1, -1,  1,
-1, -1, -1,  1, -1, -1, -1, -1, -1, -1, -1,  1,  1,  1, -1, -1, -1,
-1, -1, -1, -1, -1, -1, -1, -1,  1,  1, -1,  1,  1,  1, -1,  1,  1,
-1, -1, -1, -1, -1,  1, -1, -1, -1, -1, -1,  1, -1, -1, -1,  1, -1,
-1,  1,  1, -1, -1, -1, -1, -1, -1,  1, -1, -1, -1, -1, -1, -1, -1,
 1, -1, -1, -1, -1, -1,  1, -1, -1,  1, -1, -1, -1, -1, -1, -1, -1,
-1, -1, -1, -1, -1,  1, -1,  1,  1, -1,  1, -1, -1, -1, -1, -1,  1,
-1, -1,  1, -1,  1, -1, -1,  1, -1,  1, -1, -1, -1, -1, -1, -1, -1,
-1,  1,  1, -1, -1, -1, -1, -1, -1,  1, -1, -1, -1, -1, -1, -1, -1,
-1, -1, -1,  1, -1, -1, -1, -1, -1, -1, -1,  1, -1,  1, -1, -1,  1,
-1, -1, -1, -1, -1,  1,  1, -1,  1, -1,  1, -1, -1, -1, -1, -1,  1,
-1, -1,  1, -1,  1, -1,  1,  1, -1, -1, -1,  1, -1, -1, -1, -1, -1,
-1, -1, -1, -1, -1, -1,  1, -1,  1,  1, -1, -1, -1, -1, -1, -1, -1,
-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
-1,  1,  1,  1,  1,  1,  1, -1])
```

…and its shape:

[18]:
```
y.shape
```

[18]:
```
(569,)
```

Now, we can plot our training data in 3D with a 3D scatter plot (we are going to use surface plots afterwards and the new interface of plotly cannot do surface plots yet, so we are using the older style rather than plotly express):

[19]:
```
points_colorscale = [
                     [0.0, 'rgb(239, 85, 59)'],
                     [1.0, 'rgb(99, 110, 250)'],
                    ]

layout = go.Layout(scene=dict(
                              xaxis=dict(title='Feature 1'),
                              yaxis=dict(title='Featrue 2'),
                              zaxis=dict(title='Label')
                             ),
                  )

points = go.Scatter3d(x=df['Feature 1'],
                      y=df['Feature 2'],
                      z=y,
                      mode='markers',
                      text=df['Label'],
                      marker=dict(
                                  size=3,
                                  color=y,
                                  colorscale=points_colorscale
                             ),
                     )
```

```
fig2 = go.Figure(data=[points], layout=layout)
fig2.show()
```

1.4 Splitting data

Now, let's split our data into training, validation and test sets. We don't need validation data in this example and we won't be doing model selection here. So, let's use 70% and 30% for training test data, repectively.

[20]:
```
(X_train, X_test, y_train, y_test) = train_test_split(X, y, test_size=0.3,␣
 ↪random_state=0)
```

1.5 Building and visualizing a logistic regression model

Let's build our logistic regression model then by creating an object of the `LogisticRegression` class and assign the name `logreg` to the resulting object.

You can see the documentation for `LogisticRegression` here: https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html

Go ahead and do that now:

[22]:
```
### begin your code here (1 line).
logreg = LogisticRegression()
### end your code here.
```

Now, fit `logreg` to `X_train` and `y_train`:

[23]:
```
### begin your code here (1 line).
logreg.fit(X_train, y_train)
### end your code here.
```

[23]:
```
LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
          intercept_scaling=1, max_iter=100, multi_class='warn',
          n_jobs=None, penalty='l2', random_state=None, solver='warn',
          tol=0.0001, verbose=0, warm_start=False)
```

You will get a summary for the model:

> LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True, intercept_scaling=1, max_iter=100, multi_class='warn', n_jobs=None, penalty='l2', random_state=None, solver='warn', tol=0.0001, verbose=0, warm_start=False)

- You may also get a warning because you have not explicitly set a solver and that is going to change in newer versions of scikit-learn. Nothing you should be worried about here.

Let's visualize the surface generated by our logistic regression model. First, we need to generate a number of points required for creating a visualization of the decision surface:

[24]:
```
detail_steps = 100

(x_vis_0_min, x_vis_1_min) = X_train.min(axis=0)
(x_vis_0_max, x_vis_1_max) = X_train.max(axis=0)
```

```
x_vis_0_range = np.linspace(x_vis_0_min, x_vis_0_max, detail_steps)
x_vis_1_range = np.linspace(x_vis_1_min, x_vis_1_max, detail_steps)

(XX_vis_0, XX_vis_1) = np.meshgrid(x_vis_0_range, x_vis_0_range)

X_vis = np.c_[XX_vis_0.reshape(-1), XX_vis_1.reshape(-1)]
```

We need to predict the proability associated with points in this generated data in order to visualize it. You can get the probabilities associated with belonging to classes by `predict_proba` method. Let's use that to calculate probabilities for points in `X_vis`. Use `predict_proba` just like `predict` to predict probabilities instead of actual classes. Go ahead and do that now, and assign the result to variable `probs`:

[25]:
```
### begin your code here (1 line).
probs = logreg.predict_proba(X_vis)
### end your code here.
```

Let's check the shape of this variable `probs`:

[26]:
```
probs.shape
```

[26]:
```
(10000, 2)
```

As you can see, it has two column, because it gives the probability of belonging to each of the two classes. However, we care only about the probability of belonging to the positive class, so we can only choose the cloumn with index `1`. Also, the probabilities will be in $[0, 1]$ while our labels are $\{+1, 1\}$, so we will transform the probabilities to be in range $[-1, +1]$:

[27]:
```
yhat_vis = (2 * probs[:, 1]) - 1
```

Now, we can transfrom `yhat_vis` into the shape required for a surface plot and plot away:

[28]:
```
YYhat_vis = yhat_vis.reshape(XX_vis_0.shape)

surface_colorscale = [
                      [0.0, 'rgb(235, 185, 177)'],
                      [1.0, 'rgb(199, 204, 249)'],
                     ]

surface = go.Surface(
                     x=XX_vis_0,
                     y=XX_vis_1,
                     z=YYhat_vis,
                     colorscale=surface_colorscale,
                     showscale=False
                    )

fig3 = go.Figure(data=[points, surface], layout=layout)
fig3.show()
```

We can see that logistic regression has fit a surface to our data that is has the logistic (or Sigmoid) function as its intersection.

1.6 Assessing the performance

Let's check our accuracies next. First, the training accuracy. For that let's get the predictions of training data. Predict yhat_train by logreg on X_train:

[29]:
```
### begin your code here (1 line).
yhat_train = logreg.predict(X_train)
### end your code here.
```

Let's measure the accuracy:

[30]:
```
accuracy_score(yhat_train, y_train)
```

[30]:
```
0.9195979899497487
```

We got 91.95%. Let's check accuracy on the test data. Predict yhat_test:

[31]:
```
### begin your code here (1 line).
yhat_test = logreg.predict(X_test)
### end your code here.
accuracy_score(yhat_test, y_test)
```

[31]:
```
0.9532163742690059
```

95.32%. We have better performance on test data than on training data! But that's just random and it does not mean that we have perfectly generalized and have no overfitting: that is theoretically impossible!

That's it for now.