

# Johdanto

Tässä projektissa opimme, miten logistisen regressiomallin koulutus tapahtuu. Tämä on tarkoitettu johdatukseksi logistiseen regressioon, mutta emme käsittele mallin matemaattista taustaa.

Käytämme rintasyöpääineistoa, joka sisältää erittäin yksityiskohtaisia solumittauksia. Jokaisen havaintomittauksen mukana on diagnoosi siitä, onko solu pahanlaatuinen vai ei.

Tavoitteemme on kouluttaa malli, joka pystyy ennustamaan, onko annettu solu pahanlaatuinen pelkkien mittausten perusteella.

Lähde: <https://www.youtube.com/watch?v=My4JgleFdWk>

```
In [1]: import numpy as np # Linear algebra
import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv)
import seaborn as sns
```

## Tuodaan data

Aloitetaan tutustumalla aineistoon ja saadaan käsiteksi siitä, mitä se edustaa. Voimme esimerkiksi nähdä, että kaikki muuttujat – paitsi tunnus ja diagnoosi – ovat jo tyyppiä float64, mikä tarkoittaa, että ne ovat numeerisia. Tämä on hyvä lähtökohta logistisen regressiomallin käyttämiselle, koska logistinen regressiomalli ottaa syötteeksi numeerisia arvoja ja tuottaa kaksiluokkaisen luokituksen (kyllä/ei-arvon).

```
In [2]: data = pd.read_csv("data.csv")
data.head()
```

```
Out[2]:
```

	<b>id</b>	<b>diagnosis</b>	<b>radius_mean</b>	<b>texture_mean</b>	<b>perimeter_mean</b>	<b>area_mean</b>	<b>smoothn</b>
<b>0</b>	842302	M	17.99	10.38	122.80	1001.0	
<b>1</b>	842517	M	20.57	17.77	132.90	1326.0	
<b>2</b>	84300903	M	19.69	21.25	130.00	1203.0	
<b>3</b>	84348301	M	11.42	20.38	77.58	386.1	
<b>4</b>	84358402	M	20.29	14.34	135.10	1297.0	

5 rows × 33 columns



```
In [3]: data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 569 entries, 0 to 568
Data columns (total 33 columns):
 #   Column           Non-Null Count  Dtype  
--- 
 0   id               569 non-null    int64  
 1   diagnosis        569 non-null    object  
 2   radius_mean      569 non-null    float64 
 3   texture_mean     569 non-null    float64 
 4   perimeter_mean   569 non-null    float64 
 5   area_mean        569 non-null    float64 
 6   smoothness_mean  569 non-null    float64 
 7   compactness_mean 569 non-null    float64 
 8   concavity_mean   569 non-null    float64 
 9   concave_points_mean 569 non-null    float64 
 10  symmetry_mean   569 non-null    float64 
 11  fractal_dimension_mean 569 non-null    float64 
 12  radius_se        569 non-null    float64 
 13  texture_se       569 non-null    float64 
 14  perimeter_se    569 non-null    float64 
 15  area_se          569 non-null    float64 
 16  smoothness_se   569 non-null    float64 
 17  compactness_se  569 non-null    float64 
 18  concavity_se   569 non-null    float64 
 19  concave_points_se 569 non-null    float64 
 20  symmetry_se    569 non-null    float64 
 21  fractal_dimension_se 569 non-null    float64 
 22  radius_worst    569 non-null    float64 
 23  texture_worst   569 non-null    float64 
 24  perimeter_worst 569 non-null    float64 
 25  area_worst       569 non-null    float64 
 26  smoothness_worst 569 non-null    float64 
 27  compactness_worst 569 non-null    float64 
 28  concavity_worst 569 non-null    float64 
 29  concave_points_worst 569 non-null    float64 
 30  symmetry_worst  569 non-null    float64 
 31  fractal_dimension_worst 569 non-null    float64 
 32  Unnamed: 32      0 non-null    float64 
dtypes: float64(31), int64(1), object(1)
memory usage: 146.8+ KB
```

In [4]: `data.describe()`

Out[4]:

	<b>id</b>	<b>radius_mean</b>	<b>texture_mean</b>	<b>perimeter_mean</b>	<b>area_mean</b>	<b>smoothness</b>
<b>count</b>	5.690000e+02	569.000000	569.000000	569.000000	569.000000	569.000000
<b>mean</b>	3.037183e+07	14.127292	19.289649	91.969033	654.889104	0.132084
<b>std</b>	1.250206e+08	3.524049	4.301036	24.298981	351.914129	0.305381
<b>min</b>	8.670000e+03	6.981000	9.710000	43.790000	143.500000	0.059902
<b>25%</b>	8.692180e+05	11.700000	16.170000	75.170000	420.300000	0.105946
<b>50%</b>	9.060240e+05	13.370000	18.840000	86.240000	551.100000	0.129981
<b>75%</b>	8.813129e+06	15.780000	21.800000	104.100000	782.700000	0.153016
<b>max</b>	9.113205e+08	28.110000	39.280000	188.500000	2501.000000	0.176051

8 rows × 32 columns



## Puhdista data

Käyttämällä heatmapia voimme helposti visualisoida aineistossa olevat puuttuvat arvot (NA) ja käsitellä ne asianmukaisesti. Tässä esimerkissä aineistossa on kokonainen sarake puuttuvia arvoja. Pudotamme sen pois yhdessä tunnussarakkeen kanssa (joka on tarpeeton tarkoitukiimme) ja jatkamme analyysiä.

Muunna myös kohdemuuttujamme 1:ksi ja 0:ksi mallin kouluttamista varten.

Muuten aineisto näyttää olevan melko puhdas, joten emme tarvitse enempää puhdistusta.

In [5]:

```
# visualize NAs in heatmap
sns.heatmap(data.isnull(), yticklabels=False, cbar=False, cmap='viridis')
```

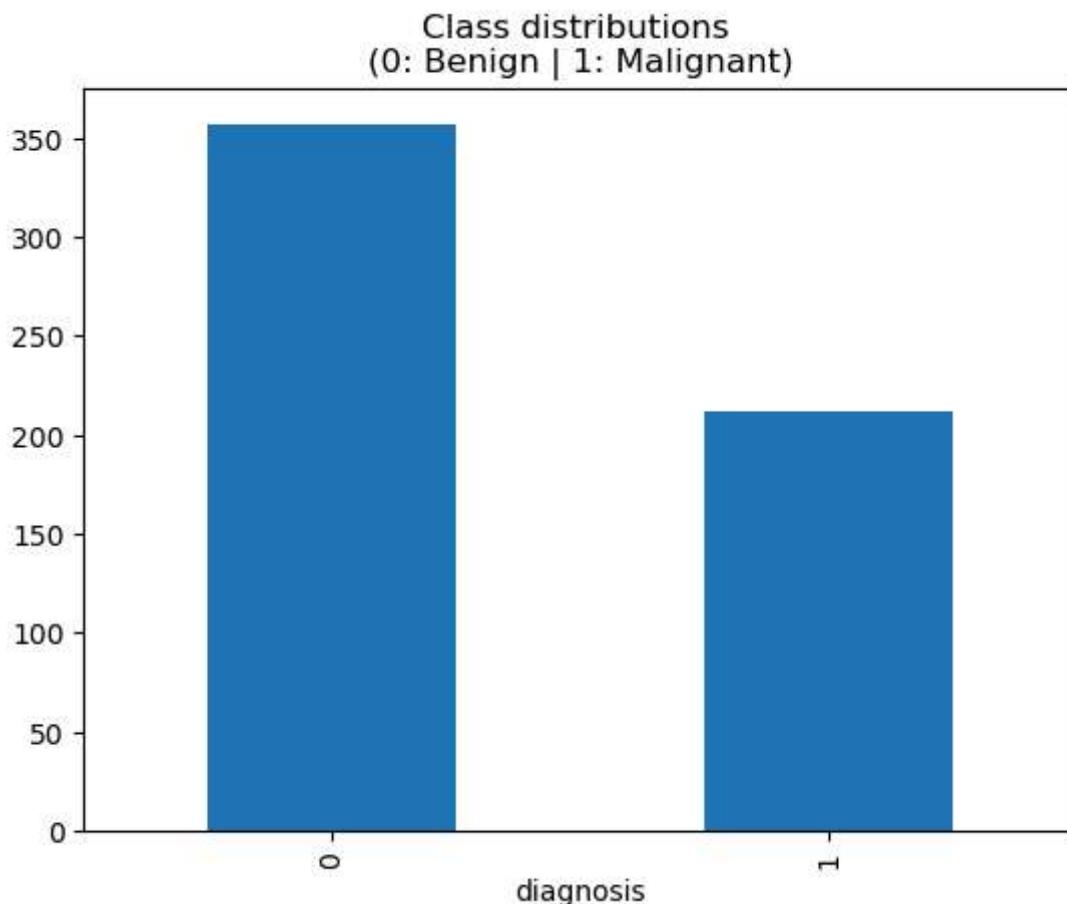
Out[5]: &lt;Axes: &gt;

```
id - diagnosis  
radius_mean -  
texture_mean -  
perimeter_mean -  
area_mean -  
smoothness_mean -  
compactness_mean -  
concavity_mean -  
concave points_mean -  
symmetry_mean -  
fractal_dimension_mean -  
radius_se -  
texture_se -  
perimeter_se -  
area_se -  
smoothness_se -  
compactness_se -  
concavity_se -  
concave points_se -  
symmetry_se -  
fractal_dimension_se -  
radius_worst -  
texture_worst -  
perimeter_worst -  
area_worst -  
smoothness_worst -  
compactness_worst -  
concavity_worst -  
concave points_worst -  
symmetry_worst -  
fractal_dimension_worst -  
Unnamed_32 -
```

```
In [6]: # drop id and empty column  
data.drop(['Unnamed: 32', "id"], axis=1, inplace=True)
```

```
In [7]: # turn target variable into 1s and 0s  
data.diagnosis =[1 if value == "M" else 0 for value in data.diagnosis]
```

```
In [8]: # turn the target variable into categorical data  
data['diagnosis'] = data['diagnosis'].astype('category', copy=False)  
plot = data['diagnosis'].value_counts().plot(kind='bar', title="Class distributions")  
fig = plot.get_figure()
```



# Logistic Regression

## Esikäsittely

Kun aineistomme on puhdas ja tiedämme, että muuttujamme ovat luotettavia, voimme siirtyä mallin kouluttamiseen. Ensimmäinen tehtävä on erottaa kohdemuuttuja (tässä kutsuttu "y":ksi) ja selittävät muuttujat (tässä kutsuttu "X":ksi). Huomaa, että käytämme isoa X-kirjainta noudattaaksemme matemaattista konventiota. Matematiikassa iso kirjain edustaa monitahoista muuttujaa (matriisia).

```
In [9]: # Prepare the model
y = data["diagnosis"] # our target variable
X = data.drop(["diagnosis"], axis=1) # our predictors
```

## Normalisoi data

Houkutuksesta huolimatta älä käytä dataa suoraan koulutus- ja testijoukkoihin jakamiseen., sillä data ei ole vielä normalisoitu. Tämä voi olla ongelma, koska muuttujien yksiköt eivät välttämättä ole samoissa mittayksiköissä. Lisäksi voi olla poikkeavia arvoja, jotka voivat heikentää mallin suorituskykyä.

Näissä tapauksissa normalisoimme datan ennen sen syöttämistä malliin. Tämä parantaa koneoppimisalgoritmin suorituskykyä.

```
In [10]: from sklearn.preprocessing import StandardScaler

# Create a scaler object
scaler = StandardScaler()

# Fit the scaler to the data and transform the data
X_scaled = scaler.fit_transform(X)

# X_scaled is now a numpy array with normalized data
```

Seuraavaksi jaamme aineiston koulutus- ja testijoukkoihin. Molemmilla on samat muuttujat (sarakkeet), mutta eri havainnot (rivit). Tätä varten käytämme kätevää Scikit-Learn-kirjaston funktiota nimeltä *train\_test\_split*. Tämä funktio ottaa selittävät muuttujat ja kohdemuuttujan ja jakaa ne satunnaisesti koulutus- ja testijoukkoihin. Funktio palauttaa neljä arvoa:

- Koulutusjoukon selittävät muuttujat, jotka tallennamme Python-muuttujaan nimeltä *X\_train*.
- Koulutusjoukon kohdemuuttujat, jotka tallennamme Python-muuttujaan nimeltä *y\_train*.
- Testijoukon selittävät muuttujat, jotka tallennamme Python-muuttujaan nimeltä *X\_test*.
- Testijoukon kohdemuuttujat, jotka tallennamme Python-muuttujaan nimeltä *y\_test*.

Jokainen havainto (rivi) *X*:ssä vastaa kohdearvoa *y*:ssä.

Lisäksi *train\_test\_split*-funktio ottaa käyttöön parametrin *test\_size*, joka määrittää testijoukon koon, ja parametrin *random\_state*, joka on mielivaltainen kokonaisluku, jonka avulla voimme toistaa jaon myöhemmin samalla tavalla. Usein valitsemme arvoksi 42, koska se on vastaus kaikkeen.

```
In [11]: from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X_scaled, y, test_size=0.30, ra
```

```
In [12]: from sklearn.linear_model import LogisticRegression

# Create Logistic regression model
lr = LogisticRegression()

# Train the model on the training data
lr.fit(X_train, y_train)

# Predict the target variable on the test data
y_pred = lr.predict(X_test)
```

## Arvioi malli

Tässä notebookissa olemme kouluttaneet logistisen regressiomallin ennustamaan kohdemuuttuja käyttämällä aineiston syötemuuttujia. Kuten tässä näemme, koulutettuamme mallin koulutusdatalla ja arvioituaamme sen suorituskyvyn testidatalla, saavutimme lopulliseksi tarkkuudeksi 0,98. Tämä on vahva suoritus ja osoittaa, että malli pystyy tekemään tarkkoja ennusteita uudesta, aiemmin näkemättömästä datasta.

On kuitenkin tärkeää huomata, että tarkkuus on vain yksi mittari mallin suorituskyvystä, eikä se välttämättä ole kaikissa tapauksissa paras mittari. Riippuen ongelmasta ja sovelluksen erityisvaatimuksista, muut mittarit, kuten precision, recall tai F1-score, voivat olla merkityksellisempia. Toisessa solussa käytämme Scikit-learnin *classification\_report*-funktiota näiden mittarien laskemiseen.

```
In [13]: from sklearn.metrics import accuracy_score

# Evaluate the accuracy of the model
accuracy = accuracy_score(y_test, y_pred)
print(f'Accuracy: {accuracy:.2f}')
```

Accuracy: 0.98

```
In [14]: from sklearn.metrics import classification_report
print(classification_report(y_test,y_pred))
```

	precision	recall	f1-score	support
0	0.99	0.98	0.99	108
1	0.97	0.98	0.98	63
accuracy			0.98	171
macro avg	0.98	0.98	0.98	171
weighted avg	0.98	0.98	0.98	171

## Johtopäätökset

Olemme nyt saaneet analyysimme valmiiksi. Käytimme avoimen Breast Cancer -aineiston dataa rakentaaksemme mallin, joka ennustaa, onko solu pahanlaatuinen tiettyjen tumamittausten perusteella. Tämä malli, kun se on koulutettu, voi olla erittäin hyödyllinen solujen analysointiin sairaalassa.

Koska malli voidaan helposti kutsua Python-funktiona ennustuksia varten, se voidaan myös helposti liittää palvelinteknologiaan, kuten Flaskiin, ja tarjota käyttöliittymän kautta, jota lääkärit voivat käyttää. Esimerkiksi voisimme rakentaa käyttöliittymän, jossa lääkäri syöttää suorittamiaan mittauksia, ja malli antaisi vastauksena tiedon siitä, onko solu pahanlaatuinen vai ei. Tai realistiksempi käyttötapa voisi olla yhdistää taustajärjestelmä laitteeseen, joka ottaa kudosnäytteen, mittaa solut ja tekee diagnoosin.

Pytonia käyttämällä tällaisen sovelluksen API olisi erittäin yksinkertainen, ja kaikki, mitä teimme ihmishenkien pelastamiseksi, oli kouluttaa logistisen regressiomalli sairaalan datan perusteella.

In [ ]: