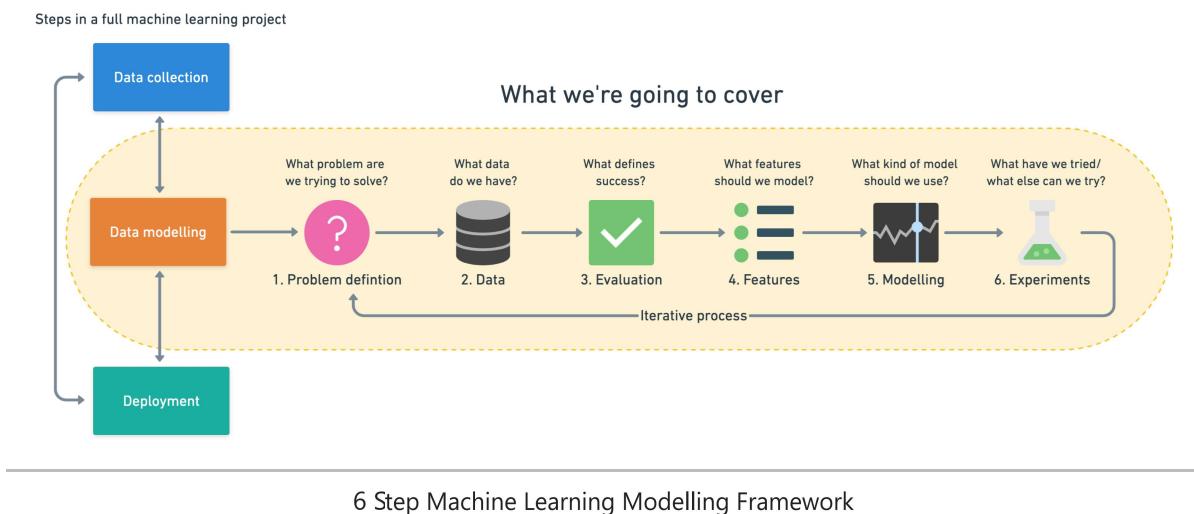


Predicting Heart Disease using Machine Learning

What we'll end up with

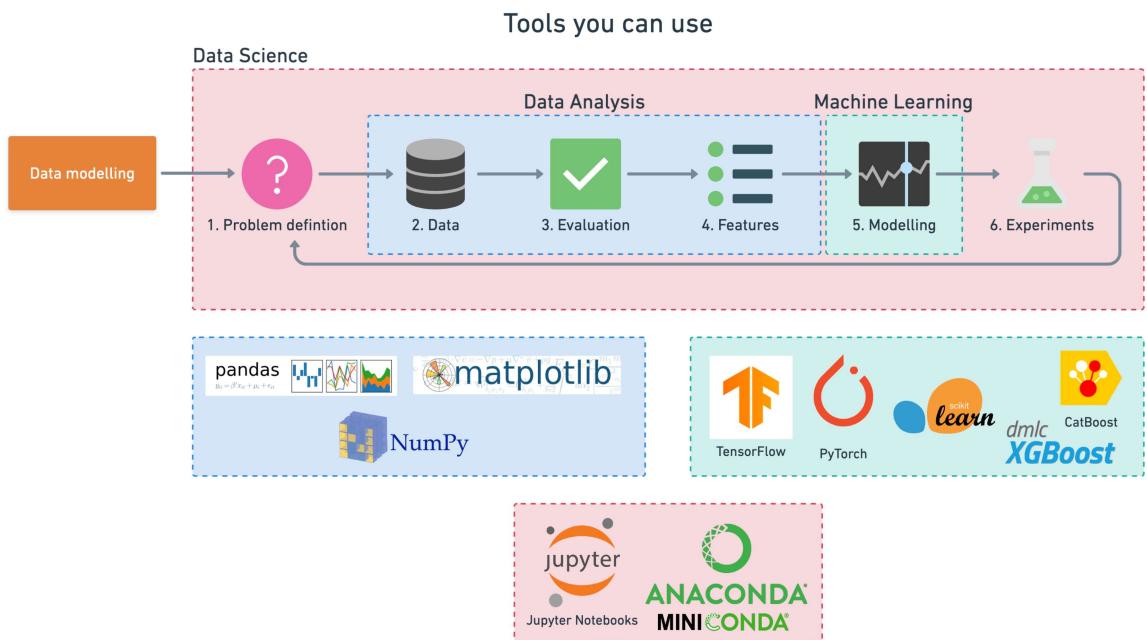
Since we already have a dataset, we'll approach the problem with the following machine learning modelling framework.



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.

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

1. Problem Definition

Here we'll be exploring **binary classification** as we will use number of features to predict whether a person have heart disease or not.

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**.

Evaluation

Goal: 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.

4. Features

Features are different parts of the data. During this step, we'll want to start finding out what we can do about the data. One of the most common ways to do this, is to create a **data dictionary**.

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

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 esophageal 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 cholestorol 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
 - can range from mild symptoms to severe problems
 - signals non-normal heart beat
 - 2: Possible or definite left ventricular hypertrophy
 - Enlarged heart's main pumping chamber
8. thalach - maximum heart rate achieved
9. exang - exercise induced angina(chest pain caused by reduced blood flow to the heart muscles): (1 = yes; 0 = no)
10. oldpeak - ST depression induced by exercise relative to rest
 - looks at stress of heart during excercise
 - unhealthy heart will stress more
11. slope - the slope of the peak exercise ST segment
 - 0: Upsloping: better heart rate with excercise (uncommon)
 - 1: Flatsloping: minimal change (typical healthy heart)
 - 2: Downsloping: 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 & Data

The libraries we use:

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

```
In [ ]: #importing
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline
import seaborn as sns

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

## Model evaluators
from sklearn.model_selection import train_test_split, cross_val_score, GridSearchCV,
from sklearn.metrics import confusion_matrix, precision_score, recall_score, confusi
from sklearn.metrics import plot_roc_curve, classification_report, f1_score
```

```
In [ ]: df = pd.read_csv('./data/heart-disease.csv')
df.shape
```

```
Out[ ]: (303, 14)
```

Data Exploration (exploratory data analysis or EDA)

short check list 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?

```
In [ ]: df.head()
```

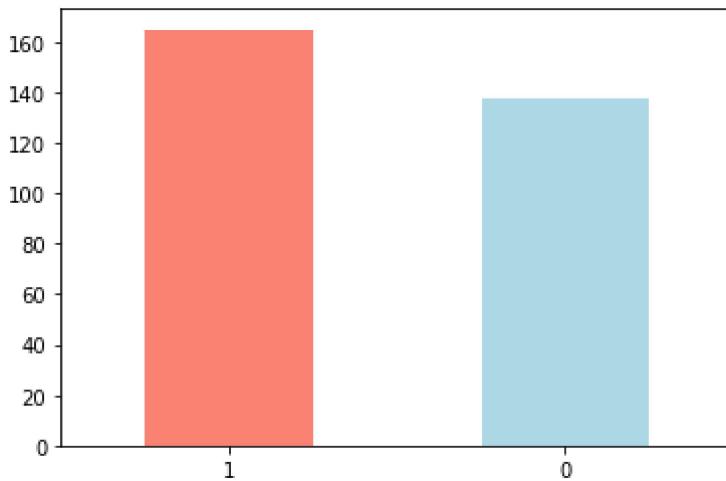
```
Out[ ]:   age  sex  cp  trestbps  chol  fbs  restecg  thalach  exang  oldpeak  slope  ca  thal  target
0    63     1    3      145   233     1      0     150      0     2.3      0     0     1     1
1    37     1    2      130   250     0      1     187      0     3.5      0     0     2     1
2    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 [ ]: df['target'].value_counts(normalize= True)
```

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

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

```
In [ ]: df['target'].value_counts().plot(kind='bar',color=["salmon", "lightblue"]);  
plt.xticks(rotation=0);
```



```
In [ ]: df.info()
```

```
<class 'pandas.core.frame.DataFrame'>  
RangeIndex: 303 entries, 0 to 302  
Data columns (total 14 columns):  
 #   Column      Non-Null Count  Dtype     
---  --          -----          -----  
 0   age         303 non-null    int64    
 1   sex         303 non-null    int64    
 2   cp          303 non-null    int64    
 3   trestbps    303 non-null    int64    
 4   chol        303 non-null    int64    
 5   fbs         303 non-null    int64    
 6   restecg     303 non-null    int64    
 7   thalach     303 non-null    int64    
 8   exang       303 non-null    int64    
 9   oldpeak     303 non-null    float64  
 10  slope        303 non-null    int64    
 11  ca          303 non-null    int64    
 12  thal        303 non-null    int64    
 13  target       303 non-null    int64    
dtypes: float64(1), int64(13)  
memory usage: 33.3 KB
```

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

Heart Disease Frequency according to Gender

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.

```
In [ ]:
```

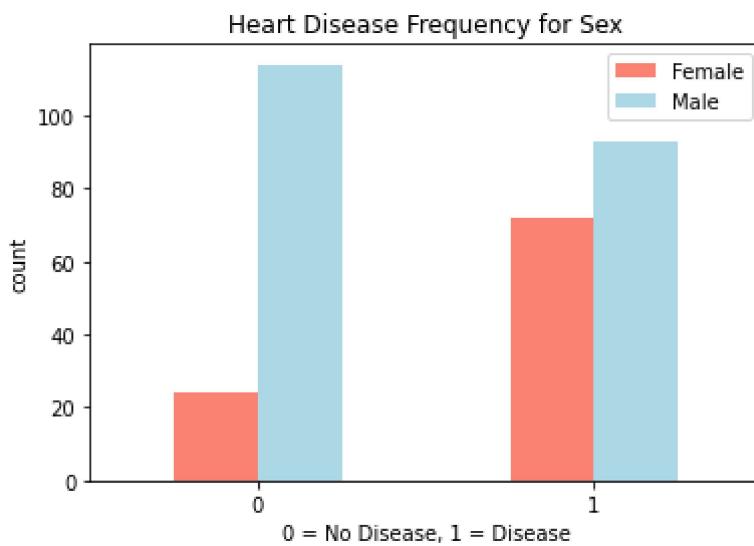
```
pd.DataFrame(df.sex.value_counts().values,columns=[ 'count'],index = [ 'male','female'])
```

```
Out[ ]:      count
male        207
female       96
```

```
In [ ]: pd.crosstab(df.target,df.sex)
```

```
Out[ ]:   sex  0    1
target
  0  24  114
  1  72  93
```

```
In [ ]: pd.crosstab(df.target,df.sex).plot(kind = 'bar',color=["salmon", "lightblue"])
# Add some attributes to it
plt.title("Heart Disease Frequency for Sex")
plt.xlabel("0 = No Disease, 1 = Disease")
plt.ylabel("count")
plt.legend(["Female", "Male"])
plt.xticks(rotation=0); # keep the Labels on the x-axis vertical
```



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.

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 []:

```
plt.figure(figsize=(10,6))
plt.scatter(df.age[df.target==1],df.thalach[df.target==1],color = 'salmon')
plt.scatter(df.age[df.target==0],df.thalach[df.target==0],color = 'lightblue')
plt.title('Heart Disease in function of Age and Max Heart Rate')
plt.xlabel('Age')
plt.ylabel('Max Heart Rate')
plt.legend(["Disease", "No Disease"]);
```



What can we infer from this?

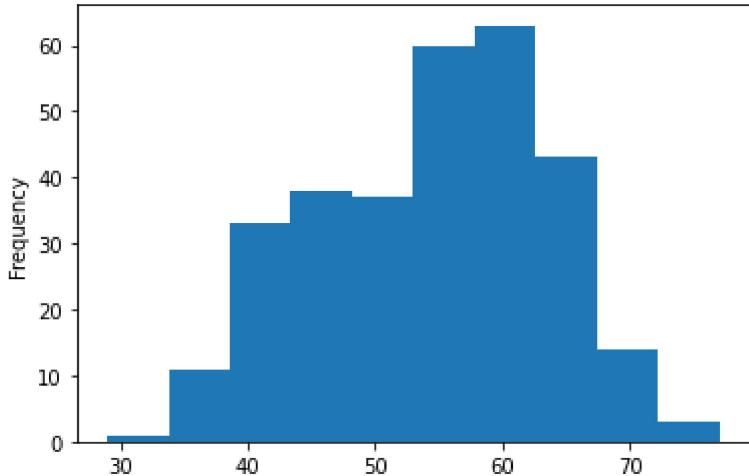
It seems the younger someone is, the higher their max heart rate (Red dots are higher on the left of the graph) and the older someone is, the more Blue dots there are. But this may be because there are more dots all together on the right side of the graph (older participants).

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**.

In []:

```
df.age.plot.hist();
```



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

Heart Disease Frequency per Chest Pain Type

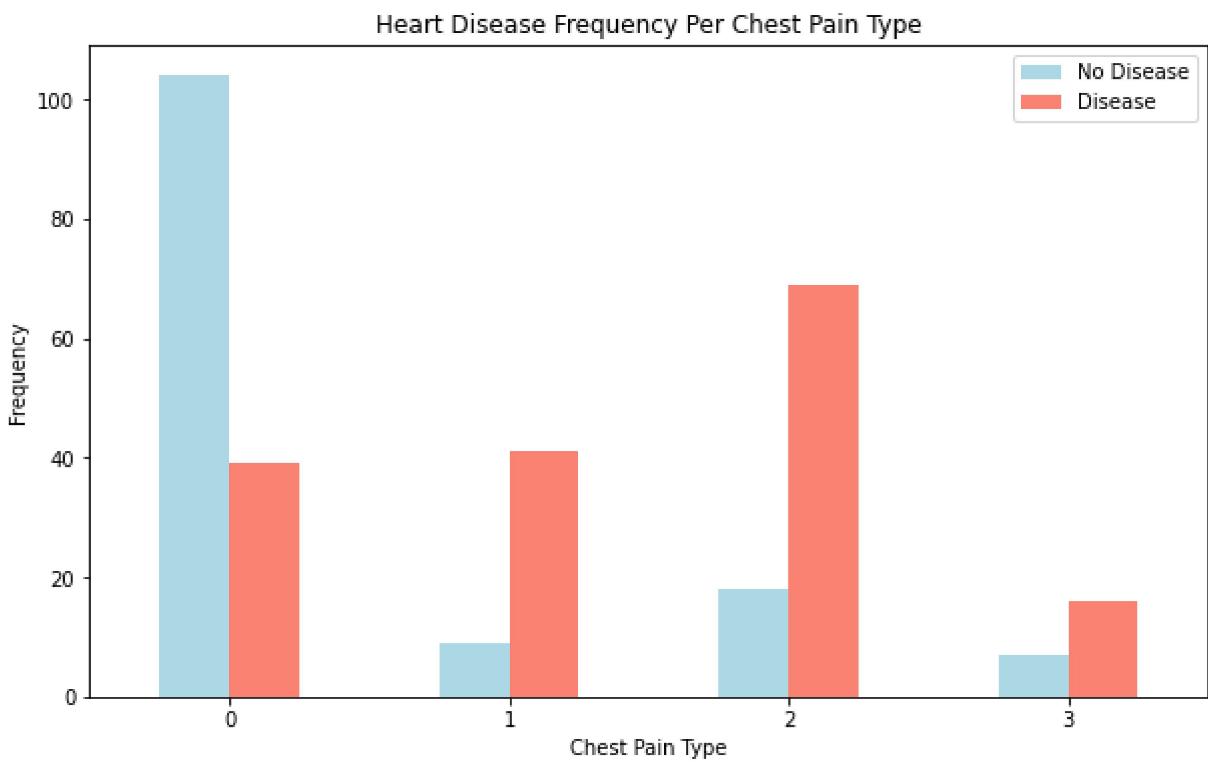
Let's try another independent variable. This time, `cp` (chest pain).

```
In [ ]: pd.crosstab(df.cp, df.target)
```

```
Out[ ]: target    0    1
cp
-----
0    104   39
1     9   41
2    18   69
3     7   16
```

```
In [ ]: 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?

The different levels of chest pain are:

- 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 esophageal spasms (non heart related)
 - 3: Asymptomatic: chest pain not showing signs of disease

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

What does atypical agina even mean?

- According to PubMed, it seems [even some medical professionals are confused by the term.](#)

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

We'll compare all of the independent variables in one hit.

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

In []:

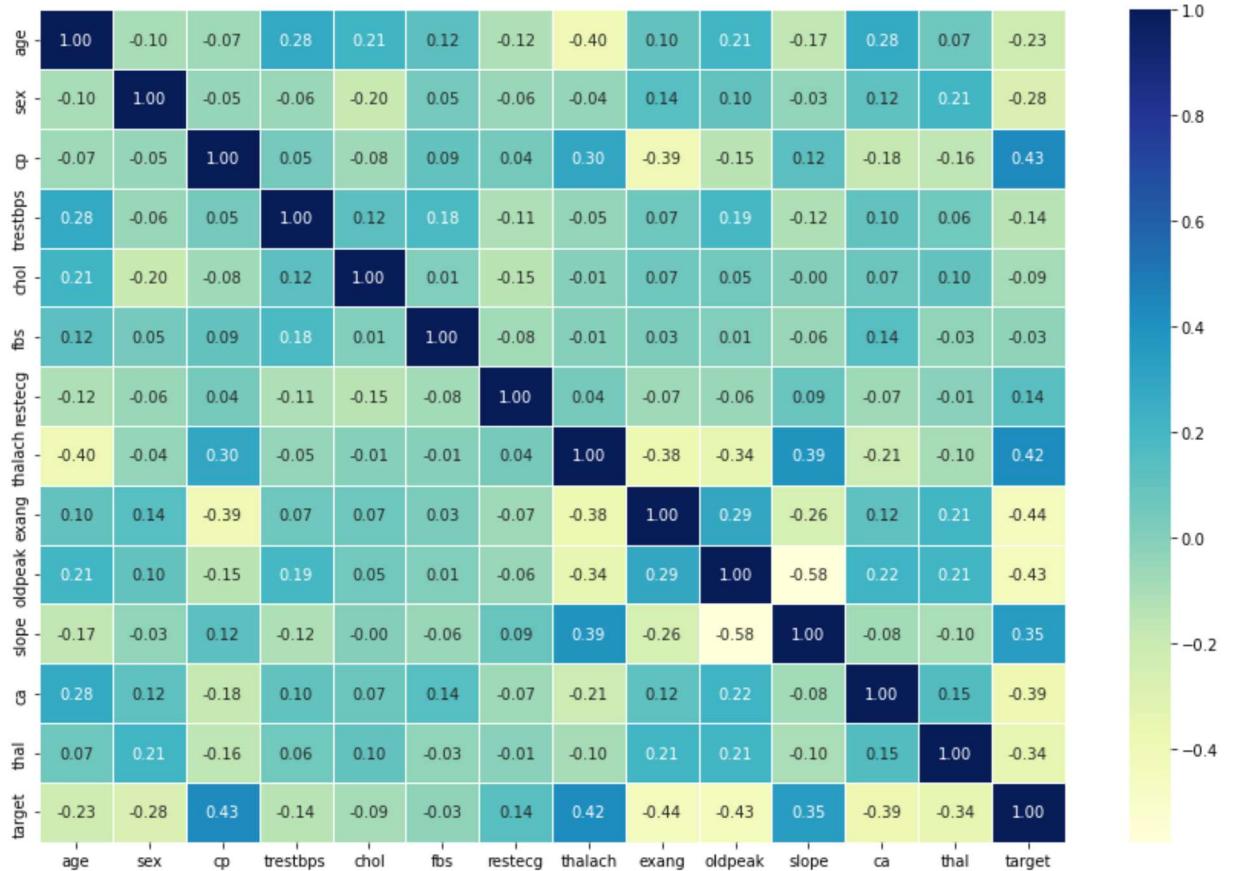
```
cor_matrix = df.corr()
cor_matrix
```

Out[]:

	age	sex	cp	trestbps	chol	fb	restecg	thalach	e
age	1.000000	-0.098447	-0.068653	0.279351	0.213678	0.121308	-0.116211	-0.398522	0.09
sex	-0.098447	1.000000	-0.049353	-0.056769	-0.197912	0.045032	-0.058196	-0.044020	0.14
cp	-0.068653	-0.049353	1.000000	0.047608	-0.076904	0.094444	0.044421	0.295762	-0.39
trestbps	0.279351	-0.056769	0.047608	1.000000	0.123174	0.177531	-0.114103	-0.046698	0.06
chol	0.213678	-0.197912	-0.076904	0.123174	1.000000	0.013294	-0.151040	-0.009940	0.06
fb	0.121308	0.045032	0.094444	0.177531	0.013294	1.000000	-0.084189	-0.008567	0.02
restecg	-0.116211	-0.058196	0.044421	-0.114103	-0.151040	-0.084189	1.000000	0.044123	-0.07
thalach	-0.398522	-0.044020	0.295762	-0.046698	-0.009940	-0.008567	0.044123	1.000000	-0.37
exang	0.096801	0.141664	-0.394280	0.067616	0.067023	0.025665	-0.070733	-0.378812	1.00
oldpeak	0.210013	0.096093	-0.149230	0.193216	0.053952	0.005747	-0.058770	-0.344187	0.28
slope	-0.168814	-0.030711	0.119717	-0.121475	-0.004038	-0.059894	0.093045	0.386784	-0.25
ca	0.276326	0.118261	-0.181053	0.101389	0.070511	0.137979	-0.072042	-0.213177	0.11
thal	0.068001	0.210041	-0.161736	0.062210	0.098803	-0.032019	-0.011981	-0.096439	0.20
target	-0.225439	-0.280937	0.433798	-0.144931	-0.085239	-0.028046	0.137230	0.421741	-0.43

In []:

```
plt.figure(figsize=(15,10))
sns.heatmap(cor_matrix,
            annot=True,
            linewidth = 0.5,
            fmt = '.2f',
            cmap = "YlGnBu"
            );
```



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

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.

```
In [ ]:
X = df.drop('target', axis=1)
y = df.target.values
```

Training and test split

```
In [ ]:
np.random.seed(42)

X_train, X_test, y_train, y_test = train_test_split(X,y,test_size=0.2)
```

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()`
3. RandomForest - `RandomForestClassifier()`

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.

```
In [ ]:
models = {'KNN':KNeighborsClassifier(),
          'Logistic Regression': LogisticRegression(),
          'Random Forset': RandomForestClassifier()}

def fit_and_score (models,X_train,X_test,y_train,y_test):
    """
    Fits and evaluates given machine learning models and returns models scores in a
    Args:
        models ([dict]): [a dict of different Scikit-Learn machine learning models]
        X_train ([DataFrame]): [training data]
        X_test ([DataFrame]): [testing data]
        y_train ([numpy.ndarray]): [labels for training data]
        y_test ([numpy.ndarray]): [labels for testing data]
    """
    np.random.seed(42)
    model_scores = {}
    for name,model in models.items():
        model.fit(X_train,y_train)
        model_scores[name] = model.score(X_test, y_test)

    return model_scores
```

```
In [ ]:
model_scores = fit_and_score(models=models,
                             X_train=X_train,
                             X_test=X_test,
                             y_train=y_train,
                             y_test=y_test)
```

```
In [ ]:
model_scores
```

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

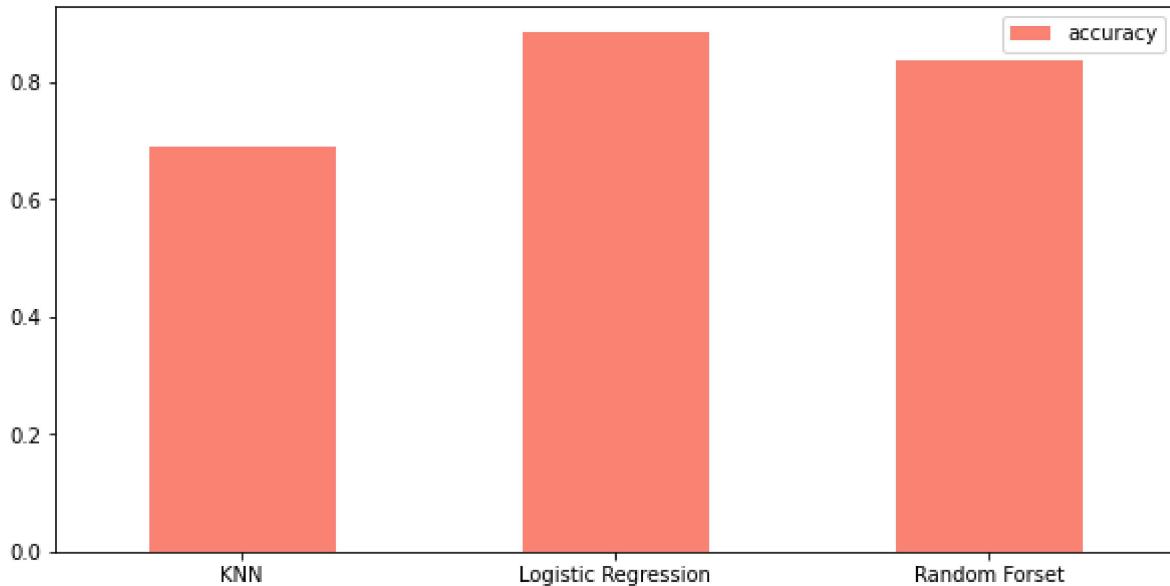
Model Comparison

Since we've saved our models scores to a dictionary, we can plot them by first converting them to a DataFrame.

```
In [ ]:
model_compare = pd.DataFrame(model_scores, index=['accuracy'])
model_compare
```

```
Out[ ]:
      KNN  Logistic Regression  Random Forset
accuracy  0.688525           0.885246       0.836066
```

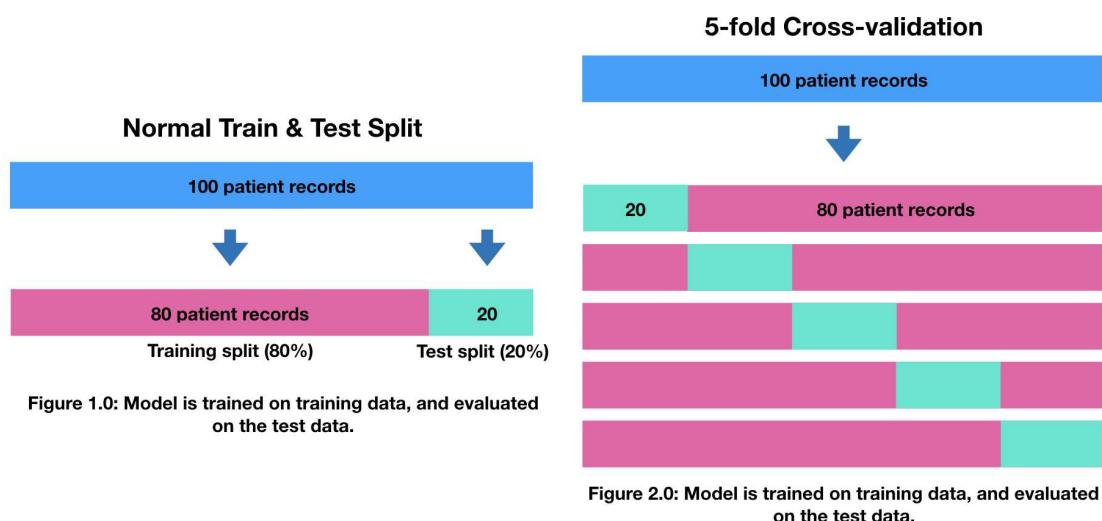
```
In [ ]:
model_compare.T.plot.bar(color='salmon',
                           figsize = (10,5))
plt.xticks(rotation = 0);
```



Hyperparameter tuning and cross-validation

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

- **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.

1. Tune KNeighborsClassifier (K-Nearest Neighbors or KNN) by hand

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`).

For now, let's try a few different values of `n_neighbors` .

```
In [ ]: train_scores=[]
test_scores=[]

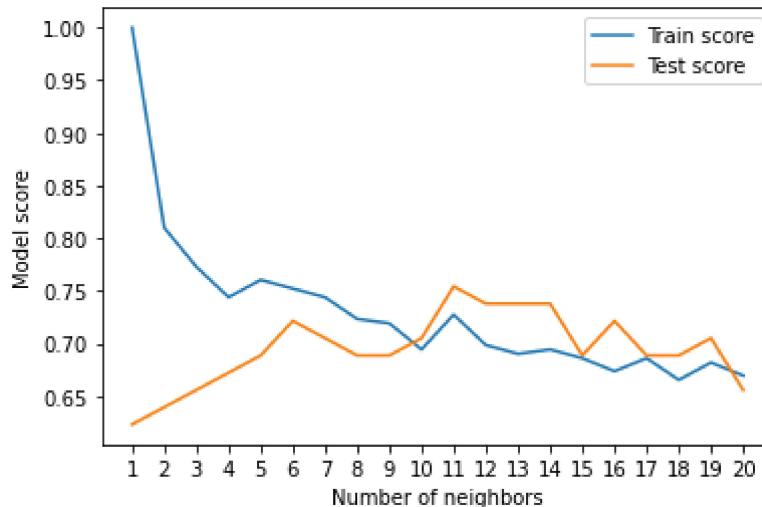
neighbors = range(1,21)
knn = KNeighborsClassifier()
for i in neighbors:
    knn.set_params(n_neighbors = i)
    knn.fit(X_train,y_train)
    train_scores.append(knn.score(X_train, y_train))
    test_scores.append(knn.score(X_test, y_test))
```

```
In [ ]: train_scores
```

```
Out[ ]: [1.0,
0.8099173553719008,
```

```
0.7727272727272727,  
0.743801652892562,  
0.7603305785123967,  
0.7520661157024794,  
0.743801652892562,  
0.7231404958677686,  
0.71900826446281,  
0.6942148760330579,  
0.7272727272727273,  
0.6983471074380165,  
0.6900826446280992,  
0.6942148760330579,  
0.6859504132231405,  
0.6735537190082644,  
0.6859504132231405,  
0.6652892561983471,  
0.6818181818181818,  
0.6694214876033058]
```

```
In [ ]: plt.plot(neighbors,train_scores, label="Train score")  
plt.plot(neighbors,test_scores, label="Test score")  
plt.xticks(np.arange(1, 21, 1))  
plt.xlabel("Number of neighbors")  
plt.ylabel("Model score")  
plt.legend();
```



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.

2. Tuning models with with `RandomizedSearchCV`

```
In [ ]: # Let's create a hyperparameter grid (a dictionary of different hyperparameters)  
# for each and then test them out.  
  
#different Logistic regression parameters:  
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)}
```

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 [ ]: np.random.seed(42)  
  
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 [ ]: rs_log_reg.best_params_  
  
Out[ ]: {'solver': 'liblinear', 'C': 0.23357214690901212}
```

```
In [ ]: rs_log_reg.score(X_test, y_test)  
  
Out[ ]: 0.8852459016393442
```

We'll do the same for `RandomForestClassifier`.

```
In [ ]: np.random.seed(42)  
  
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 [ ]: rs_rf.best_params_  
  
Out[ ]: {'n_estimators': 210,  
        'min_samples_split': 4,  
        'min_samples_leaf': 19,  
        'max_depth': 3}  
  
In [ ]: rs_rf.score(X_test, y_test)  
  
Out[ ]: 0.8688524590163934
```

since `LogisticRegression` is pulling out in front, we'll try tuning it further with `GridSearchCV`.

3. 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.

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



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

We get the same results as before since our grid only has a maximum of 20 different hyperparameter combinations.

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()`

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

ROC Curve and AUC Scores

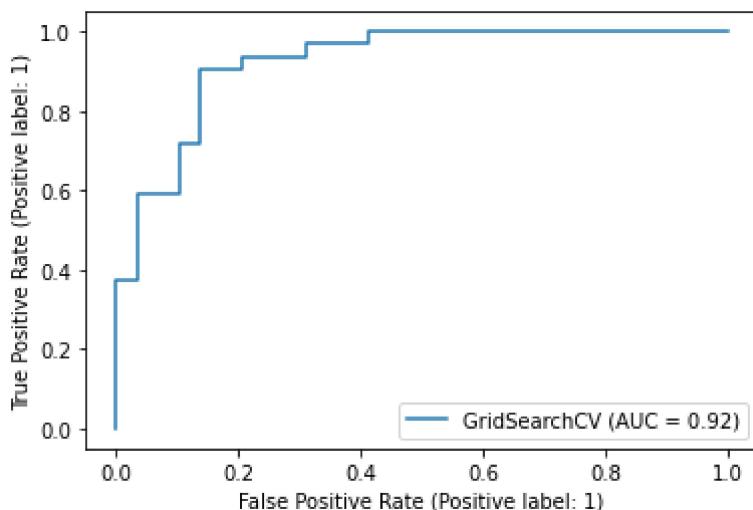
It's a way of understanding how your model is performing by comparing the true positive rate to the false positive rate.

Reading the documentation on the `RocCurveDisplay` 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`.

```
In [ ]: from sklearn.metrics import RocCurveDisplay
```

```
In [ ]: # Plot ROC curve and calculate AUC metric
RocCurveDisplay.from_estimator(gs_log_reg, X_test, y_test);
```



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.

```
In [ ]: print(confusion_matrix(y_test, y_preds))
```

```
[[25  4]
 [ 3 29]]
```

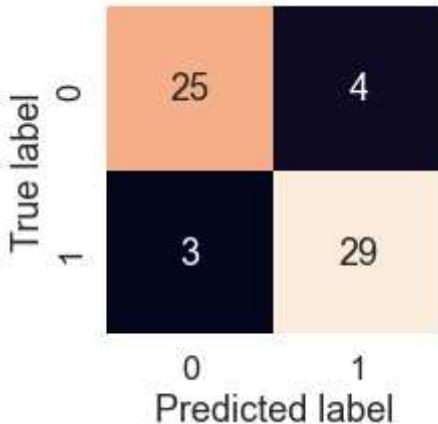
```
In [ ]: sns.set(font_scale=1.5) # Increase font size
```

```

def plot_conf_mat(y_test, y_preds):
    """
    Plots a confusion matrix using Seaborn's heatmap().
    """
    fig, ax = plt.subplots(figsize=(3, 3))
    ax = sns.heatmap(confusion_matrix(y_test, y_preds),
                      annot=True, # Annotate the boxes
                      cbar=False)
    plt.xlabel("Predicted label") # predictions go on the x-axis
    plt.ylabel("True label") # true labels go on the y-axis

plot_conf_mat(y_test, y_preds)

```



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.

```
In [ ]: print(classification_report(y_test, y_preds))
```

	precision	recall	f1-score	support
0	0.89	0.86	0.88	29
1	0.88	0.91	0.89	32
accuracy			0.89	61
macro avg	0.89	0.88	0.88	61
weighted avg	0.89	0.89	0.89	61

- **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).

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

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

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



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

Now we'll do the same for other classification metrics.

```
In [ ]: # 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[ ]: 0.8215873015873015
```



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

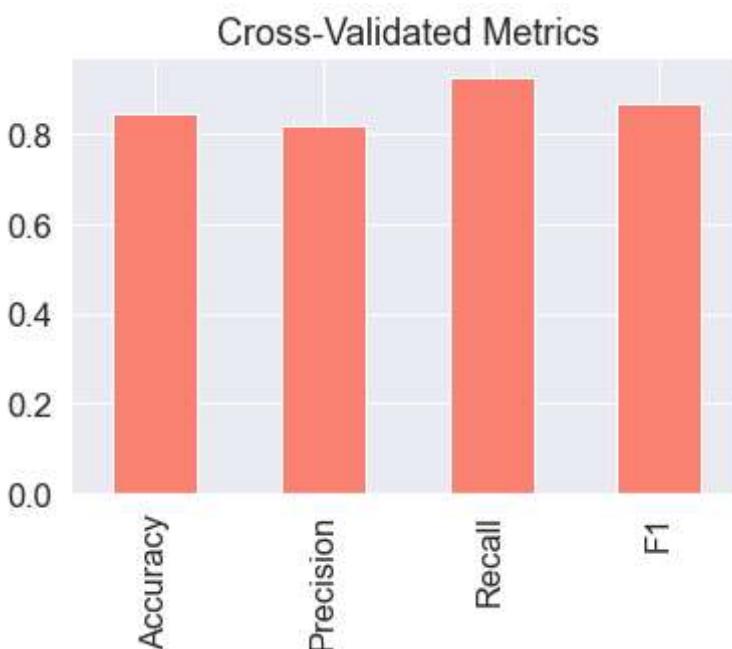
cv_recall
Out[ ]: 0.9272727272727274
```

```
In [ ]:  
# 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[ ]: 0.8705403543192143
```

Let's visualize them.

```
In [ ]:  
# Visualizing cross-validated metrics  
cv_metrics = pd.DataFrame({ "Accuracy": cv_acc,  
                            "Precision": cv_precision,  
                            "Recall": cv_recall,  
                            "F1": cv_f1},  
                           index=[0])  
cv_metrics.T.plot.bar(title="Cross-Validated Metrics",  
                      legend=False,  
                      color='salmon');
```



Feature importance

Feature importance is a 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?

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

To do so, we'll use the `coef_` attribute. Looking at the [Scikit-Learn documentation for LogisticRegression](#), the `coef_` attribute is the coefficient of the features in the decision function.

```
In [ ]: clf.fit(X_train,y_train);
```

```
In [ ]: clf.coef_
```

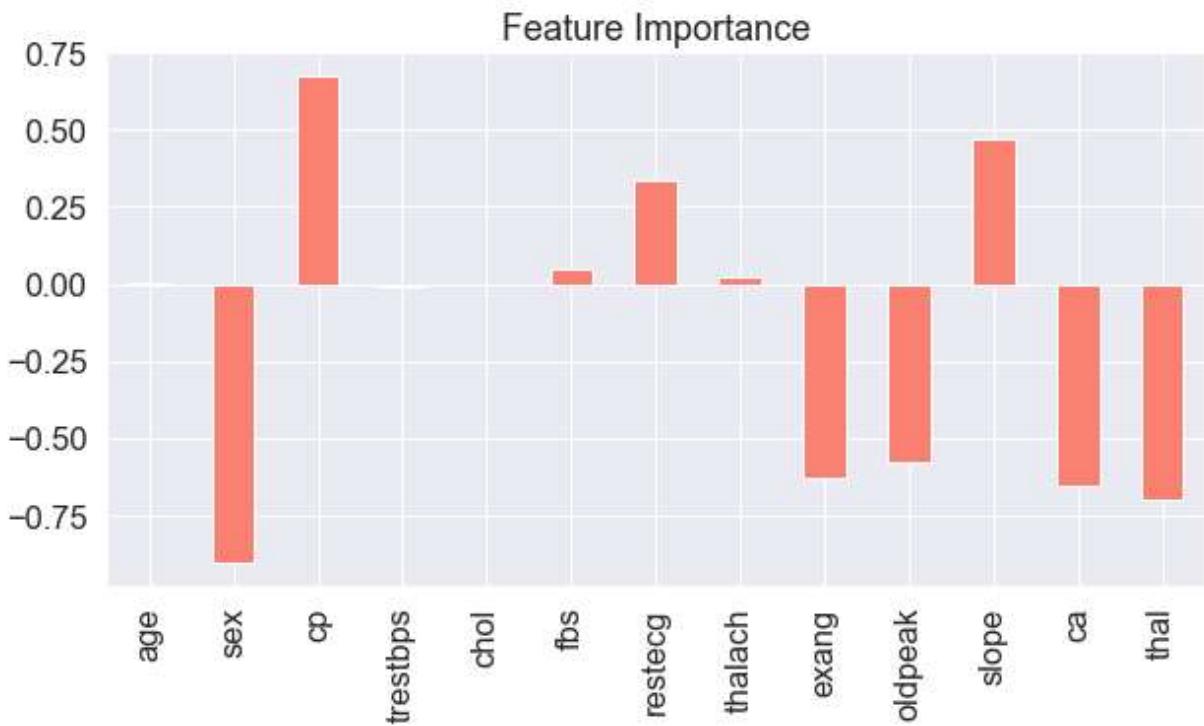
```
Out[ ]: array([[ 0.00369922, -0.90424087,  0.67472828, -0.0116134 , -0.00170364,
   0.04787689,  0.33490186,  0.02472938, -0.63120403, -0.57590919,
   0.4709512 , -0.6516535 , -0.69984202]])
```

```
In [ ]: feature_importance = pd.DataFrame(clf.coef_,columns=X.columns,index=[ 'feature importance'])
feature_importance
```

```
Out[ ]:      feature importance
```

age	0.003699
sex	-0.904241
cp	0.674728
trestbps	-0.011613
chol	-0.001704
fbs	0.047877
restecg	0.334902
thalach	0.024729
exang	-0.631204
oldpeak	-0.575909
slope	0.470951
ca	-0.651654
thal	-0.699842

```
In [ ]: # Visualize feature importance
feature_importance.plot.bar(figsize=(10,5),
                           title="Feature Importance",
                           legend=False,
                           color='salmon');
```



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.

```
In [ ]: # ignoring lower importance features
features = feature_importance[np.absolute(feature_importance['feature importance']) >
features
```

Out[]:

feature importance	
sex	-0.904241
cp	0.674728
restecg	0.334902
exang	-0.631204
oldpeak	-0.575909
slope	0.470951
ca	-0.651654
thal	-0.699842

```
In [ ]: features.plot.bar(figsize=(12,8),
                      title="Feature Importance",
                      legend=False,
                      color='salmon')
plt.xticks(rotation = 0);
```



Model persistence

```
In [ ]: import sys
```

```
In [ ]: mkdir saved_model
```

A subdirectory or file saved_model already exists.

```
In [ ]: import pickle
# Save an existing model to file
pickle.dump((clf), open("./saved_model/logistic_regression_model_1.pkl", "wb"))
```

Next steps

Observation and Notes:

we didn't reach the accuracy target (95%), so we know the current model we're using (a tuned version of `LogisticRegression`) along with our specific data set doesn't hit the target we set ourselves. Questions to ask:

- Could we collect more data?
- Could we try a better model? like: `CatBoost` or `XGBoost`.
- Could we improve the current models (beyond what we've done so far)?

Deploy

```
In [ ]: import pickle
model = pickle.load(open('./saved_model/logistic_regression_model_1.pkl','rb'))
```

```
In [ ]: # creating dictionary from dataframe using DictVectorizer
from sklearn.feature_extraction import DictVectorizer
dv = DictVectorizer(sparse=False)
X_vec = X.to_dict(orient='rows')
dv.fit(X_vec);
X_vec[1]
```

```
Out[ ]: {'age': 37,
'sex': 1,
'cp': 2,
'trestbps': 130,
'chol': 250,
'fbs': 0,
'restecg': 1,
'thalach': 187,
'exang': 0,
'oldpeak': 3.5,
'slope': 0,
'ca': 0,
'thal': 2}
```

```
In [ ]: def predict_single(patient, dv, model):
    X = dv.transform([patient])
    y_pred = model.predict_proba(X)[:, 1]
    return y_pred[0]
```

```
In [ ]: patient = X_vec[1]
patient
```

```
Out[ ]: {'age': 37,
'sex': 1,
'cp': 2,
'trestbps': 130,
'chol': 250,
'fbs': 0,
'restecg': 1,
'thalach': 187,
'exang': 0,
'oldpeak': 3.5,
'slope': 0,
'ca': 0,
'thal': 2}
```

```
In [ ]: #true value
df.target[1]
```

```
Out[ ]: 1
```

```
In [ ]: #pred value
predict_single(patient, dv, model)
```

```
Out[ ]: 0.9999919608702539
```

```
In [ ]: #saving the model and the DictVectorizer
pickle.dump((clf,dv), open("./saved_model/logistic_regression_model_2.pkl", "wb"))
```

using flask to request and post

we w'll use the `Heart_classification.py` file to create host then we w'll test it here

```
In [ ]: import requests
```

run `Heart_classification.py` then run this cell

```
In [ ]: url = 'http://localhost:9696/predict'  
response = requests.post(url, json=patient)  
result = response.json()
```

```
In [ ]: result
```

```
Out[ ]: {'heart_disease': False, 'heart_disease_probability': 3.6004906153090464e-19}
```