

A Report
By
Shruti Agrawal
On
Predicting Heart Disease Using
Machine Learning

Putting machine learning to work

Predicting Heart Disease Using Machine Learning

This notebook will introduce some foundation machine learning and data science concepts by exploring the problem of heart disease **classification**.

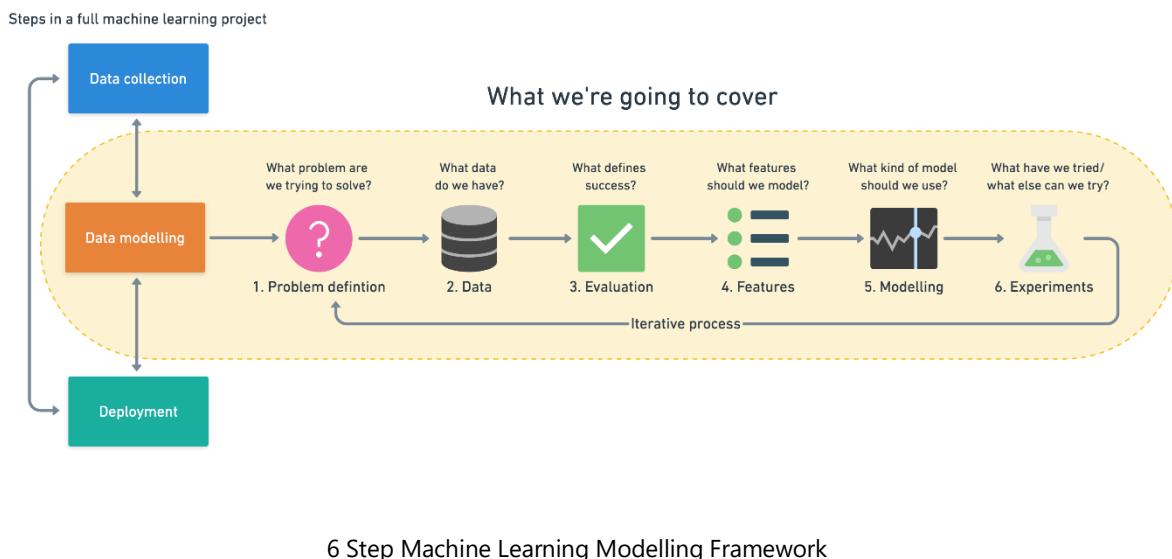
It is intended to be an end-to-end example of what a data science and machine learning **proof of concept** might look like.

What is classification?

Classification involves deciding whether a sample is part of one class or another (**single-class classification**). If there are multiple class options, it's referred to as **multi-class classification**.

What we'll end up with

Since we already have a dataset, we'll approach the problem with the following machine learning modelling framework.

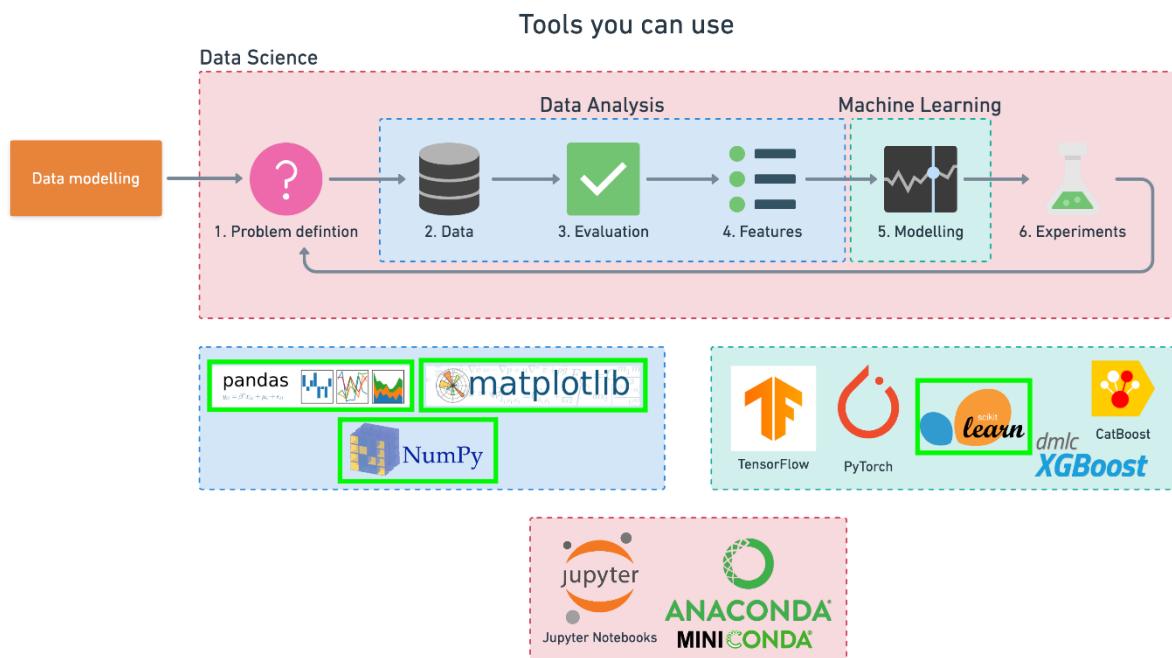


More specifically, we'll look at the following topics.

- **Exploratory data analysis (EDA)** - the process of going through a dataset and finding out more about it.
- **Model training** - create model(s) to learn to predict a target variable based on other variables.
- **Model evaluation** - evaluating a models predictions using problem-specific evaluation metrics.

- **Model comparison** - comparing several different models to find the best one.
- **Model fine-tuning** - once we've found a good model, how can we improve it?
- **Feature importance** - since we're predicting the presence of heart disease, are there some things which are more important for prediction?
- **Cross-validation** - if we do build a good model, can we be sure it will work on unseen data?
- **Reporting what we've found** - if we had to present our work, what would we show someone?

To work through these topics, we'll use pandas, Matplotlib and NumPy for data analysis, as well as, Scikit-Learn for machine learning and modelling tasks.



Tools which can be used for each step of the machine learning modelling process.

We'll work through each step and by the end of the notebook, we'll have a handful of models, all which can predict whether or not a person has heart disease based on a number of different parameters at a considerable accuracy.

You'll also be able to describe which parameters are more indicative than others, for example, sex may be more important than age.

Predicting heart disease using machine learning

This notebook looks into using various Python-based machine learning and data science libraries in an attempt to build a machine learning model capable of predicting whether or not someone has heart disease based on their medical attributes.

We're going to take the following approach:

1. Problem definition
2. Data
3. Evaluation
4. Features
5. Modelling
6. Experimentation

1. Problem Definition

In our case, the problem we will be exploring is **binary classification** (a sample can only be one of two things).

This is because we're going to be using a number of different **features** (pieces of information) about a person to predict whether they have heart disease or not.

In a statement,

Given clinical parameters about a patient, can we predict whether or not they have heart disease?

2. Data

What you'll want to do here is dive into the data your problem definition is based on. This may involve, sourcing, defining different parameters, talking to experts about it and finding out what you should expect.

The original data came from the [Cleveland database](#) from UCI Machine Learning Repository.

However, we've downloaded it in a formatted way from [Kaggle](#).

The original database contains 76 attributes, but here only 14 attributes will be used. **Attributes** (also called **features**) are the variables what we'll use to predict our **target variable**.

Attributes and features are also referred to as **independent variables** and a target variable can be referred to as a **dependent variable**.

We use the independent variables to predict our dependent variable.

Or in our case, the independent variables are a patients different medical attributes and the dependent variable is whether or not they have heart disease.

3. Evaluation

The evaluation metric is something you might define at the start of a project.

Since machine learning is very experimental, you might say something like,

If we can reach 95% accuracy at predicting whether or not a patient has heart disease during the proof of concept, we'll pursue this project.

The reason this is helpful is it provides a rough goal for a machine learning engineer or data scientist to work towards.

However, due to the nature of experimentation, the evaluation metric may change over time.

4. Features

Features are different parts of the data. During this step, you'll want to start finding out what you can about the data.

One of the most common ways to do this, is to create a **data dictionary**.

Heart Disease Data Dictionary

A data dictionary describes the data you're dealing with. Not all datasets come with them so this is where you may have to do your research or ask a **subject matter expert** (someone who knows about the data) for more.

The following are the features we'll use to predict our target variable (heart disease or no heart disease).

Create data dictionary

1. age - age in years
2. sex - (1 = male; 0 = female)
3. cp - chest pain type
 - 0: Typical angina: chest pain related decrease blood supply to the heart
 - 1: Atypical angina: chest pain not related to heart
 - 2: Non-anginal pain: typically oesophageal spasms (non heart related)
 - 3: Asymptomatic: chest pain not showing signs of disease
4. trestbps - resting blood pressure (in mm Hg on admission to the hospital) anything above 130-140 is typically cause for concern
5. Chol - serum cholesterol in mg/dl
 - serum = LDL + HDL + .2 * triglycerides
 - above 200 is cause for concern
6. fbs - (fasting blood sugar > 120 mg/dl) (1 = true; 0 = false)
 - '>126' mg/dL signals diabetes
7. restecg - resting electrocardiographic results
 - 0: Nothing to note
 - 1: ST-T Wave abnormality
 1. can range from mild symptoms to severe problems
 2. signals non-normal heart beat
 - 2: Possible or definite left ventricular hypertrophy
 1. Enlarged heart's main pumping chamber
8. thalach - maximum heart rate achieved
9. exang - exercise induced angina (1 = yes; 0 = no)
10. oldpeak - ST depression induced by exercise relative to rest looks at stress of heart during exercise unhealthy heart will stress more
11. slope - the slope of the peak exercise ST segment
 - 0: Upsloping: better heart rate with exercise (uncommon)

- 1: Flatsloping: minimal change (typical healthy heart)
 - 2: Downslopins: signs of unhealthy heart
12. ca - number of major vessels (0-3) colored by flourosopy
- colored vessel means the doctor can see the blood passing through
 - the more blood movement the better (no clots)
13. thal - thalium stress result
- 1,3: normal
 - 6: fixed defect: used to be defect but ok now
 - 7: reversable defect: no proper blood movement when excercising
14. target - have disease or not (1=yes, 0=no) (= the predicted attribute)

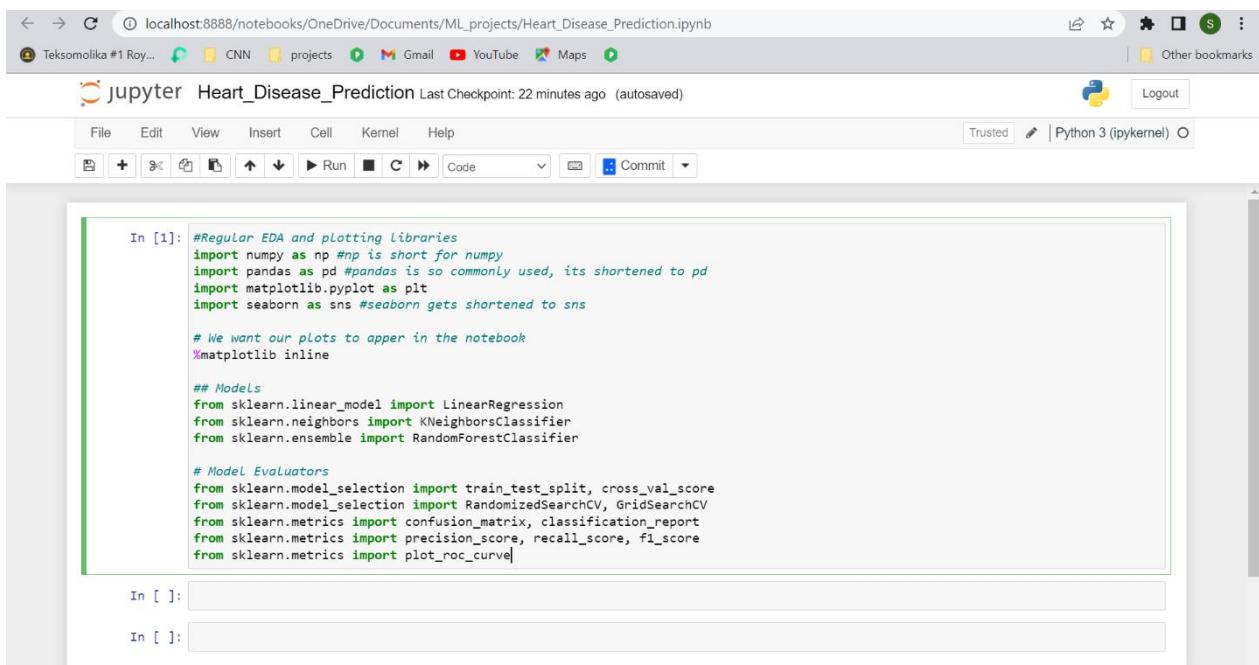
Preparing the tools

At the start of any project, it's custom to see the required libraries imported in a big chunk like you can see below.

However, in practice, your projects may import libraries as you go. After you've spent a couple of hours working on your problem, you'll probably want to do some tidying up. This is where you may want to consolidate every library you've used at the top of your notebook (like the cell below).

The libraries you use will differ from project to project. But there are a few which will you'll likely take advantage of during almost every structured data project.

- [pandas](#) for data analysis.
- [NumPy](#) for numerical operations.
- [Matplotlib/seaborn](#) for plotting or data visualization.
- [Scikit-Learn](#) for machine learning modelling and evaluation.



The screenshot shows a Jupyter Notebook interface with the following details:

- Header:** The title bar shows "localhost:8888/notebooks/OneDrive/Documents/ML_projects/Heart_Disease_Prediction.ipynb".
- Toolbar:** Includes standard Jupyter Notebook buttons for File, Edit, View, Insert, Cell, Kernel, Help, and a Python 3 (ipykernel) kernel selector.
- Code Cell:** The main content area contains a code cell labeled "In [1]:" with the following Python code:

```
#Regular EDA and plotting Libraries
import numpy as np #np is short for numpy
import pandas as pd #pandas is so commonly used, its shortened to pd
import matplotlib.pyplot as plt
import seaborn as sns #seaborn gets shortened to sns

# We want our plots to appear in the notebook
%matplotlib inline

## Models
from sklearn.linear_model import LinearRegression
from sklearn.neighbors import KNeighborsClassifier
from sklearn.ensemble import RandomForestClassifier

## Model Evaluators
from sklearn.model_selection import train_test_split, cross_val_score
from sklearn.model_selection import RandomizedSearchCV, GridsearchCV
from sklearn.metrics import confusion_matrix, classification_report
from sklearn.metrics import precision_score, recall_score, f1_score
from sklearn.metrics import plot_roc_curve
```
- Output Cells:** Below the code cell, there are two empty output cells labeled "In []:".

Load Data

There are many different kinds of ways to store data. The typical way of storing **tabular data**, data similar to what you'd see in an Excel file is in .csv format. .csv stands for comma separated values.

Pandas has a built-in function to read .csv files called `read_csv()` which takes the file pathname of your .csv file. You'll likely use this a lot.



```
from sklearn.metrics import confusion_matrix, classification_report
from sklearn.metrics import precision_score, recall_score, f1_score
from sklearn.metrics import plot_roc_curve

In [2]: df = pd.read_csv("datasets/heart-disease.csv") # DataFrame shortened to df
df.shape # (rows, columns)

Out[2]: (303, 14)

In [ ]:
```

Data Exploration (exploratory data analysis or EDA)

Once you've imported a dataset, the next step is to explore. There's no set way of doing this. But what you should be trying to do is become more and more familiar with the dataset.

Compare different columns to each other, compare them to the target variable. Refer back to your **data dictionary** and remind yourself of what different columns mean.

Your goal is to become a subject matter expert on the dataset you're working with. So if someone asks you a question about it, you can give them an explanation and when you start building models, you can sound check them to make sure they're not performing too well (**overfitting**) or why they might be performing poorly (**underfitting**).

Since EDA has no real set methodology, the following is a short check list you might want to walk through:

1. What question(s) are you trying to solve (or prove wrong)?
2. What kind of data do you have and how do you treat different types?
3. What's missing from the data and how do you deal with it?
4. Where are the outliers and why should you care about them?
5. How can you add, change or remove features to get more out of your data?

One of the quickest and easiest ways to check your data is with the head() function. Calling it on any DataFrame will print the top 5 rows, tail() calls the bottom 5. You can also pass a number to them like head(10) to show the top 10 rows.

```
In [3]: # Let's check the top 5 data frames of our table  
df.head()
```

```
Out[3]:
```

	age	sex	cp	trestbps	chol	fb	restecg	thalach	exang	oldpeak	slope	ca	thal	target
0	63	1	3	145	233	1	0	150	0	2.3	0	0	1	1
1	37	1	2	130	250	0	1	187	0	3.5	0	0	2	1
2	41	0	1	130	204	0	0	172	0	1.4	2	0	2	1
3	56	1	1	120	236	0	1	178	0	0.8	2	0	2	1
4	57	0	0	120	354	0	1	163	1	0.6	2	0	2	1

```
In [4]: # And top 10  
df.head(10)
```

```
Out[4]:
```

	age	sex	cp	trestbps	chol	fb	restecg	thalach	exang	oldpeak	slope	ca	thal	target
0	63	1	3	145	233	1	0	150	0	2.3	0	0	1	1
1	37	1	2	130	250	0	1	187	0	3.5	0	0	2	1
2	41	0	1	130	204	0	0	172	0	1.4	2	0	2	1
3	56	1	1	120	236	0	1	178	0	0.8	2	0	2	1
4	57	0	0	120	354	0	1	163	1	0.6	2	0	2	1
5	57	1	0	140	192	0	1	148	0	0.4	1	0	1	1
6	56	0	1	140	294	0	0	153	0	1.3	1	0	2	1
7	44	1	1	120	263	0	1	173	0	0.0	2	0	3	1
8	52	1	2	172	199	1	1	162	0	0.5	2	0	3	1
9	57	1	2	150	168	0	1	174	0	1.6	2	0	2	1

```
In [ ]:
```

value_counts() allows you to show how many times each of the values of a **categorical** column appear.

```
In [5]: # Let's see how many positive (1) and negative (0) samples we have in our dataframe  
df.target.value_counts()
```

```
Out[5]: 1    165  
0    138  
Name: target, dtype: int64
```

Since these two values are close to even, our target column can be considered **balanced**.

An **unbalanced** target column, meaning some classes have far more samples, can be harder to model than a balanced set. Ideally, all of your target classes have the same number of samples.

If you'd prefer these values in percentages, value_counts() takes a parameter, normalize which can be set to true.

```
In [6]: # Normalized value counts  
df.target.value_counts(normalize=True)
```

```
Out[6]: 1    0.544554  
0    0.455446  
Name: target, dtype: float64
```

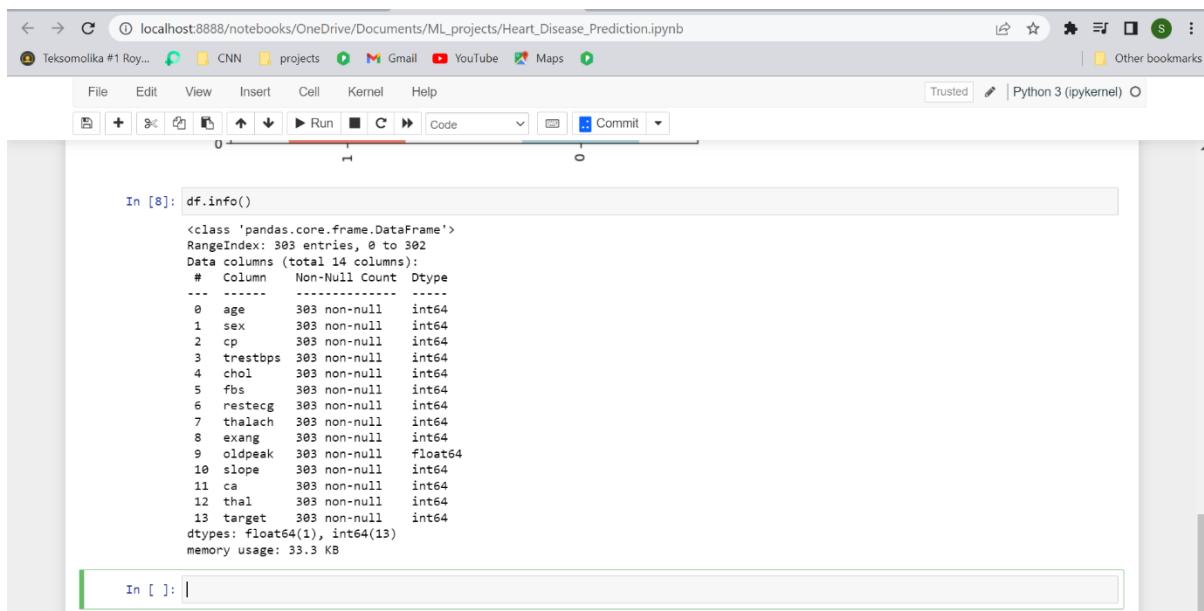
We can plot the target column value counts by calling the plot() function and telling it what kind of plot we'd like, in this case, bar is good.

In [7]:



df.info() shows a quick insight to the number of missing values you have and what type of data you're working with.

In our case, there are no missing values and all of our columns are numerical in nature.



Another way to get some quick insights on your dataframe is to use df.describe().

describe() shows a range of different metrics about your numerical columns such as mean, max and standard deviation.

```
In [9]: df.describe()
Out[9]:
   age      sex      cp      trestbps      chol      fbs      restecg      thalach      exang      oldpeak      slope      ca
count  303.000000  303.000000  303.000000  303.000000  303.000000  303.000000  303.000000  303.000000  303.000000  303.000000  303.000000
mean   54.366337  0.683168  0.966997  131.623762  246.264026  0.148515  0.528053  149.646865  0.326733  1.039604  1.399340  0.729373  2.31
std    9.082101  0.466011  1.032052  17.538143  51.830751  0.356198  0.525860  22.905161  0.469794  1.161075  0.616226  1.022606  0.61
min   28.000000  0.000000  0.000000  94.000000  126.000000  0.000000  0.000000  71.000000  0.000000  0.000000  0.000000  0.000000  0.00
25%   47.500000  0.000000  0.000000  120.000000  211.000000  0.000000  0.000000  133.500000  0.000000  0.000000  1.000000  0.000000  2.00
50%   55.000000  1.000000  1.000000  130.000000  240.000000  0.000000  1.000000  153.000000  0.000000  0.800000  1.000000  0.000000  2.00
75%   61.000000  1.000000  2.000000  140.000000  274.500000  0.000000  1.000000  166.000000  1.000000  1.600000  2.000000  1.000000  3.00
max   77.000000  1.000000  3.000000  200.000000  564.000000  1.000000  2.000000  202.000000  1.000000  6.200000  2.000000  4.000000  3.00
```

Heart Disease Frequency according to Gender

If you want to compare two columns to each other, you can use the function pd.crosstab(column_1, column_2).

This is helpful if you want to start gaining an intuition about how your independent variables interact with your dependent variables.

Let's compare our target column with the sex column.

Remember from our data dictionary, for the target column, 1 = heart disease present, 0 = no heart disease. And for sex, 1 = male, 0 = female.

The screenshot shows a Jupyter Notebook interface with the following details:

- Toolbar:** File, Edit, View, Insert, Cell, Kernel, Help, Trusted, Python 3 (ipykernel).
- In [10]:** df.sex.value_counts()
Out[10]:

1	207
0	96
Name: sex, dtype: int64	
- In [11]:** # Compare target column with sex column
pd.crosstab(df.target, df.sex)
Out[11]:

sex	0	1
target		
0	24	114
1	72	93
- In []:** (empty cell)

There are 207 males and 96 females in our study.

What can we infer from this? Let's make a simple heuristic.

Since there are about 100 women and 72 of them have a positive value of heart disease being present, we might infer, based on this one variable if the participant is a woman, there's a 75% chance she has heart disease.

As for males, there's about 200 total with around half indicating a presence of heart disease. So we might predict, if the participant is male, 50% of the time he will have heart disease.

Averaging these two values, we can assume, based on no other parameters, if there's a person, there's a 62.5% chance they have heart disease.

This can be our very simple **baseline**, we'll try to beat it with machine learning.

Making our crosstab visual

You can plot the crosstab by using the plot() function and passing it a few parameters such as, kind (the type of plot you want), figsize=(length, width) (how big you want it to be) and color=[colour_1, colour_2] (the different colours you'd like to use).

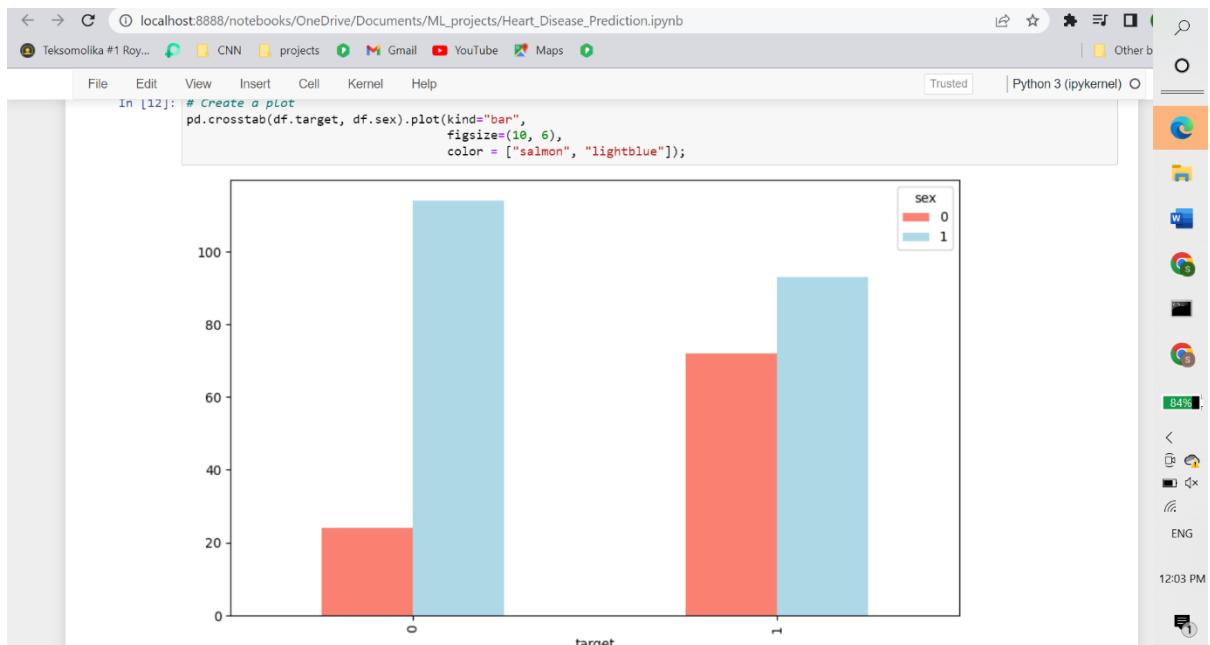
Different metrics are represented best with different kinds of plots. In our case, a bar graph is great. We'll see examples of more later. And with a bit of practice, you'll gain an intuition of which plot to use with different variables.

Nice! But our plot is looking pretty bare. Let's add some attributes.

We'll create the plot again with crosstab() and plot(), then add some helpful labels to it with plt.title(), plt.xlabel() and more.

To add the attributes, you call them on plt within the same cell as where you make create the graph.

```
In [13]: 1 # Create a plot
2 pd.crosstab(df.target, df.sex).plot(kind = "bar",
3             figsize = (10,6),
4             color = ["salmon", "lightblue"])
5 # Add some attributes to it
6 plt.title("Heart Disease Frequency For Sex")
7 plt.xlabel("0 = No Disease, 1 = Disease")
8 plt.ylabel("Amount")
9 plt.legend(["Female", "Male"])
10 plt.xticks(rotation = 0); # keep the labels on the x axis vertical
```



Age vs Max Heart rate for Heart Disease

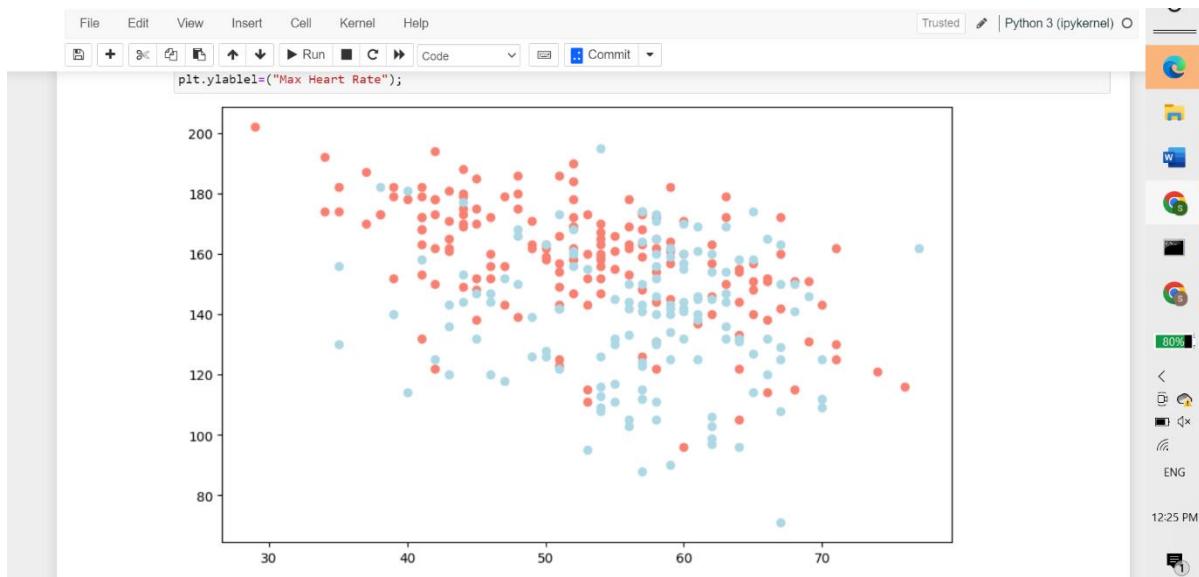
Let's try combining a couple of independent variables, such as, age and thalach (maximum heart rate) and then comparing them to our target variable heart disease.

Because there are so many different values for age and thalach, we'll use a scatter plot.

```
In [14]: # Create another figure
plt.figure(figsize=(10,6))

# Start with positive examples
plt.scatter(df.age[df.target==1],
            df.thalach[df.target==1],
            c = "salmon") # define it as scatter figure

# Now for negative examples come we want them on same plot so, we call plt again
plt.scatter(df.age[df.target==0],
            df.thalach[df.target==0],
            c = "lightblue") # axis always come as x, y
# Add some helpful info
plt.title("Heart Disease in function of Age and Max Heart Rate")
plt.xlabel("Age")
plt.legend(["Disease", "No Disease"])
plt.ylabel("Max Heart Rate");
```

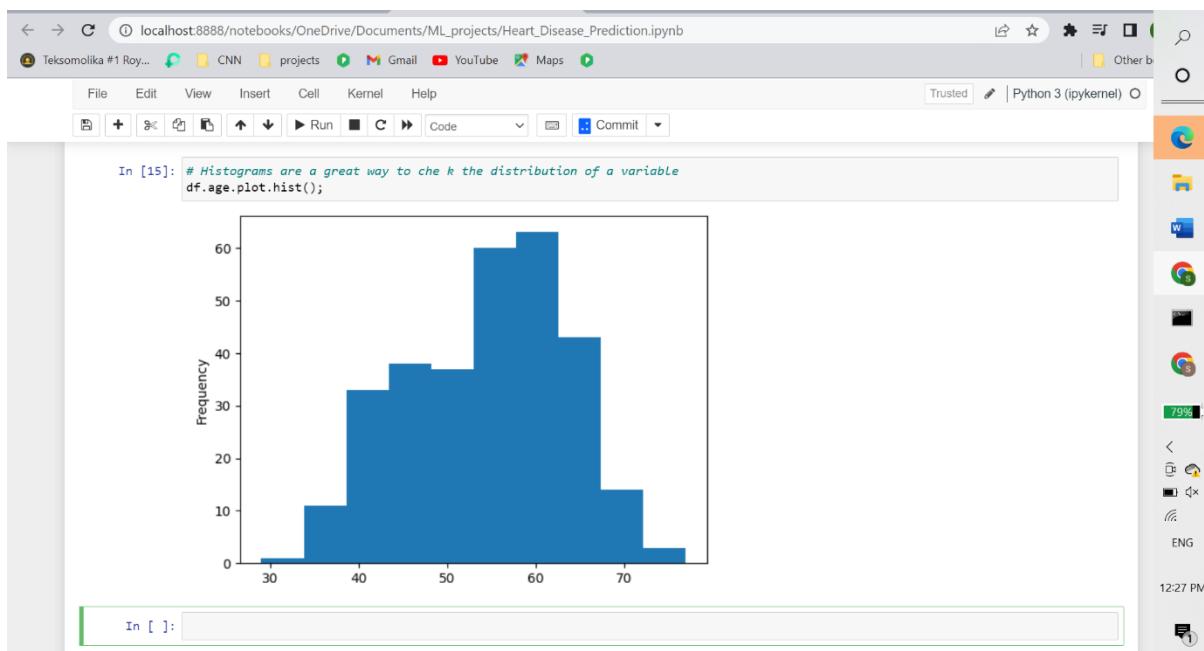


what can we infer from this?

It seems the younger someone is, the higher their max heart rate (dots are higher on the left of the graph) and the older someone is, the more green dots there are. But this may be because there are more dots all together on the right side of the graph (older participants).

Both of these are observational of course, but this is what we're trying to do, build an understanding of the data.

Let's check the **age distribution**.



We can see it's a **normal distribution** but slightly swaying to the right, which reflects in the scatter plot above.

Let's keep going.

Heart Disease Frequency per Chest Pain Type

Let's try another independent variable. This time, cp (chest pain).

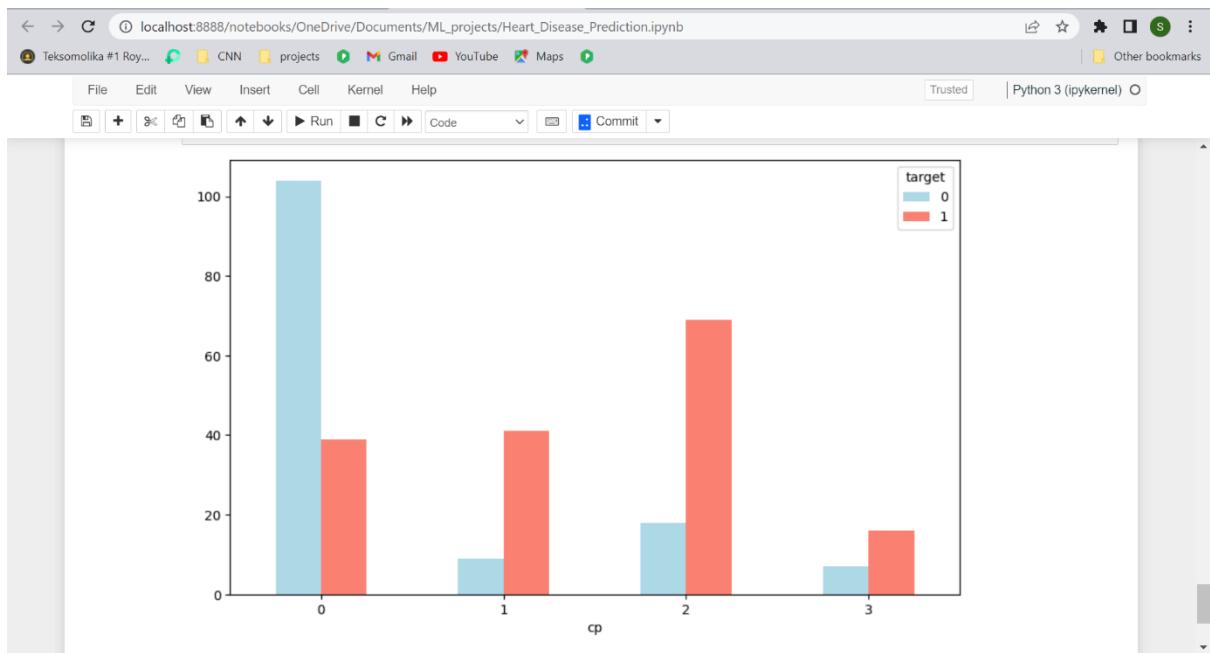
We'll use the same process as we did before with sex.

The screenshot shows a Jupyter Notebook interface with two code cells. The first cell (In [16]) contains the command `pd.crosstab(df.cp, df.target)`. The output (Out[16]) is a table:

target	0	1
cp		
0	104	39
1	9	41
2	18	69
3	7	16

The second cell (In [17]) contains Python code to create a bar plot:

```
# Create a new crosstab and baseplot
pd.crosstab(df.cp, df.target).plot(kind="bar",
                                         figsize=(10,6),
                                         color=[["lightblue", "salmon"]])
#Add attributes to the plot to make it more readable
plt.title("Heart Disease Frequency Per Chest Pain Type")
plt.xlabel("Chest Pain Type")
plt.ylabel("Frequency")
plt.legend(["No Disease", "Disease"])
plt.xticks(rotation=0);
```



What can we infer from this?

Remember from our data dictionary what the different levels of chest pain are.

1. cp - chest pain type
 - 0: Typical angina: chest pain related decrease blood supply to the heart
 - 1: Atypical angina: chest pain not related to heart
 - 2: Non-anginal pain: typically oesophageal spasms (non-heart related)
 - 3: Asymptomatic: chest pain not showing signs of disease

It's interesting the atypical angina (value 1) states it's not related to the heart but seems to have a higher ratio of participants with heart disease than not.

Wait...?

What does a typical angina even mean?

At this point, it's important to remember, if your data dictionary doesn't supply you enough information, you may want to do further research on your values. This research may come in the form of asking a **subject matter expert** (such as a cardiologist or the person who gave you the data) or googling to find out more.

Today, 23 years later, "atypical chest pain" is still popular in medical circles. Its meaning, however, remains unclear. A few articles have the term in their title, but do not define or discuss it in their text. In other articles, the term refers to noncardiac causes of chest pain.

Although not conclusive, this graph above is a hint at the confusion of definitions being represented in data.

Correlation between independent variables

Finally, we'll compare all of the independent variables in one hit.

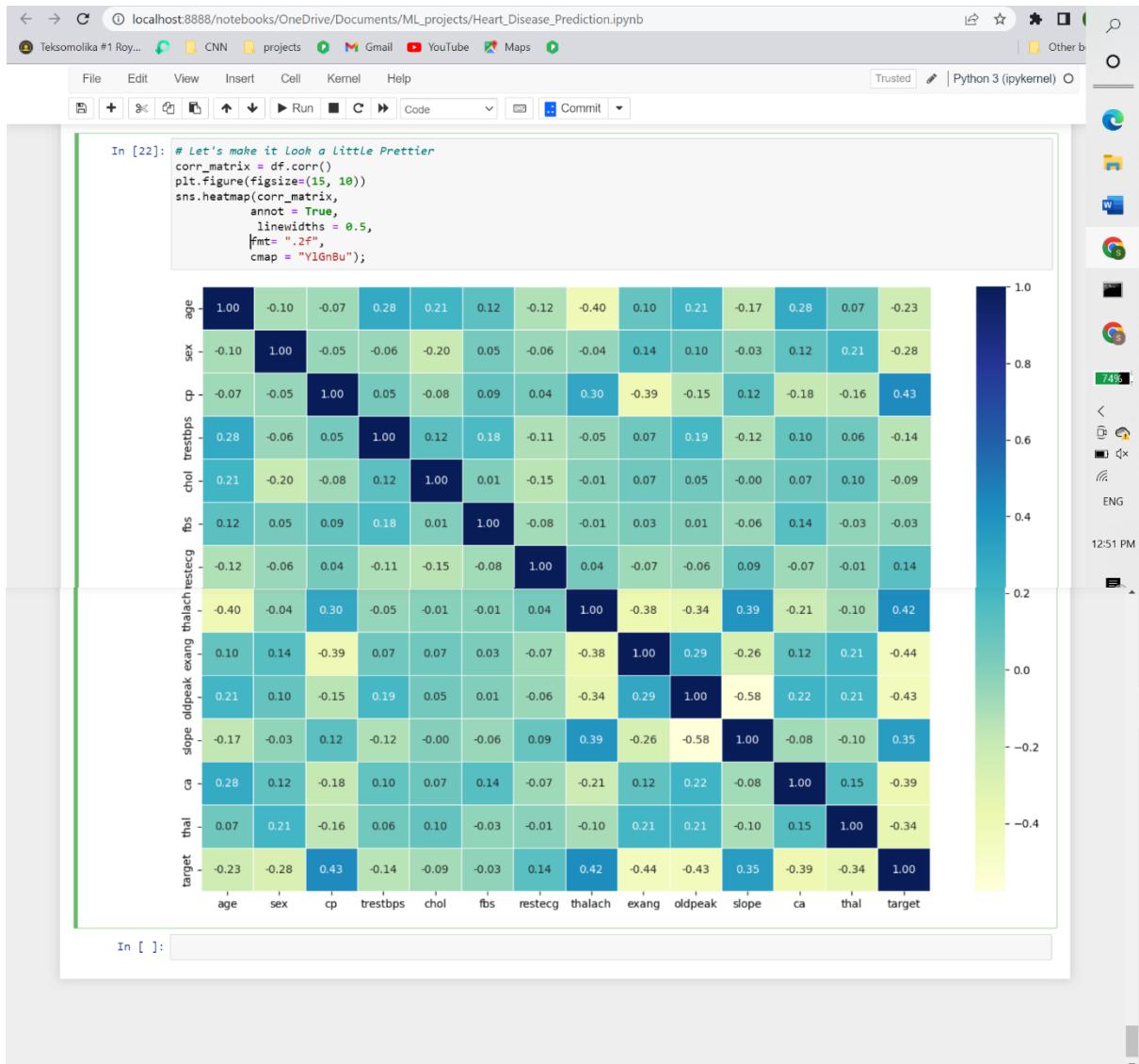
Why?

Because this may give an idea of which independent variables may or may not have an impact on our target variable.

We can do this using df.corr() which will create a **correlation matrix** for us, in other words, a big table of numbers telling us how related each variable is the other.

The screenshot shows a Jupyter Notebook interface with several tabs at the top: 'Create a New Repository', 'zero-to-mastery-ml/end...', 'Heart_Disease_Prediction', 'OneDrive/Documents/ML_projects/Heart_Disease_Prediction.ipynb' (which is the active tab), and 'How to take full webpage...'. Below the tabs is a toolbar with icons for file operations like 'File', 'Edit', 'View', 'Insert', 'Cell', 'Kernel', 'Help', and a 'Run' button. The main area contains two code cells. The first cell, 'In [18]', contains the Python code: '# Find the correlation between our independent variables corr_matrix = df.corr() corr_matrix'. The second cell, 'Out[18]', displays the resulting correlation matrix as a table:

	age	sex	cp	trestbps	chol	fbs	restecg	thalach	exang	oldpeak	slope	ca	thal	target
age	1.000000	-0.098447	-0.068653	0.279351	0.213678	0.121308	-0.116211	-0.398522	0.096801	0.210013	-0.168814	0.276326	0.068001	-0.225439
sex	-0.098447	1.000000	-0.049353	-0.056769	-0.197912	0.045032	-0.058196	-0.044020	0.141664	0.096093	-0.030711	0.118261	0.210041	-0.280937
cp	-0.068653	-0.049353	1.000000	0.047608	-0.076904	0.094444	0.044421	0.295762	-0.394280	-0.149230	0.119717	-0.181053	-0.161736	0.433798
trestbps	0.279351	-0.056769	0.047608	1.000000	0.123174	0.177531	-0.114103	-0.046698	0.067616	0.193216	-0.121475	0.101389	0.062210	-0.144931
chol	0.213678	-0.197912	-0.076904	0.123174	1.000000	0.013294	-0.151040	-0.009940	0.067023	0.053952	-0.004030	0.070511	0.098803	-0.085239
fbs	0.121308	0.045032	0.094444	0.177531	0.013294	1.000000	-0.084189	-0.008567	0.025665	0.005747	-0.059894	0.137979	-0.032019	-0.028046
restecg	-0.116211	-0.058196	0.044421	-0.114103	-0.151040	-0.084189	1.000000	0.044123	-0.070733	-0.058770	0.093045	-0.072042	-0.011981	0.137230
thalach	-0.398522	-0.044020	0.295762	-0.046698	-0.008940	-0.008567	0.044123	1.000000	-0.378812	-0.344187	0.386784	-0.213177	-0.096439	0.421741
exang	0.096801	0.141664	-0.394280	0.067616	0.067023	0.025665	-0.070733	-0.378812	1.000000	0.288223	-0.257748	0.115739	0.206754	-0.436757
oldpeak	0.210013	0.096093	-0.149230	0.193216	0.053952	0.005747	-0.058770	-0.344187	0.288223	1.000000	-0.577537	0.222682	0.210244	-0.430696
slope	-0.168814	-0.030711	0.119717	-0.121475	-0.004038	-0.059894	0.093045	0.386784	-0.257748	-0.577537	1.000000	-0.080155	-0.104764	0.345877
ca	0.276326	0.118261	-0.181053	0.101389	0.070511	0.137979	-0.072042	-0.213177	0.115739	0.222682	-0.080155	1.000000	0.151832	-0.391724
thal	0.068001	0.210041	-0.161736	0.062210	0.098803	-0.032019	-0.011981	-0.096439	0.206754	0.210244	-0.104764	0.151832	1.000000	-0.344029
target	-0.225439	-0.280937	0.433798	-0.144931	-0.085239	-0.028046	0.137230	0.421741	-0.436757	-0.430696	0.345877	-0.391724	-0.344029	1.000000



Much better. A higher positive value means a potential positive correlation (increase) and a higher negative value means a potential negative correlation (decrease).

Enough EDA, let's model

Remember, we do exploratory data analysis (EDA) to start building an intuition of the dataset.

What have we learned so far? Aside from our baseline estimate using sex, the rest of the data seems to be pretty distributed.

So what we'll do next is **model driven EDA**, meaning, we'll use machine learning models to drive our next questions. A few extra things to remember:

- Not every EDA will look the same, what we've seen here is an example of what you could do for structured, tabular dataset.
- You don't necessarily have to do the same plots as we've done here, there are many more ways to visualize data, I encourage you to look at more.
- We want to quickly find:
 - Distributions (`df.column.hist()`)
 - Missing values (`df.info()`)
 - Outliers

Let's build some models.

5. Modeling

We've explored the data, now we'll try to use machine learning to predict our target variable based on the 13 independent variables.

Remember our problem?

Given clinical parameters about a patient, can we predict whether or not they have heart disease?

That's what we'll be trying to answer.

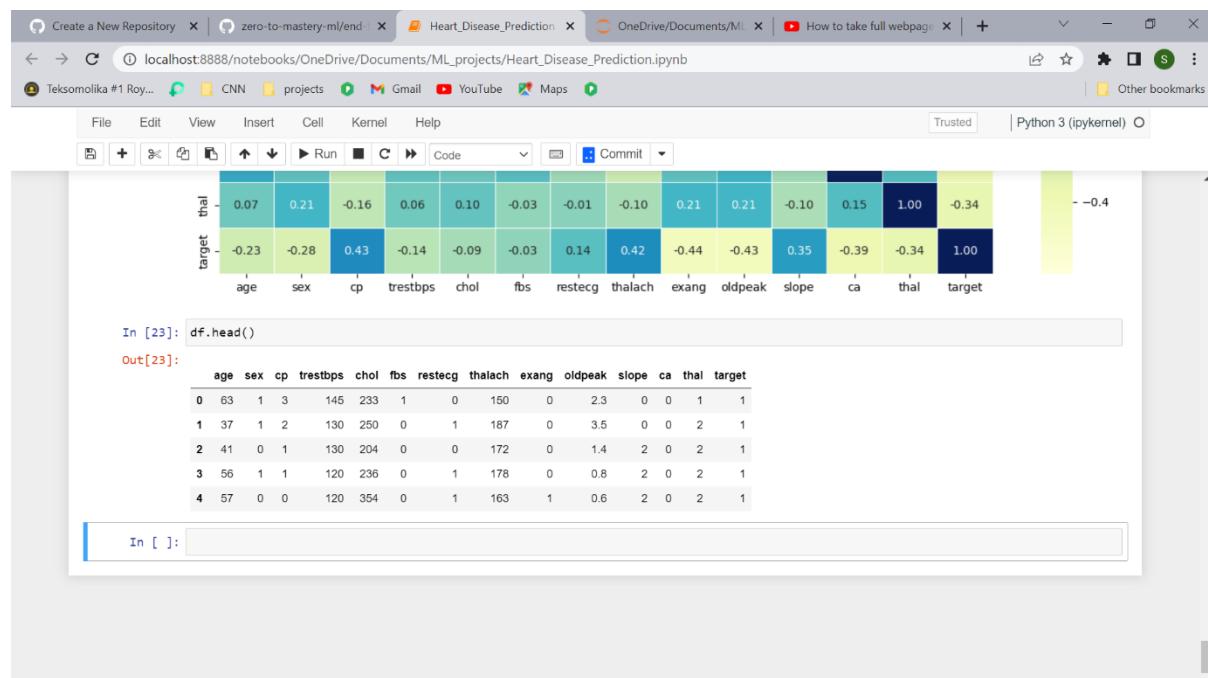
And remember our evaluation metric?

If we can reach 95% accuracy at predicting whether or not a patient has heart disease during the proof of concept, we'll pursue this project.

That's what we'll be aiming for.

But before we build a model, we have to get our dataset ready.

Let's look at it again.



We're trying to predict our target variable using all of the other variables.

To do this, we'll split the target variable from the rest.

```
In [21]: # Everything except target variable  
X = df.drop("target", axis=1)  
  
# Target variable  
y = df.target.values
```

Let's see our new variables.

```
In [22]: # Independent variables (no target column)
X.head()

Out[22]:
   age sex cp trestbps chol fbs restecg thalach exang oldpeak slope ca thal
0  63   1   3    145  233   1     0    150   0    2.3   0  0   1
1  37   1   2    130  250   0     1    187   0    3.5   0  0   2
2  41   0   1    130  204   0     0    172   0    1.4   2  0   2
3  56   1   1    120  236   0     1    178   0    0.8   2  0   2
4  57   0   0    120  354   0     1    163   1    0.6   2  0   2
```



```
In [23]: # Targets
y

Out[23]: 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])
```

Training and Test split

Now comes one of the most important concepts in machine learning, the **training/test split**.

This is where you'll split your data into a **training set** and a **test set**.

You use your training set to train your model and your test set to test it.

The test set must remain separate from your training set.

Why not use all the data to train a model?

Let's say you wanted to take your model into the hospital and start using it on patients. How would you know how well your model goes on a new patient not included in the original full dataset you had?

This is where the test set comes in. It's used to mimic taking your model to a real environment as much as possible.

And it's why it's important to never let your model learn from the test set, it should only be evaluated on it.

To split our data into a training and test set, we can use Scikit-Learn's [train_test_split\(\)](#) and feed it our independent and dependent variables (X & y).

```
In [29]: # Random seed for reproducability
np.random.seed(42)

#Split into train and test test
X_train, X_test, y_train, y_test = train_test_split(X, # independent variable
                                                    y, # dependent variable
                                                    test_size = 0.2) # percentage of data to use for test set
```

The `test_size` parameter is used to tell the `train_test_split()` function how much of our data we want in the test set.

A rule of thumb is to use 80% of your data to train on and the other 20% to test on.

For our problem, a train and test set are enough. But for other problems, you could also use a validation (train/validation/test) set or cross-validation (we'll see this in a second).

Let's look at our training data.

The screenshot shows a Jupyter Notebook interface with two windows side-by-side. Both windows have the same URL: `localhost:8888/notebooks/OneDrive/Documents/ML_projects/Heart_Disease_Prediction.ipynb`. The top window displays code cells for `X_train`, `y_train`, and `X_test`. The bottom window displays the same code cells. The code in cell [35] shows the head of the `X_train` dataset:

```
In [35]: X_train.head()
Out[35]:
   age  sex  cp  trestbps  chol  fbs  restecg  thalach  exang  oldpeak  slope  ca  thal
132  42    1    1  120    295    0     1    162     0    0.0     2    0    2
202  58    1    0  150    270    0     0    111     1    0.8     2    0    3
196  46    1    2  150    231    0     1    147     0    3.6     1    0    2
  75  55    0    1  135    250    0     0    161     0    1.4     1    0    2
176  60    1    0  117    230    1     1    160     1    1.4     2    2    3
```

The code in cell [36] shows the length of `y_train`:

```
In [36]: len(y_train)
Out[36]: (array([1, 0, 0, 1, 0, 1, 1, 0, 1, 0, 1, 0, 1, 0, 0, 1, 1, 0, 0,
       1, 1, 0, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 0,
       1, 1, 1, 0, 1, 0, 0, 1, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1,
       0, 0, 0, 1, 1, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 1, 0,
       0, 1, 1, 1, 0, 0, 1, 0, 1, 0, 0, 1, 0, 1, 0, 1, 1, 1, 0, 0,
       0, 1, 0, 1, 0, 0, 1, 1, 0, 1, 0, 1, 0, 0, 1, 0, 1, 0, 1, 0, 1,
       0, 1, 0, 1, 0, 0, 0, 1, 1, 0, 0, 1, 0, 1, 0, 1, 0, 0, 1, 0, 1,
       1, 0, 1, 0, 0, 0, 1, 0, 0, 1, 0, 1, 0, 1, 1, 0, 0, 0, 1, 0, 1,
       1, 0, 1, 0, 0, 1, 1, 0, 1, 0, 0, 1, 1, 0, 0, 1, 1, 0, 0, 0, 1, 0,
       0, 0, 0, 1, 1, 0, 0, 0, 0, 1, 1, 0, 1, 0, 0, 1, 0, 1, 1, 1, 1,
       1, 0, 1, 1, 0, 0, 0, 1, 0, 1, 0, 0, 0, 1, 0, 1, 0, 0, 0, 1, 0, 1,
       1, 0, 1, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 1, 0, 0, 0, 1, 0, 1, 0, 1,
       0, 0, 1, 1, 0, 0, 0, 0, 1, 1, 0, 0, 1, 0, 0, 1, 1, 0, 0, 0, 1, 0,
       0, 0, 1, 1, 0, 0, 0, 0, 1, 1, 0, 0, 1, 0, 0, 1, 1, 0, 0, 0, 0, 1,
       1, 0, 1, 1, 0, 0, 0, 1, 1, 0, 0, 1, 0, 0, 1, 1, 0, 0, 0, 0, 0, 1,
       1, 0, 0, 1, 1, 0, 0, 0, 1, 1, 0, 0, 1, 0, 0, 1, 1, 0, 0, 0, 0, 1],
      dtype=int64),
  242)
```

The code in cell [37] shows the head of the `X_test` dataset:

```
In [37]: X_test.head()
Out[37]:
   age  sex  cp  trestbps  chol  fbs  restecg  thalach  exang  oldpeak  slope  ca  thal
179  57    1    0  150    276    0     0    112     1    0.6     1    1    1
228  59    1    3  170    288    0     0    159     0    0.2     1    0    3
111  57    1    2  150    126    1     1    173     0    0.2     2    1    3
246  56    0    0  134    409    0     0    150     1    1.9     1    2    3
  60  71    0    2  110    265    1     0    130     0    0.0     2    1    2
```

The code in cell [38] shows the length of `y_test`:

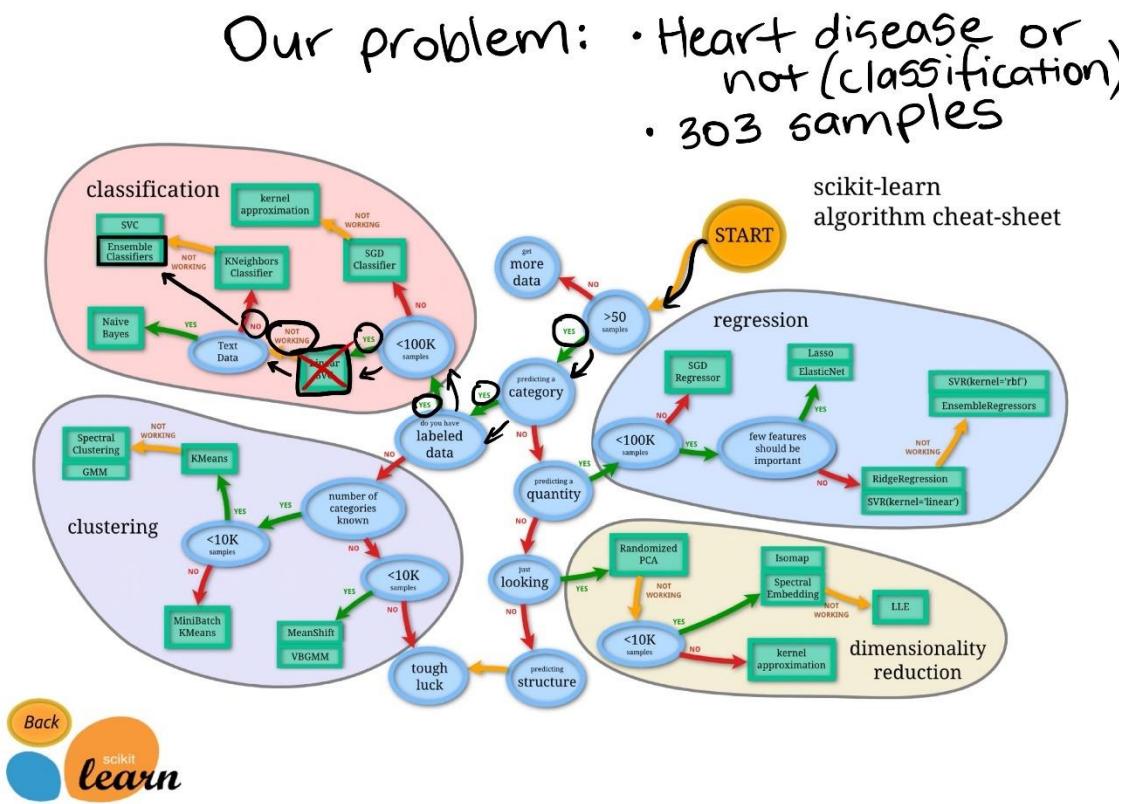
```
In [38]: len(y_test)
Out[38]: (array([0, 0, 1, 0, 1, 1, 0, 0, 1, 1, 1, 0, 1, 1, 0, 0, 0, 1, 0,
       0, 1, 1, 0, 0, 1, 0, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 0, 0, 1, 0, 0, 0, 1, 1, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0],
      dtype=int64),
  61)
```

The bottom window has an empty cell labeled `In []:`.

Model choices

Now we've got our data prepared, we can start to fit models. We'll be using the following and comparing their results.

1. Logistic Regression - `LogisticRegression()`
2. K-Nearest Neighbors - `KNeighborsClassifier()`



An example path we can take using the Scikit-Learn Machine Learning Map

"Wait, I don't see Logistic Regression and why not use LinearSVC?"

Good questions.

And as for LinearSVC, let's pretend we've tried it, and it doesn't work, so we're following other options in the map.

For now, knowing each of these algorithms inside and out is not essential.

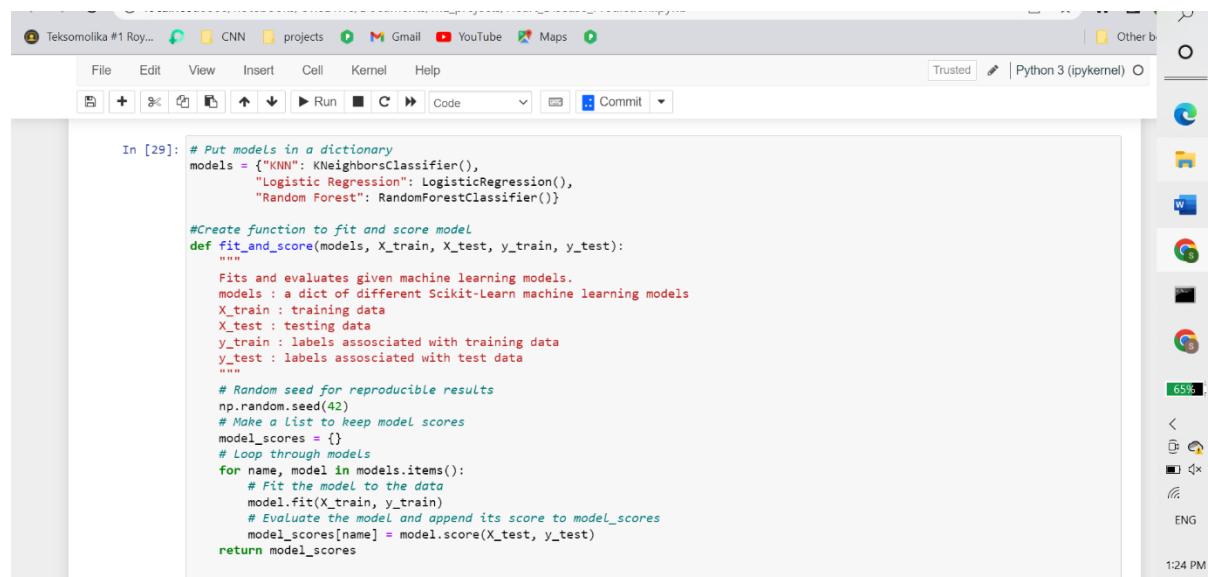
Machine learning and data science is an iterative practice. These algorithms are tools in your toolbox.

In the beginning, on your way to becoming a practitioner, it's more important to understand your problem (such as, classification versus regression) and then knowing what tools you can use to solve it.

Since our dataset is relatively small, we can experiment to find algorithm performs best.

All of the algorithms in the Scikit-Learn library use the same functions, for training a model, `model.fit(X_train, y_train)` and for scoring a model `model.score(X_test, y_test)`. `score()` returns the ratio of correct predictions (1.0 = 100% correct).

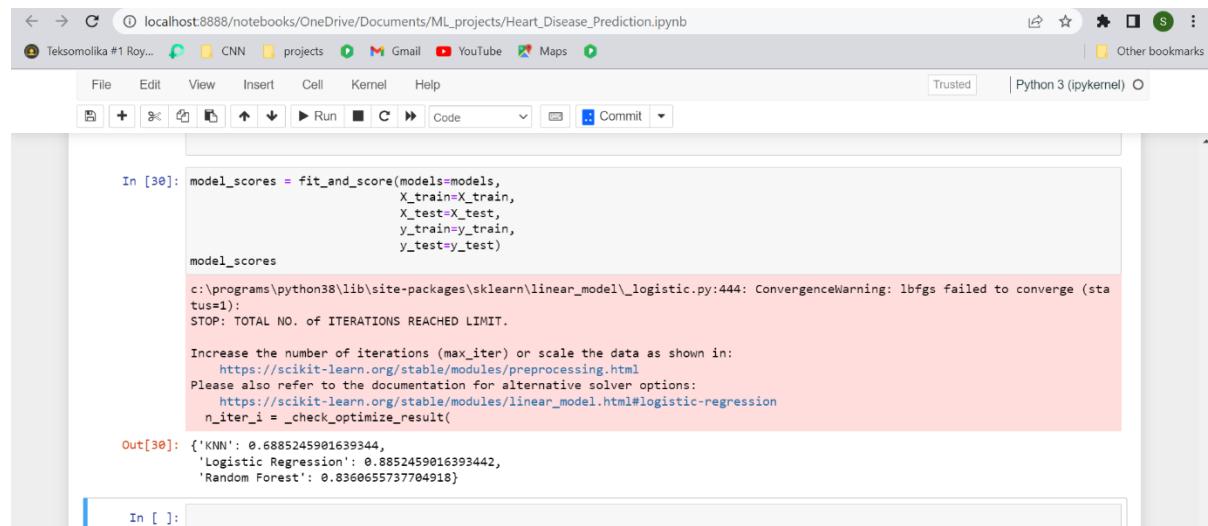
Since the algorithms we've chosen implement the same methods for fitting them to the data as well as evaluating them, let's put them in a dictionary and create a which fits and scores them.



The screenshot shows a Jupyter Notebook interface with two code cells. The first cell (In [29]) contains Python code to define a dictionary of models and a function to fit and score them. The second cell (In [30]) runs the function and prints the resulting scores.

```
# Put models in a dictionary
models = {"KNN": KNeighborsClassifier(),
          "Logistic Regression": LogisticRegression(),
          "Random Forest": RandomForestClassifier()}

#Create function to fit and score model
def fit_and_score(models, X_train, X_test, y_train, y_test):
    """
    Fits and evaluates given machine learning models.
    models : a dict of different Scikit-Learn machine learning models
    X_train : training data
    X_test : testing data
    y_train : labels associated with training data
    y_test : labels associated with test data
    """
    # Random seed for reproducible results
    np.random.seed(42)
    # Make a list to keep model scores
    model_scores = {}
    # Loop through models
    for name, model in models.items():
        # Fit the model to the data
        model.fit(X_train, y_train)
        # Evaluate the model and append its score to model_scores
        model_scores[name] = model.score(X_test, y_test)
    return model_scores
```

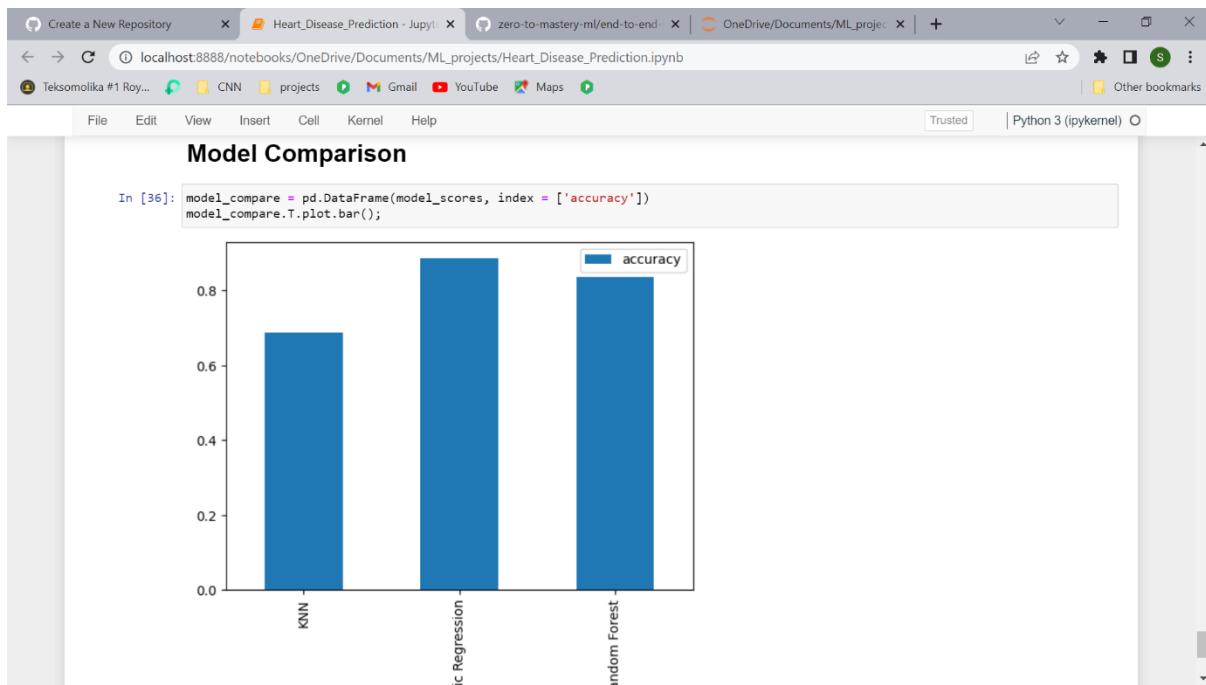



The screenshot shows the execution of the code from In [30]. It includes a warning message about convergence and the resulting scores for each model.

```
In [30]: model_scores = fit_and_score(models=models,
                                     X_train=X_train,
                                     X_test=X_test,
                                     y_train=y_train,
                                     y_test=y_test)
model_scores
c:\programs\python38\lib\site-packages\sklearn\linear_model\_logistic.py:444: ConvergenceWarning: lbfgs failed to converge (status=1):
STOP: TOTAL NO. OF ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression
n_iter_i = _check_optimize_result()

Out[30]: {'KNN': 0.6885245901639344,
          'Logistic Regression': 0.8852459016393442,
          'Random Forest': 0.8360655737704918}
```

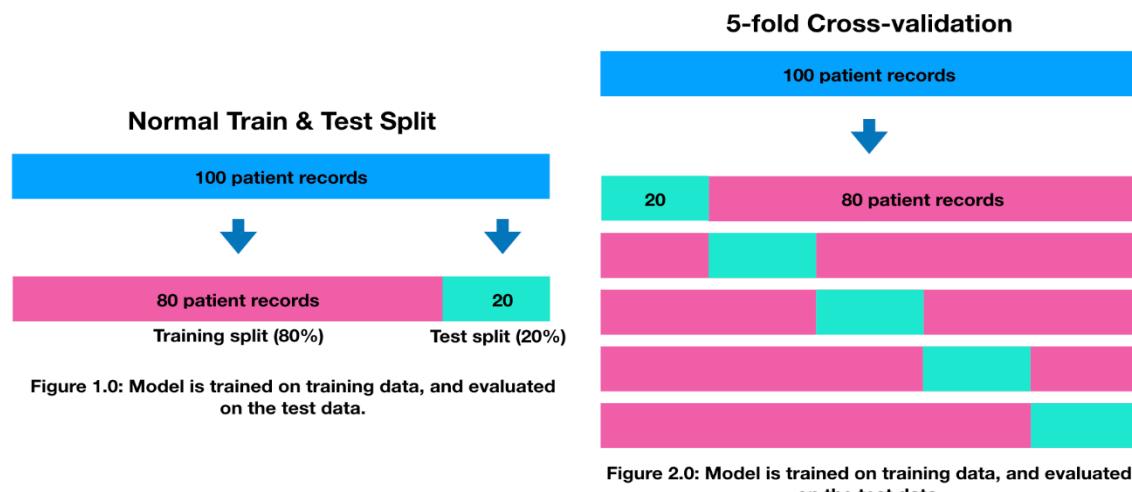


Let's briefly go through each before we see them in action.

- **Hyperparameter tuning** - Each model you use has a series of dials you can turn to dictate how they perform. Changing these values may increase or decrease model performance.
- **Feature importance** - If there are a large amount of features we're using to make predictions, do some have more importance than others? For example, for predicting heart disease, which is more important, sex or age?
- **Confusion matrix** - Compares the predicted values with the true values in a tabular way, if 100% correct, all values in the matrix will be top left to bottom right (diagnol line).
- **Cross-validation** - Splits your dataset into multiple parts and train and tests your model on each part and evaluates performance as an average.
- **Precision** - Proportion of true positives over total number of samples. Higher precision leads to less false positives.
- **Recall** - Proportion of true positives over total number of true positives and false negatives. Higher recall leads to less false negatives.
- **F1 score** - Combines precision and recall into one metric. 1 is best, 0 is worst.
- **Classification report** - Sklearn has a built-in function called `classification_report()` which returns some of the main classification metrics such as precision, recall and f1-score.
- **ROC Curve** - [Receiver Operating Characterisitc](#) is a plot of true positive rate versus false positive rate.
- **Area Under Curve (AUC)** - The area underneath the ROC curve. A perfect model achieves a score of 1.0.

Hyperparameter tuning and cross-validation

- To cook your favourite dish, you know to set the oven to 180 degrees and turn the grill on. But when your roommate cooks their favourite dish, they set use 200 degrees and the fan-forced mode. Same oven, different settings, different outcomes.
- The same can be done for machine learning algorithms. You can use the same algorithms but change the settings (hyperparameters) and get different results.
- But just like turning the oven up too high can burn your food, the same can happen for machine learning algorithms. You change the settings and it works so well, it **overfits** (does too well) the data.
- We're looking for the goldilocks model. One which does well on our dataset but also does well on unseen examples.
- To test different hyperparameters, you could use a **validation set** but since we don't have much data, we'll use **cross-validation**.
- The most common type of cross-validation is *k-fold*. It involves splitting your data into *k-fold's* and then testing a model on each. For example, let's say we had 5 folds ($k = 5$). This what it might look like.



Normal train and test split versus 5-fold cross-validation

We'll be using this setup to tune the hyperparameters of some of our models and then evaluate them. We'll also get a few more metrics like **precision**, **recall**, **F1-score** and **ROC** at the same time. Here's the game plan:

1. Tune model hyperparameters, see which performs best
2. Perform cross-validation
3. Plot ROC curves
4. Make a confusion matrix
5. Get precision, recall and F1-score metrics
6. Find the most important model features

Tuning models with with KNeighborsClassifier

There's one main hyperparameter we can tune for the K-Nearest Neighbors (KNN) algorithm, and that is number of neighbours. The default is 5 (n_neighbors=5).

What are neighbours?

Imagine all our different samples on one graph like the scatter graph we have above. KNN works by assuming dots which are closer together belong to the same class. If n_neighbors=5 then it assume a dot with the 5 closest dots around it are in the same class.

We've left out some details here like what defines close or how distance is calculated but I encourage you to research them.

For now, let's try a few different values of n neighbors.



```
In [37]: # # Create a list of train scores
train_scores = []

# Create a List of test scores
test_scores = []

# Create a List of different values for n_neighbors
neighbors = range(1, 21) # 1 to 20

# Setup algorithm
knn = KNeighborsClassifier()

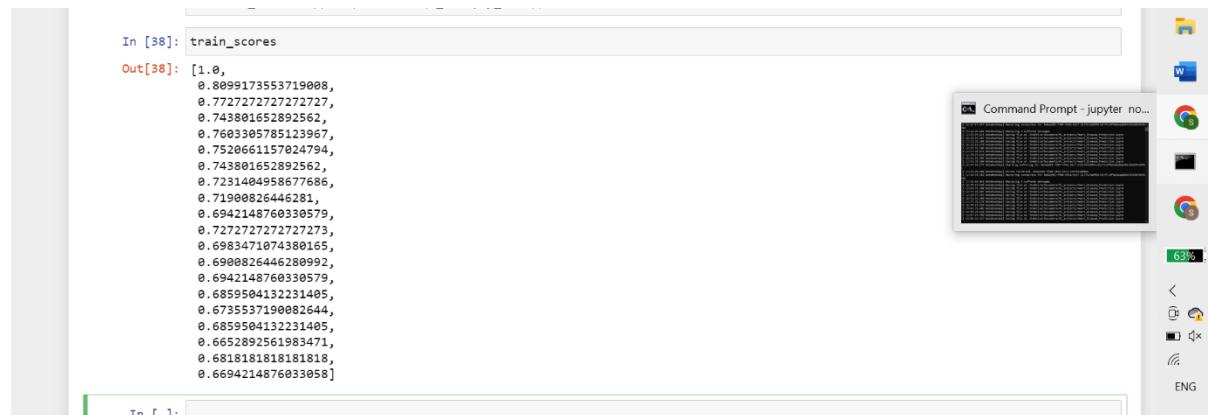
# Loop through different neighbors values
for i in neighbors:
    knn.set_params(n_neighbors = i) # set neighbors value

    # Fit the algorithm
    knn.fit(X_train, y_train)

    # Update the training scores
    train_scores.append(knn.score(X_train, y_train))

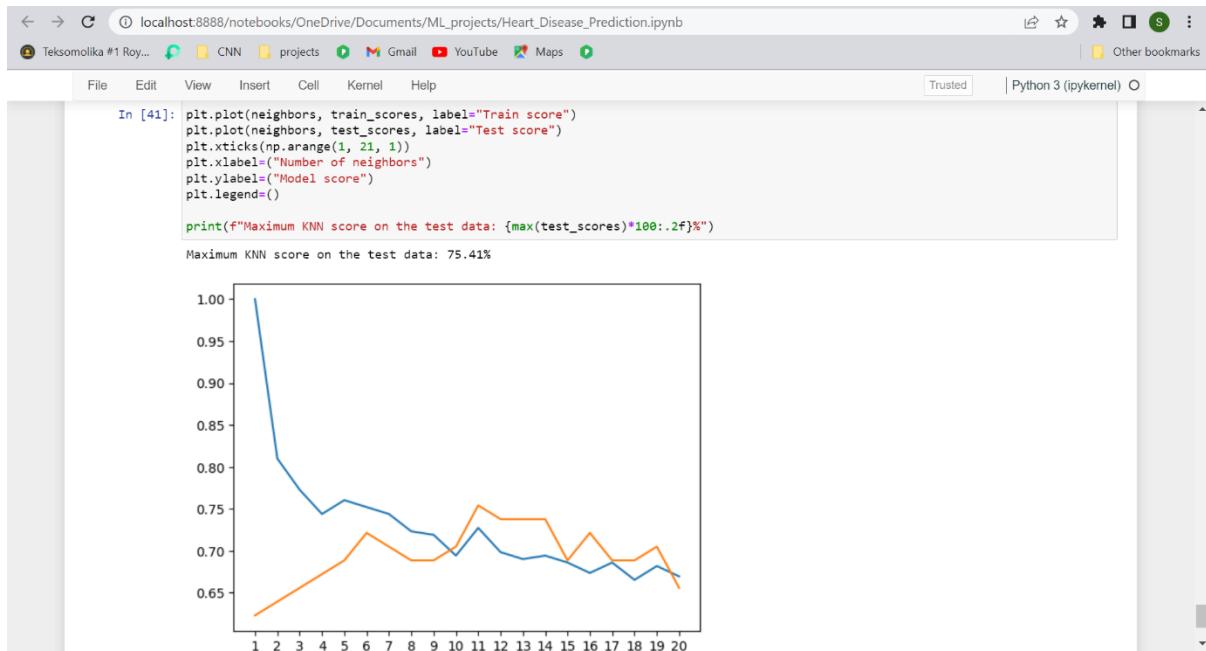
    # Update the test scores
    test_scores.append(knn.score(X_test, y_test))
```

Let's look at KNN's train scores.



```
In [38]: train_scores
Out[38]: [1.0,
 0.8099173553719008,
0.7727272727272727,
0.743801652892562,
0.7603305785123967,
0.7526661157024794,
0.743801652892562,
0.7231404958677686,
0.71900826446281,
0.6942148760330579,
0.7272727272727273,
0.69834710743880165,
0.6900826446280992,
0.6942148760330579,
0.6859504132231405,
0.6735537190082644,
0.6859504132231405,
0.6652892561983471,
0.6818181818181818,
0.6694214876033058]
```

Let's Plot it:



Looking at the graph, $n_{neighbors} = 11$ seems best.

Even knowing this, the KNN's model performance didn't get near what LogisticRegression or the RandomForestClassifier did.

Because of this, we'll discard KNN and focus on the other two.

We've tuned KNN by hand but let's see how we can LogisticsRegression and RandomForestClassifier using [RandomizedSearchCV](#).

Instead of us having to manually try different hyperparameters by hand, RandomizedSearchCV tries a number of different combinations, evaluates them and saves the best.

Tuning models with with RandomizedSearchCV

Reading the Scikit-Learn documentation for [LogisticRegression](#), we find there's a number of different hyperparameters we can tune.

The same for [RandomForestClassifier](#).

Let's create a hyperparameter grid (a dictionary of different hyperparameters) for each and then test them out.

Now let's use RandomizedSearchCV to try and tune our LogisticRegression model.

We'll pass it the different hyperparameters from `log_reg_grid` as well as set `n_iter = 20`. This means, RandomizedSearchCV will try 20 different combinations of hyperparameters from `log_reg_grid` and save the best ones.

```
In [42]: # Different LogisticRegression hyperparameters
log_reg_grid = {"C": np.logspace(-4, 4, 20),
                 "solver": ["liblinear"]}

# Different RandomForestClassifier hyperparameters
rf_grid = {"n_estimators": np.arange(10, 1000, 50),
            "max_depth": [None, 3, 5, 10],
            "min_samples_split": np.arange(2, 20, 2),
            "min_samples_leaf": np.arange(1, 20, 2)}

In [43]: # Setup random seed
np.random.seed(42)

# Setup random hyperparameter search for LogisticRegression
rs_log_reg = RandomizedSearchCV(LogisticRegression(),
                                  param_distributions=log_reg_grid,
                                  cv=5,
                                  n_iter=20,
                                  verbose=True)

# Fit random hyperparameter search model
rs_log_reg.fit(X_train, y_train);

Fitting 5 folds for each of 20 candidates, totalling 100 fits

In [44]: rs_log_reg.best_params_
Out[44]: {'solver': 'liblinear', 'C': 0.23357214690901212}
```

Now we've tuned LogisticRegression using RandomizedSearchCV, we'll do the same for RandomForestClassifier.

```
Out[46]: 0.8852459016393442

In [49]: # Setup random seed
np.random.seed(42)

# Setup random hyperparameter search for RandomForestClassifier
rs_rf = RandomizedSearchCV(RandomForestClassifier(),
                           param_distributions=rf_grid,
                           cv=5,
                           n_iter=20,
                           verbose=True)

# Fit random hyperparameter search model
rs_rf.fit(X_train, y_train);

Fitting 5 folds for each of 20 candidates, totalling 100 fits

In [51]: # Find the best parameters
rs_rf.best_params_
Out[51]: {'n_estimators': 210,
          'min_samples_split': 4,
          'min_samples_leaf': 19,
          'max_depth': 3}

In [52]: # Evaluate the randomized search random forest model
rs_rf.score(X_test, y_test)
Out[52]: 0.8688524590163934
```

Tuning the hyperparameters for each model saw a slight performance boost in both the RandomForestClassifier and LogisticRegression.

This is akin to tuning the settings on your oven and getting it to cook your favourite dish just right.

But since LogisticRegression is pulling out in front, we'll try tuning it further with [GridSearchCV](#).

Tuning a model with GridSearchCV

The difference between RandomizedSearchCV and GridSearchCV is where RandomizedSearchCV searches over a grid of hyperparameters performing n_iter combinations, GridSearchCV will test every single possible combination.

In short:

- RandomizedSearchCV - tries n_iter combinations of hyperparameters and saves the best.
- GridSearchCV - tries every single combination of hyperparameters and saves the best.

Let's see it in action.

The screenshot shows a Jupyter Notebook interface with multiple tabs at the top: 'Create a New Repository', 'Heart Disease Prediction', 'zero-to-mastery-ml/end', 'OneDrive/Documents/ML...', 'how to convert ipynb to ...', and '+'. The main area displays a code cell titled 'Tuning a model with GridSearchCV'.

```
# Different LogisticRegression hyperparameters
log_reg_grid = {'C': np.logspace(-4, 4, 20),
                 "solver": ["liblinear"]}

# Setup grid hyperparameter search for LogisticRegression
gs_log_reg = GridSearchCV(LogisticRegression(),
                           param_grid=log_reg_grid,
                           cv=5,
                           verbose=True)

# Fit grid hyperparameter search model
gs_log_reg.fit(X_train, y_train);
Fitting 5 folds for each of 20 candidates, totalling 100 fits

In [55]: # Check the best parameters
gs_log_reg.best_params_
Out[55]: {'C': 0.23357214690901212, 'solver': 'liblinear'}

In [56]: # Evaluate the model
gs_log_reg.score(X_test, y_test)
Out[56]: 0.8852459016393442
```

The notebook interface includes a toolbar with file operations like File, Edit, View, Insert, Cell, Kernel, Help, and a Trusted button. On the right, there's a sidebar with icons for GitHub, Google Drive, and other services, and a status bar showing '45%', 'ENG', and '2:25 PM'.

In this case, we get the same results as before since our grid only has a maximum of 20 different hyperparameter combinations.

Note: If there are a large amount of hyperparameters combinations in your grid, GridSearchCV may take a long time to try them all out. This is why it's a good idea to start with RandomizedSearchCV, try a certain amount of combinations and then use GridSearchCV to refine them.

Evaluating a classification model, beyond accuracy

Now we've got a tuned model, let's get some of the metrics we discussed before. We want:

- ROC curve and AUC score - `plot_roc_curve()`
 - Confusion matrix - `confusion_matrix()`
 - Classification report - `classification_report()`
 - Precision - `precision_score()`
 - Recall - `recall_score()`
 - F1-score - `f1_score()`

Luckily, Scikit-Learn has these all built-in

To access them, we'll have to use our model to make predictions on the test set. You can make predictions by calling `predict()` on a trained model and passing it the data you'd like to predict on. We'll make predictions on the test data.

Out[56]: 0.8852459016393442

Evaluating a classification model, beyond accuracy

```
In [57]: # Make predictions on test data  
y_preds = gs_log_reg.predict(X_test)
```

```
In [58]: y_preds
```

```
Out[58]: array([0, 1, 1, 0, 1, 1, 1, 0, 0, 1, 1, 0, 1, 0, 1, 1, 1, 0, 0, 0, 1, 0,  
    0, 1, 1, 1, 1, 0, 1, 0, 0, 0, 1, 0, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1,  
    1, 0, 1, 1, 0, 0, 0, 1, 1, 0, 0, 0, 1, 0, 0, 0], dtype=int64)
```

```
In [59]: y_test
```

```
Out[59]: array([0, 0, 1, 0, 1, 1, 0, 0, 1, 1, 1, 0, 1, 0, 1, 1, 1, 0, 0, 0, 1, 0,  
    0, 1, 1, 0, 0, 1, 0, 1, 1, 0, 0, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1,  
    1, 0, 0, 1, 0, 0, 0, 0, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0], dtype=int64)
```

```
In [ ]:
```

ROC Curve and AUC Scores

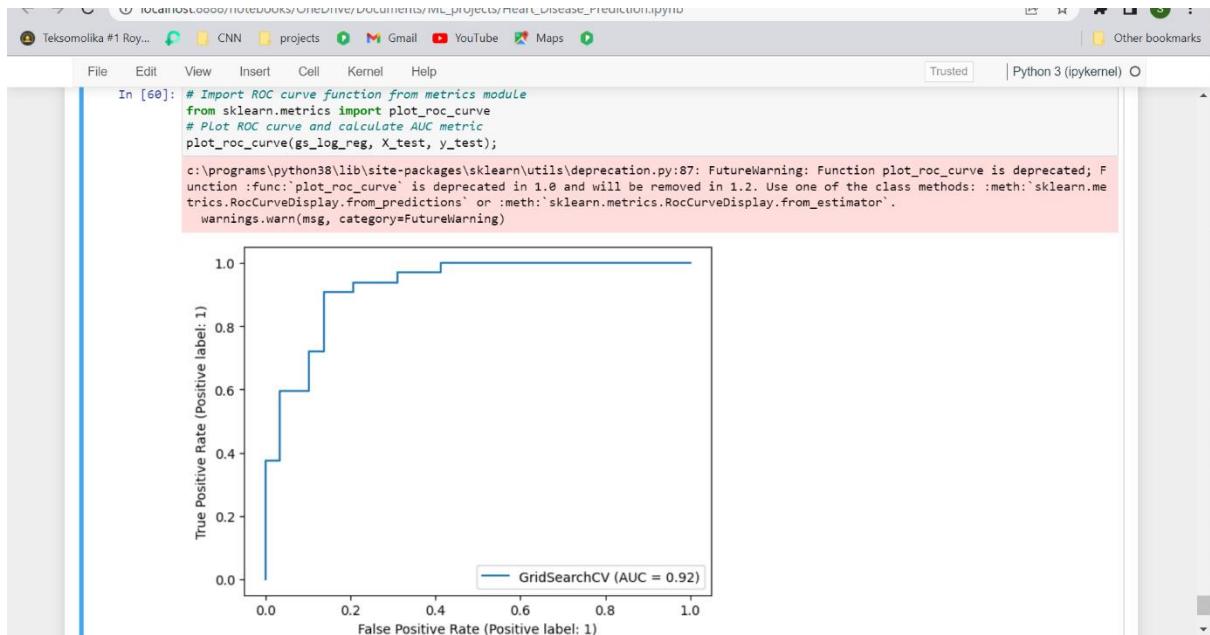
What's a ROC curve?

It's a way of understanding how your model is performing by comparing the true positive rate to the false positive rate. In our case...

To get an appropriate example in a real-world problem, consider a diagnostic test that seeks to determine whether a person has a certain disease. A false positive in this case occurs when the person tests positive, but does not actually have the disease. A false negative, on the other hand, occurs when the person tests negative, suggesting they are healthy, when they actually do have the disease.

Scikit-Learn implements a function `plot_roc_curve` which can help us create a ROC curve as well as calculate the area under the curve (AUC) metric. Reading the documentation on the `plot_roc_curve` function we can see it takes `(estimator, X, y)` as inputs. Where `estimator` is a fitted machine learning model and `X` and `y` are the data you'd like to test it on.

In our case, we'll use the GridSearchCV version of our LogisticRegression estimator, `gs_log_reg` as well as the test data, `X_test` and `y_test`.



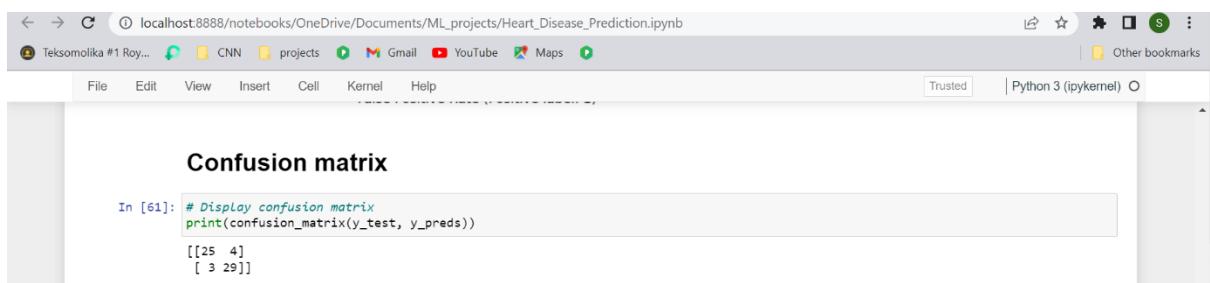
This is great, our model does far better than guessing which would be a line going from the bottom left corner to the top right corner, AUC = 0.5. But a perfect model would achieve an AUC score of 1.0, so there's still room for improvement.

Let's move onto the next evaluation request, a confusion matrix.

Confusion matrix

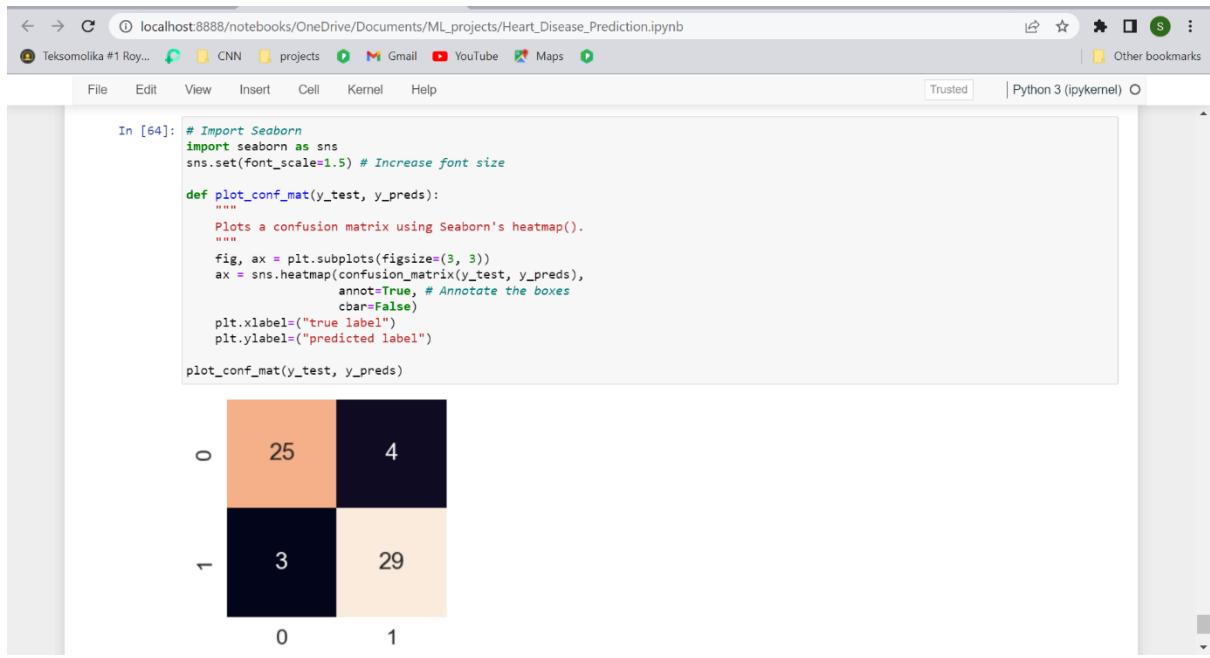
A confusion matrix is a visual way to show where your model made the right predictions and where it made the wrong predictions (or in other words, got confused).

Scikit-Learn allows us to create a confusion matrix using `confusion_matrix()` and passing it the true labels and predicted labels.



As you can see, Scikit-Learn's built-in confusion matrix is a bit bland. For a presentation you'd probably want to make it visual.

Let's create a function which uses Seaborn's `heatmap()` for doing so.



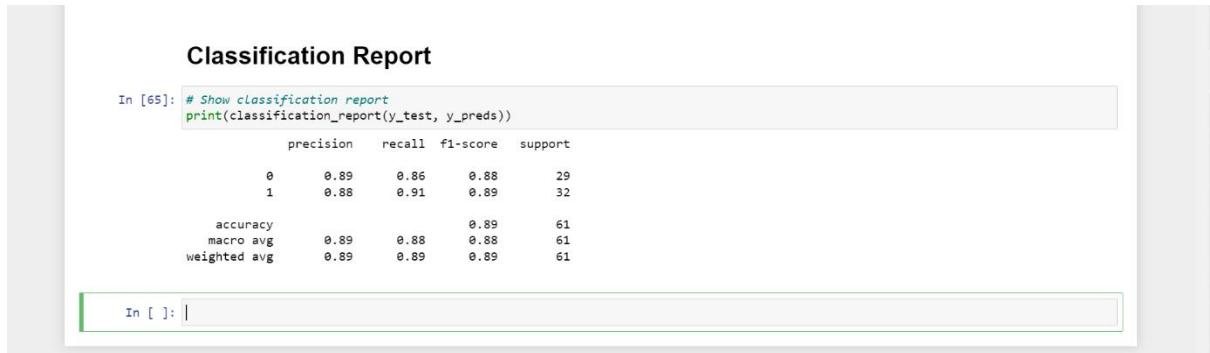
That looks much better.

You can see the model gets confused (predicts the wrong label) relatively the same across both classes. In essence, there are 4 occasions where the model predicted 0 when it should've been 1 (false negative) and 3 occasions where the model predicted 1 instead of 0 (false positive).

Classification report

We can make a classification report using `classification_report()` and passing it the true labels as well as our models predicted labels.

A classification report will also give us information of the precision and recall of our model for each class.



What's going on here?

Let's get a refresh.

- **Precision** - Indicates the proportion of positive identifications (model predicted class 1) which were actually correct. A model which produces no false positives has a precision of 1.0.

- **Recall** - Indicates the proportion of actual positives which were correctly classified. A model which produces no false negatives has a recall of 1.0.
- **F1 score** - A combination of precision and recall. A perfect model achieves an F1 score of 1.0.
- **Support** - The number of samples each metric was calculated on.
- **Accuracy** - The accuracy of the model in decimal form. Perfect accuracy is equal to 1.0.
- **Macro avg** - Short for macro average, the average precision, recall and F1 score between classes. Macro avg doesn't class imbalance into effort, so if you do have class imbalances, pay attention to this metric.
- **Weighted avg** - Short for weighted average, the weighted average precision, recall and F1 score between classes. Weighted means each metric is calculated with respect to how many samples there are in each class. This metric will favour the majority class (e.g. will give a high value when one class out performs another due to having more samples).

Ok, now we've got a few deeper insights on our model. But these were all calculated using a single training and test set.

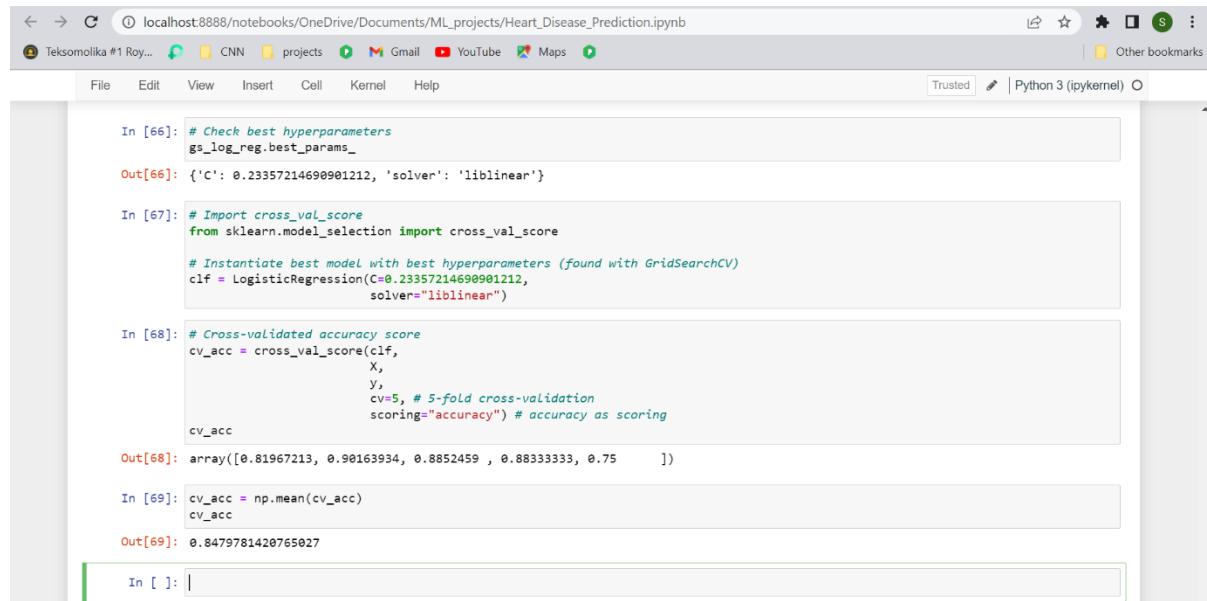
What we'll do to make them more solid is calculate them using cross-validation.

How?

We'll take the best model along with the best hyperparameters and use `cross_val_score()` along with various scoring parameter values.

`cross_val_score()` works by taking an estimator (machine learning model) along with data and labels. It then evaluates the machine learning model on the data and labels using cross-validation and a defined scoring parameter.

Let's remind ourselves of the best hyperparameters and then see them in action.



The screenshot shows a Jupyter Notebook interface with the following code and output:

```

In [66]: # Check best hyperparameters
gs_log_reg.best_params_
Out[66]: {'C': 0.23357214690901212, 'solver': 'liblinear'}
```

```

In [67]: # Import cross_val_score
from sklearn.model_selection import cross_val_score

# Instantiate best model with best hyperparameters (found with GridSearchCV)
clf = LogisticRegression(C=0.23357214690901212,
                         solver="liblinear")
```

```

In [68]: # Cross-validated accuracy score
cv_acc = cross_val_score(clf,
                        X,
                        y,
                        cv=5, # 5-fold cross-validation
                        scoring="accuracy") # accuracy as scoring
cv_acc
Out[68]: array([0.81967213, 0.90163934, 0.8852459 , 0.88333333, 0.75      ])
```

```

In [69]: cv_acc = np.mean(cv_acc)
cv_acc
Out[69]: 0.8479781420765027
```

Now we've got an instantiated classifier, let's find some cross-validated metrics.

```

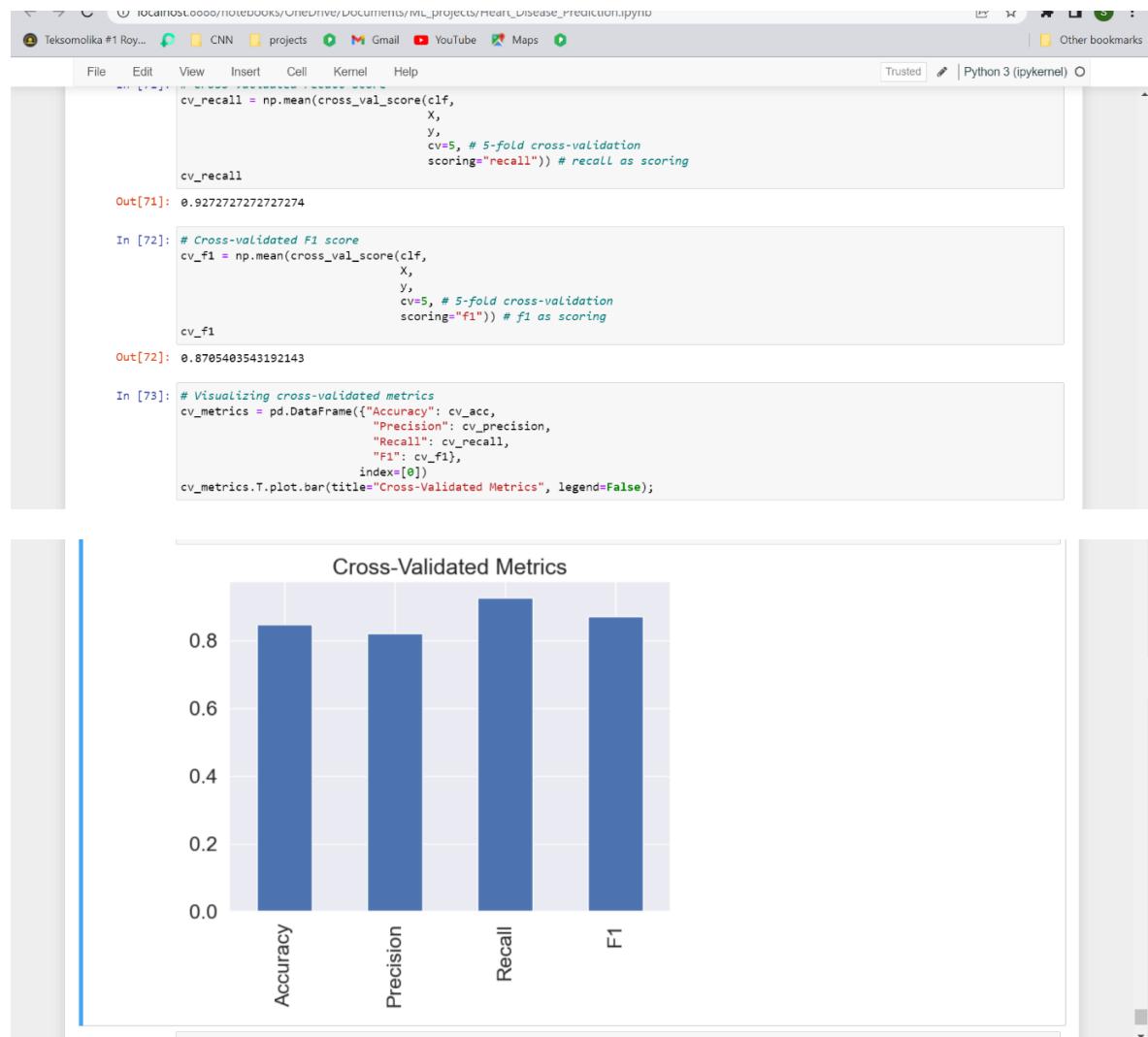
In [70]: # Cross-validated precision score
cv_precision = np.mean(cross_val_score(clf,
                                         X,
                                         y,
                                         cv=5, # 5-fold cross-validation
                                         scoring="precision")) # precision as scoring
cv_precision
Out[70]: 0.8215873015873015

In [71]: # Cross-validated recall score
cv_recall = np.mean(cross_val_score(clf,
                                         X,
                                         y,
                                         cv=5, # 5-fold cross-validation
                                         scoring="recall")) # recall as scoring
cv_recall
Out[71]: 0.9272727272727274

In [72]: # Cross-validated F1 score
cv_f1 = np.mean(cross_val_score(clf,
                                         X,
                                         y,
                                         cv=5, # 5-fold cross-validation
                                         scoring="f1")) # f1 as scoring
cv_f1
Out[72]: 0.8705403543192143

```

Okay, we've got cross validated metrics, now what? Let's visualize them.



This looks like something we could share. An extension might be adding the metrics on top of each bar so someone can quickly tell what they were.

What now?

The final thing to check off the list of our model evaluation techniques is feature importance.

Feature importance

Feature importance is another way of asking, "which features contributing most to the outcomes of the model?"

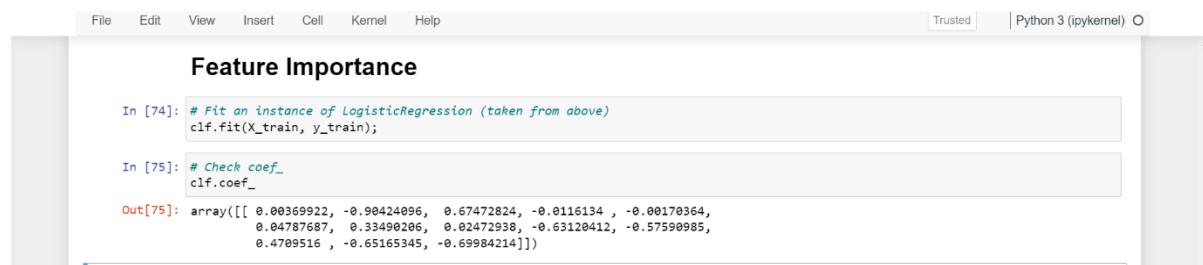
Or for our problem, trying to predict heart disease using a patient's medical characteristics, which characteristics contribute most to a model predicting whether someone has heart disease or not?

Unlike some of the other functions we've seen, because how each model finds patterns in data is slightly different, how a model judges how important those patterns are is different as well. This means for each model, there's a slightly different way of finding which features were most important.

You can usually find an example via the Scikit-Learn documentation or via searching for something like "[MODEL TYPE] feature importance", such as, "random forest feature importance".

Since we're using LogisticRegression, we'll look at one way we can calculate feature importance for it.

We can access the `coef_` attribute after we've fit an instance of LogisticRegression.



A screenshot of a Jupyter Notebook interface. The menu bar includes File, Edit, View, Insert, Cell, Kernel, and Help. The status bar shows 'Trusted' and 'Python 3 (ipykernel)'. The main area has a title 'Feature Importance'. Below it, two code cells are shown:

```
In [74]: # Fit an instance of LogisticRegression (taken from above)
clf.fit(X_train, y_train);

In [75]: # Check coef_
clf.coef_
```

The output cell (Out[75]) displays the coefficient array:

```
Out[75]: array([[ 0.00369922, -0.90424096,  0.67472824, -0.0116134 , -0.00170364,
   0.04787687,  0.33490206,  0.02472938, -0.63120412, -0.57590985,
   0.4709516 , -0.65165345, -0.69984214]])
```

Looking at this it might not make much sense. But these values are how much each feature contributes to how a model makes a decision on whether patterns in a sample of patients health data leans more towards having heart disease or not.

Even knowing this, in its current form, this `coef_` array still doesn't mean much. But it will if we combine it with the columns (features) of our dataframe.

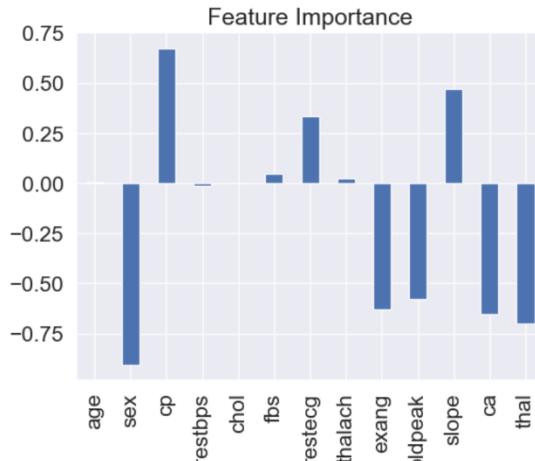
```
In [76]: # Match features to columns
features_dict = dict(zip(df.columns, list(clf.coef_[0])))
features_dict
```

```
Out[76]: {'age': 0.00369922271969309,
'sex': -0.9042409571824147,
'cp': 0.6747282417003583,
'trestbps': -0.01161339887501955,
'chol': -0.0017036434876667307,
'fbs': 0.04787687234653053,
'restecg': 0.3349020564037549,
'thalach': 0.024729381484044546,
'exang': -0.6312041160762268,
'oldpeak': -0.575909845192967,
'slope': 0.4709515982655588,
'ca': -0.6516534466983442,
'thal': -0.6998421437724929}
```

```
In [ ]:
```

File Edit View Insert Cell Kernel Help Trusted | Python 3 (ipykernel) O

```
In [77]: # Visualize feature importance
features_df = pd.DataFrame(features_dict, index=[0])
features_df.T.plot.bar(title="Feature Importance", legend=False);
```



You'll notice some are negative and some are positive.

The larger the value (bigger bar), the more the feature contributes to the models decision.

If the value is negative, it means there's a negative correlation. And vice versa for positive values.

For example, the sex attribute has a negative value of -0.904, which means as the value for sex increases, the target value decreases.

We can see this by comparing the sex column to the target column.

```
In [78]: pd.crosstab(df["sex"], df["target"])
Out[78]:
target    0   1
sex
  0    24  72
  1   114  93
```

```
In [ ]:
```

Looking back the data dictionary, we see slope is the "slope of the peak exercise ST segment" where:

- 0: Upsloping: better heart rate with exercise (uncommon)
- 1: Flatsloping: minimal change (typical healthy heart)
- 2: Downsloping: signs of unhealthy heart

According to the model, there's a positive correlation of 0.470, not as strong as sex and target but still more than 0.

This positive correlation means our model is picking up the pattern that as slope increases, so does the target value.

Is this true?

When you look at the contrast (`pd.crosstab(df["slope"], df["target"])`) it is. As slope goes up, so does target.

What can you do with this information?

This is something you might want to talk to a subject matter expert about. They may be interested in seeing where machine learning model is finding the most patterns (highest correlation) as well as where it's not (lowest correlation).

Doing this has a few benefits:

1. **Finding out more** - If some of the correlations and feature importances are confusing, a subject matter expert may be able to shed some light on the situation and help you figure out more.
2. **Redirecting efforts** - If some features offer far more value than others, this may change how you collect data for different problems. See point 3.
3. **Less but better** - Similar to above, if some features are offering far more value than others, you could reduce the number of features your model tries to find patterns in as well as improve the ones which offer the most. This could potentially lead to saving on computation, by having a model find patterns across less features, whilst still achieving the same performance levels.

6. Experimentation

If we can reach 95% accuracy at predicting whether or not a patient has heart disease during the proof of concept, we'll pursue this project.

In this case, we didn't. The highest accuracy our model achieved was below 90%.