MACHINE LEARNING WITH k NEAREST NEIGHBOUR ALGORITHM

> by Dr Juan H Klopper

- Research Fellow
- School for Data Science and Computational Thinking
- Stellenbosch University

INTRODUCTION

The k nearest neighbour (kNN) algorithm is one of the easiest machine learning algorithms to understand and implement. It can be used for classification and regression problems. In the former, the target variable is categorical. In the latter, it is numerical.

In this notebook we explore the k nearest neighbour machine learning (ML) algorithm. We start of with a simple example of classification before embarking on solving a more realistic problem. Along the way we will learn a lot of the basic concepts of ML. We end with a small example to help us understand how to use kNN in a regression problem.

To note upfront, for the same objects, ML uses different names than we use in statistics. Independent variables are referred to as **feature variables** or simply **features**. The dependent variable is termed a **target variable** or an **outcome variable** (or simply a **target** or an **outcome**. If the target variable is categorical, then the sample space elements are termed **classes**.

PACKAGES USED IN THIS NOTEBOOK

The following packages will be used in this notebook.

```
import numpy as np
from pandas import DataFrame, Series, read_csv
from sklearn.datasets import make_classification
from sklearn.neighbors import KNeighborsClassifier, KNeighborsRegressor
from sklearn.model_selection import train_test_split, cross_val_score, GridSearc
from sklearn import metrics
from sklearn.preprocessing import StandardScaler
```

```
import plotly.graph_objects as go
import plotly.express as px
import plotly.io as pio
pio.templates.default = 'plotly_white'
```

```
import matplotlib.pyplot as plt
import seaborn as sns
```

```
%config InlineBackend.figure_format = "retina" # For Retina type displays
```

```
# Format tables printed to the screen (don't put this on the same line as the co
%load_ext google.colab.data_table
```

```
from google.colab import drive  # Connect to Google Drive
```

THE NEAREST NEIGHBOURS CONCEPT

k NEAREST NEIGHBOUR CLASSIFICATION

The k nearest neighbour classifier classifies an observation based on the classes in its vicinity. Vicinity infers distance. With this ML algorithm, we measure a distance between observations.

There are various ways to define distance. Euclidean distance (a straight line on a flat surface) is most familiar to us. We consider a single numerical variable (for a featuare variable) and two classes (for a binary target variable). The code below generates a pandas dataframe object, with two appropriately named variables.

```
np.random.seed(42)

df = DataFrame(
    {'Feature':np.random.randint(10, high=20, size=7),
     'Target':np.random.choice(['A', 'B'], size=7)}
)
df
```

1 to 7 of 7 entries Filter ?

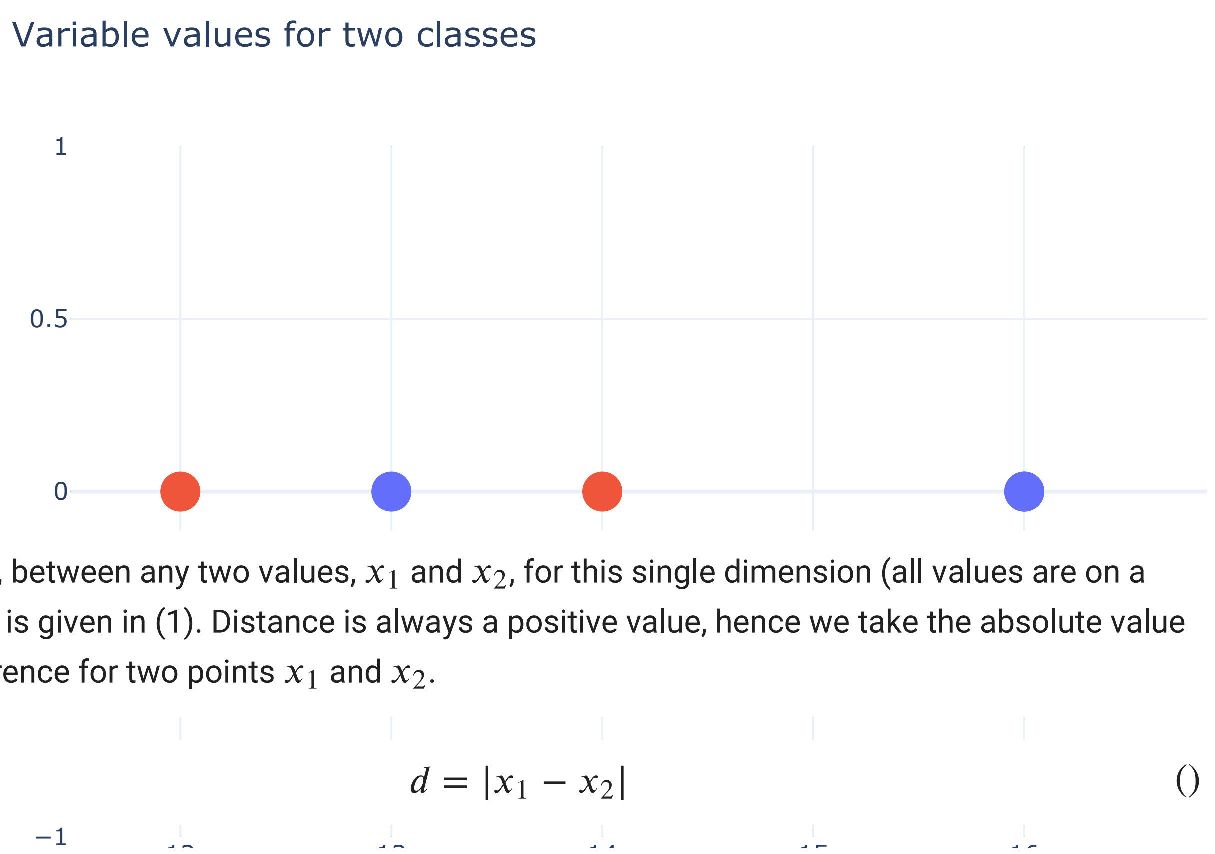
index	Feature	Target
0	16	A
1	13	A
2	17	A
3	14	B
4	16	A
5	19	B
6	12	B

With the random seed set as 42, we note four observations belonging to group A and three to group B.

A scatter plot can be used to visualise this single variable for two classes. Note that two observations in group A have a feature variable value of 16.

```
single_dim_fig = go.Figure(
    go.Scatter(
        x=df.loc[df.Target == 'A'].Feature,
        y=[0, 0, 0, 0],
        name='A',
        mode='markers',
        marker={'size':20}
    )
).add_trace(
    go.Scatter(
        x=df.loc[df.Target == 'B'].Feature,
        y=[0, 0, 0],
        name='B',
        mode='markers',
        marker={'size':20}
    )
).update_layout(
    title='Variable values for two classes',
    xaxis={'title':'Variable value'}
)

single_dim_fig
```



Distance, d, between any two values, x_1 and x_2, for this single dimension (all values are on a single axis) is given in (1). Distance is always a positive value, hence we take the absolute value of the difference for two points x_1 and x_2.

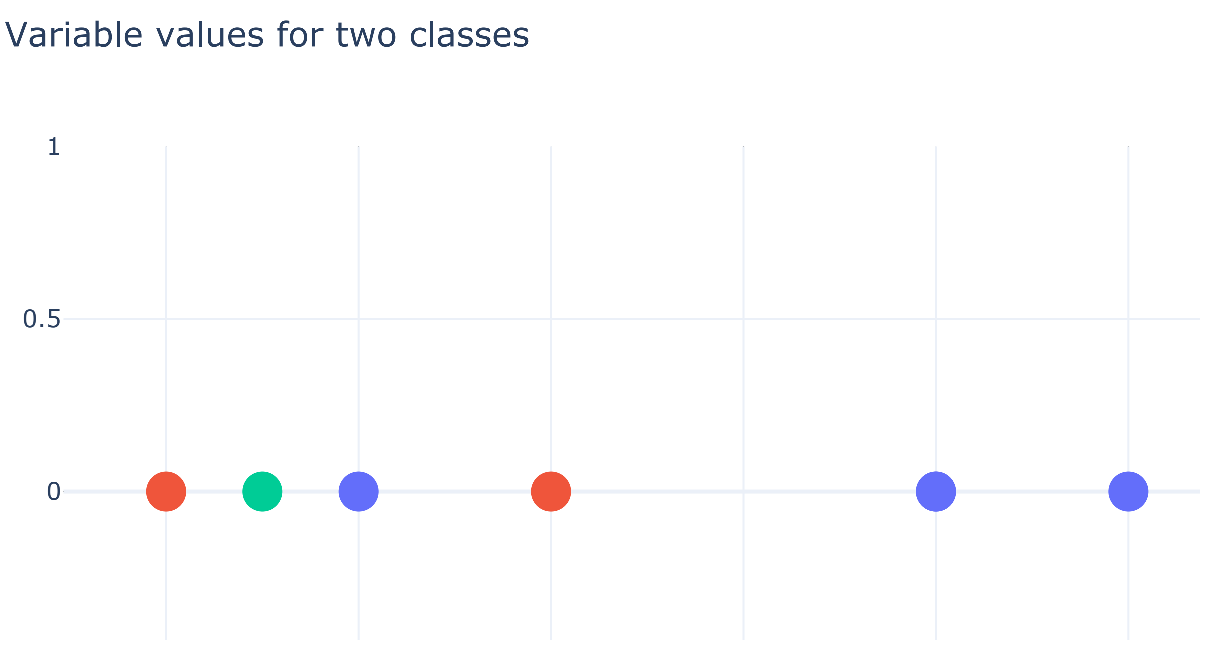
$$d = |x_1 - x_2| \qquad ()$$

The distance between 12 and 19 is therefor $|12 - 19| = |-7| = 7$.

Now we introduce an unknown observation with a value of 12.5. The k in k nearest neigbours is an integer (whole number) reflecting the number of neighbours to an observation. It is set at an odd value. In this instance, we shall say $k = 3$.

```
single_dim_fig.add_trace(
    go.Scatter(
        x=[12.5],
        y=[0],
        name='Unkown class',
        mode='markers',
        marker={'size':20}
    )
)

single_dim_fig
```



The three nearest neigbours are 12, 13, and 14. The distance to these nearest (closets) neighbours are $|12.5 - 12| = 0.5$, $|12.5 - 13| = 0.5$, and $|12.5 - 14| = 1.5$.

We note that the classes for these three nearest neigbours are group B, group B, and group A. Since we chose an odd number of neighbors, we can simply take a *majority vote*. This would be group B. The k nearest neighbour classifier would therefor classify this new observation as belonging to target class `B`.

If we add another numerical variable, we can plot the data as a scatter plot in the plane. We do this below after adding another random set of values.

```
np.random.seed(42)

df['Feature2'] = np.random.randint(100, 200, 7) / 10
```

```
df
```

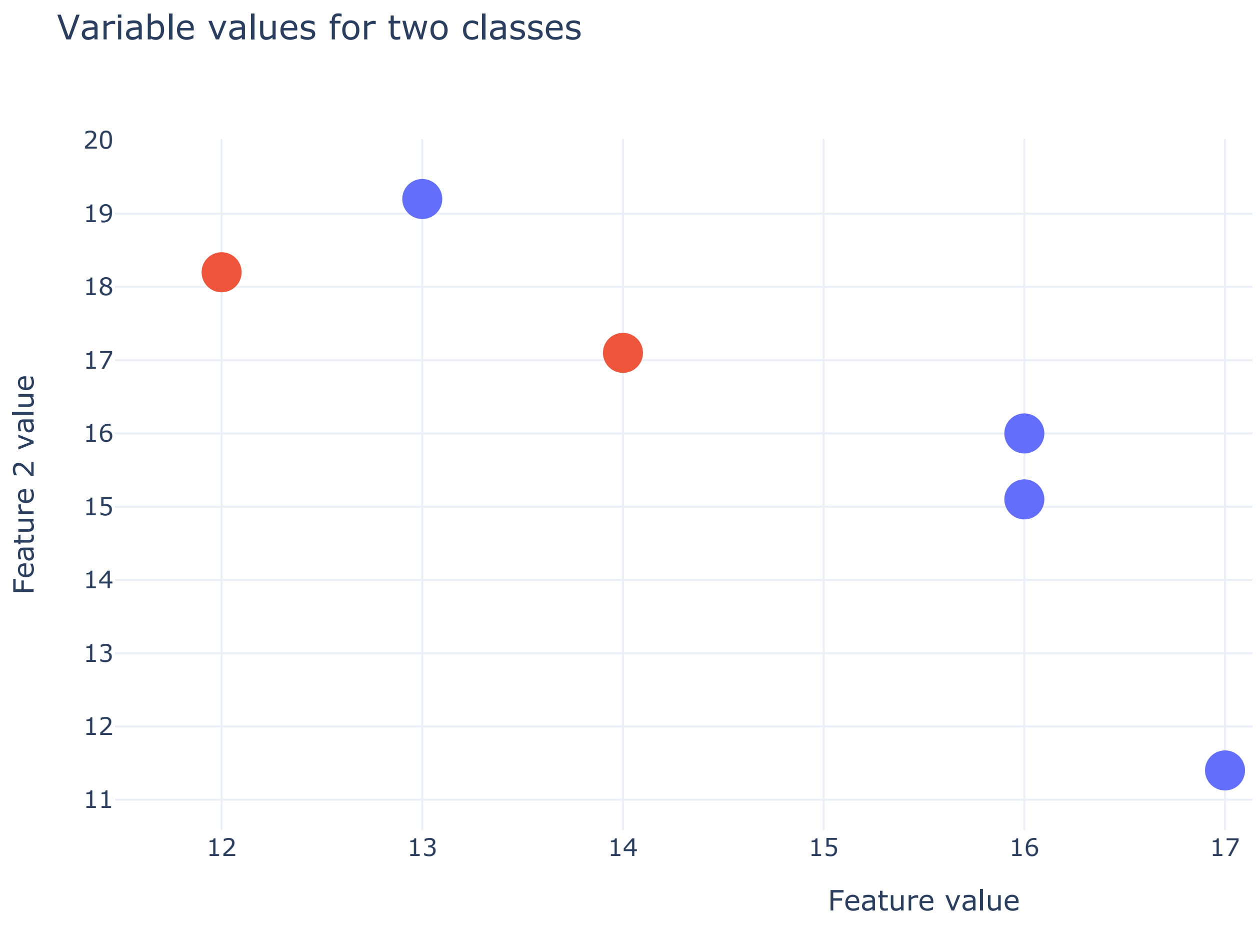
1 to 7 of 7 entries Filter ?

index	Feature	Target	Feature2
0	16	A	15.1
1	13	A	19.2
2	17	A	11.4
3	14	B	17.1
4	16	A	16.0
5	19	B	12.0
6	12	B	18.2

Show 25 per page

```
two_dim_fig = go.Figure(
    go.Scatter(
        x=df.loc[df.Target == 'A'].Feature,
        y=df.loc[df.Target == 'A'].Feature2,
        name='Group A',
        mode='markers',
        marker={'size':20}
    )
).add_trace(
    go.Scatter(
        x=df.loc[df.Target == 'B'].Feature,
        y=df.loc[df.Target == 'B'].Feature2,
        name='Group B',
        mode='markers',
        marker={'size':20}
    )
).update_layout(
    title='Variable values for two classes',
    xaxis={'title':'Feature value'},
    yaxis={'title':'Feature 2 value'}
)

two_dim_fig
```



The equation for the Euclidean distance in the plane (two dimensional space) is the Pythagorean Theorem and is shown in (2) for two points in the plane, $P_1 = (x_1, y_1)$ and

$P_2 = (x_2, y_2)$.

$$d\left(P_1, P_2\right) = \sqrt{\left(x_1 - x_2\right)^2 + \left(y_1 - y_2\right)^2} \tag{2}$$

We are only interested in the positive value of the square root. Since both expressions in the square root are squared, we will always have a value of greater than or equal to 0 and can therefor take the square root.

A new observation with values `Feature` $= 14$ and `Feature2` $= 18$ is shown below.

```
two_dim_fig.add_trace(
    go.Scatter(
        x=[14],
        y=[18],
        name='Unkown class',
        mode='markers',
        marker={'size':20}
    )
).update_yaxes(
    scaleanchor='x',
    scaleratio=1,
  ) # For same x and y axis scale

two_dim_fig
```

Variable values for two classes

The nearest $k = 3$ neigbours are group A, group B, and group B. This unknown observation is therefor classified as belonging to group B. Below, we set up a function to calculate this distance and use it for the three nearest neighbours.

```
def dist_2D(x1, y1, x2, y2):
  distance = np.sqrt((x1 - x2)**2 + (y1 - y2)**2)
  return distance
```

```
dist_2D(14, 18, 14, 17.1) # Group B observation
```

```
0.8999999999999986
```

```
dist_2D(14, 18, 13, 19.2) # Group A observation
```

```
1.5620499351813304
```

```
dist_2D(14, 18, 12, 18.2) # Group B observation
```

```
2.009975124224178
```

All other observations are *further* away.

The scikit-learn package has many ML algorithms including a k nearest neighbour classifier (for classification problems). We will use this classifier in an example. First, though, we will use the `make_classification` function from the datasets module of scikit-learn. This function generates random value datasets for ML tasks. The code comment explains the arguments used. The documentation for this function list all the other arguments, which we will leave at their default values.

```
X, y = make_classification(
    n_samples=200, # Number of observations
    n_features=5, # Number of features
    n_informative=3, # Number of features that are informative as to the class
    n_redundant=2, # Number of redundant feautres
    n_classes=2, # Setting a binary target variable
    flip_y=0.1, # Flip 10% of the observations to the other class
    random_state=42 # Seeding the pseudo-random number generator
)
```

The function returns two arrays, which we have assigned to the commonly used variable `X` for the set of feature variables and `y` for the target variable. It is worthwhile to look at the type of objects assigned to these variables and their dimensions.

```
type(X) # X is a numpy multi-dimensional array
```

```
numpy.ndarray
```

```
X.shape # Shape attribute shows 200 observations and 5 variables
```

```
(200, 5)
```

```
X.dtype # Values are 64-bit floating point values (decimals)
```

```
dtype('float64')
```

```
type(y) # y is also a multi-dimensional array
```

```
numpy.ndarray
```

```
y.shape # y contains 200 observations
```

```
(200,)
```

```
y.dtype # Two classes encoded as the 64 bit integers 0 and 1
```

```
dtype('int64')
```

We can use this random data to build a dataframe object.

```
df = DataFrame(
    X,
    columns=['Feature1', 'Feature2', 'Feature3', 'Feature4', 'Feature5']
)

df['Target'] = y # Adding target variable

df['TargetClass'] = Series(
    y,
    dtype='category'
) # Adding the target variable again, but specifying it to be categorical

df[:5] # First 5 rows
```

1 to 5 of 5 entries Filter ?

The `describe` method shows the summary statistics for the variables in the dataframe object.

```
df.loc[:, df.columns != 'Target'].describe() # Exclude the Target variable from
```

1 to 8 of 8 entries Filter ?

index	Feature1	Feature2	Feature3	Feature4
count	200.0	200.0	200.0	200.0
mean	-0.09833682324423682	-0.34922631298570556	0.5029790825544116	-0.3633490353368696
std	1.4086800247418874	1.247183864021313	1.3480167072065572	1.5428315184855816
min	-2.6232015869643073	-3.3774522470036055	-3.0091656985521182	-3.5905749411939913
25%	-1.2193563041690803	-1.1259661421673064	-0.36965072566327484	-1.5320908502265194
50%	-0.4464383691151155	-0.3829483773567014	0.6091655741127562	-0.5583745813634604
75%	0.8653559587494459	0.32653717620846257	1.5714875659807324	0.4450394048030798
max	4.598515378881905	3.2923727832868823	3.531827006207995	4.408265352517448

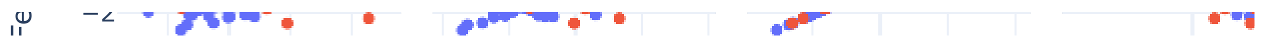
Show 25 per page

A scatter plot matrix shows us the correlation between each set of feature variables for each of the two classes.

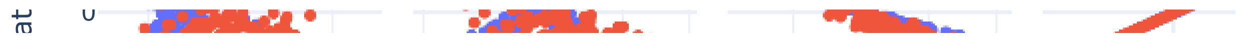
```
px.scatter_matrix(
    df,
    dimensions=['Feature1', 'Feature2', 'Feature3', 'Feature4', 'Feature5'],
    color='TargetClass', # The categorical version of the target variable
    title='Scatter plot matrix'
)
```

To use the k nearest neighbour classifier, we instantiate it and specify a value for k. We choose $k = 5$. There are more arguments available for this classifier, but we leave these at their default values.

```
neigh = KNeighborsClassifier(
    n_neighbors=5
)
```



Next up we fit the data to the instantiated classifier, using the `fit` method.



```
neigh.fit(
    X, # The multi-dimensional numpy array of feature variable valuess
    y # The numpy array of target variable values
)
```

```
KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski',
                     metric_params=None, n_jobs=None, n_neighbors=5, p=2,
                     weights='uniform')
```

The `predict` method allows us to pass values for an unknown observation and see which class the fitted classifier predicts.

```
np.random.seed(7)
unkown_obs = np.random.randn(5).reshape(1, -1) # A single observation with 5 ra

neigh.predict(unkown_obs)
```

```
array([1])
```

We see a predicted target class of `1`. The `predict_proba` method will return the probability for each target class given an observation.

```
neigh.predict_proba(unkown_obs)
```

```
array([[0., 1.]])
```

Class `1` was predicted with a 100% probability.

DATA SPLITTING

While we have *trained* a classifier, we are not sure how well it does. In most ML applications, we randomly split the dataset into a **training set** and a **test set**. The model trains of the former (as we did above). The latter is not used in the training, but is kept for obtaining metrics on our model.

This approach allows us to gauge how well a ML model might do on unseen data. This is of obvious importance, as we want to use our model on new data and have it perform well.

This brings with it the concepts of variance and bias, pertaining to how well the data does on the trainig set and how well it performs on unseen data.

High **variance** refers to a model that does very well on a training set. Such a model **overfits** the training data and might very well do poorly on unseen data. A model with high **bias** does rather worse on the training data. To some extent there is a trade-off between these.

A variety of factors influences variance and bias. The sample size is key. The more training data we have in ML, the better the model usually performs. Aspects such as the value of k in our example is termed a **hyperparameter** of the model. It is *something* about the model that we choose and must set. Note that there are techniques that can be used to let our computer *search for* the best hyperparameter values.

Below, we split the data into a 80% training set and a remainder of 20% test set. There is a trade-off here too. More data in the test set gives us a better indication of how well it will do on unseen data. We do then, however, take away observations that could have been used for training.

The `train_test_split` function takes various arguments. Below, we set the required arguments. This includes `test_size` set as a fraction of all the observations. We use the commonly used computer variables for the split data. The names are rather explanatory.

```
X_train, X_test, y_train, y_test = train_test_split(
    X,
    y,
    test_size=0.2,
    random_state=42
)
```

It is important to know that we have a fair representation of the classes in both sets. If not, we have **unbalanced** sets. This is a particulary interesting problem requiring its own solutions. The numpy `unique` function returns the sample space elements in an array. With the `return_counts` argument set to `True`, we also get a frequency count of each class.

```
np.unique(
    y_train,
    return_counts=True
)
```

```
(array([0, 1]), array([83, 77]))
```

```
np.unique(
    y_test,
    return_counts=True
)
```

```
(array([0, 1]), array([18, 22]))
```

There is a fair representation of each class in both the training and the test sets.

Now, we train the classifier again (still with $k = 5$).

```
neigh = KNeighborsClassifier(
    n_neighbors=5
) # Instantiate with k=5

neigh.fit(
    X_train,
    y_train
) # Train on the training data
```

```
KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski',
                     metric_params=None, n_jobs=None, n_neighbors=5, p=2,
                     weights='uniform')
```

METRICS

Given our trained model, we can now pass the unseen test set of feature variables to the model. The predicted target classes are assigned to the computer variable `y_pred` below.

```
y_pred = neigh.predict(X_test)
```

We can now use the predicted target classes to check on various metrics. One important metric is the accuracy. It returns the fraction of values that were precidicted correctly. We use the `accuracy_score` function from the metrics module of the scikit-learn package. As arguments, we pass the actual test target values, `y_test`, and the predicted classes for each test observation, `y_pred`

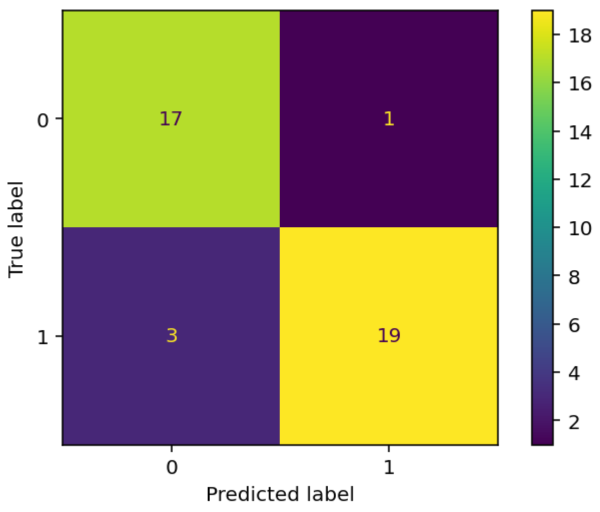
```
metrics.accuracy_score(
metrics.accuracy_score(
    y_test,
    y_pred
)
```

```
  0.9
```

Our model is 90% accurate on the unseen data.

A **confusion matrix** expresses the accuracy by showing the correctly and incorrectly predicted instances. The information in a confusion matix is clear to understand when plotted. We do this with the `plot_confusion_matrix` function from the metrics module of the scickit-learn package.

```
metrics.plot_confusion_matrix(neigh, X_test, y_test);
```



We see the true class labels along the left edge and the predicted class labels on the bottom edge. Looking at the plot, 17 class `0` observations in the test set were correctly predicted by the model as class `0`, with 19 class `1` observations correctly predicted. Three actual class `1` observations were incorrectly predicted to be class `0` and a single actual class `0` case was incorrectly prected to be class `1`.

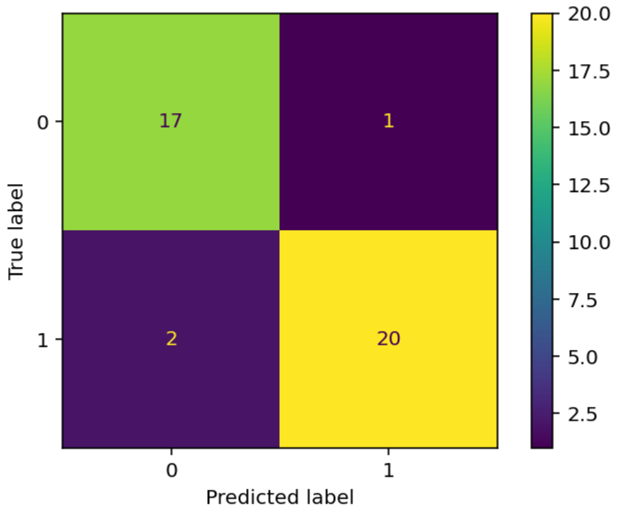
What if we changed the k hyperparameter to be 3?

```
neigh = KNeighborsClassifier(
    n_neighbors=3
) # Instantiate with k=3
neigh.fit(
    X_train,
    y_train
) # Train on the training data
```

```
  KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski',
                       metric_params=None, n_jobs=None, n_neighbors=3, p=2,
                                    weights='uniform')
```

The confusion matrix plot shows that we did a bit better.

```
metrics.plot_confusion_matrix(neigh, X_test, y_test);
```



The accuracy is now up to 92.5%.

```
y_pred = neigh.predict(X_test)

metrics.accuracy_score(
    y_test,
    y_pred
)
```

```
    0.925
```

k NEAREST NEIGHBOURS CLASSIFIER DATA SCIENCE EXAMPLE

DATA IMPORT

In this example we take a data set that can be downloaded from the internet. It contains observations for variables pertaining to the microscopic investigation of cells from breast lumps. Some of the observations are benign (non-canceorus) and some are malignant (cancerous). The spreadsheet file is contained in the `data` subfolder on this Google Drive.

```
#drive.flush_and_unmount()
```

```
# Connect to Google Drive
drive.mount('/gdrive', force_remount=True)

```

```
# Change directory to the DATA folder
%cd '/gdrive/My Drive/Stellenbosch University/School for Data Science and Comput
```

```
Mounted at /gdrive
/gdrive/My Drive/Stellenbosch University/School for Data Science and Computat
```

```
# Import the spreadsheet file
df = read_csv('breast_cancer.csv')
```

```
# First five observations
df[:5]
```

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

The `info` method gives us information about the dataframe object and the variable data types. We note that `diagnosis` is an object data type (a categorical variable).

```
# Information about the DataFrame object
df.info()
```

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

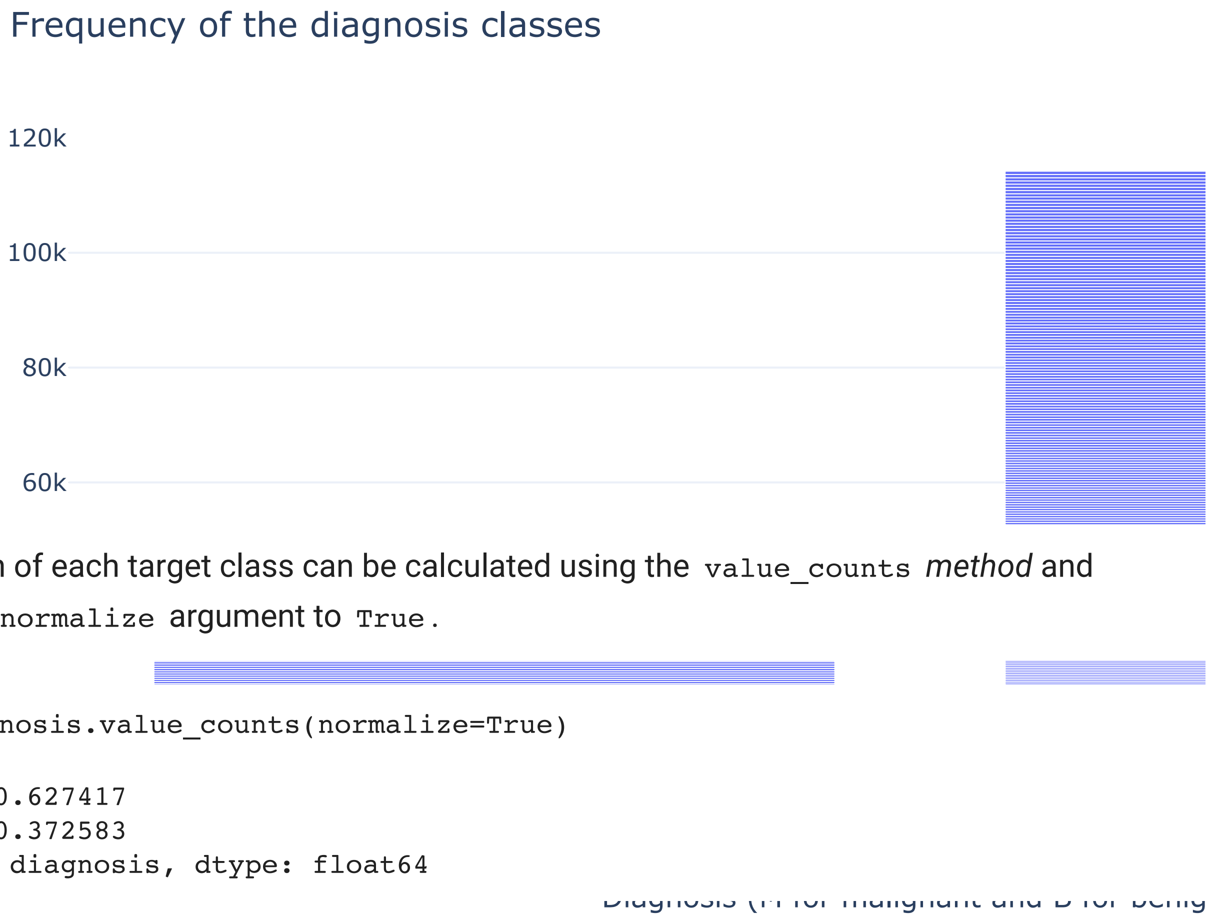
We can delete the `id` and the `Unnamed: 32` columns as it serves no purpose.

```
df.drop(
    ['id', 'Unnamed: 32'],
    axis=1,inplace=True
)
```

DATA SUMMARY

There is a known class imbalance in this dataset, with more benign disease than malignant disease. We can visualize and enumerate this.

```
px.bar(
    df,
    x='diagnosis',
    title='Frequency of the diagnosis classes',
    labels={
        'diagnosis':'Diagnosis (M for malignant and B for benign)'
    }
)
```



The fraction of each target class can be calculated using the `value_counts` *method* and setting the `normalize` argument to `True`.

```
df.diagnosis.value_counts(normalize=True)
```

```
B    0.627417
M    0.372583
Name: diagnosis, dtype: float64
```

The `describe` method is used to give a summary of the rest of the variables. The `loc` indexing is used to exclude the categorical target variable.

```
np.round(
    df.loc[:,df.columns!='diagnosis'].describe(),
    1
) # Rounding to a single secimal place
```

	radius_mean	texture_mean	perimeter_mean	area_mean	smoothness_mean
count	569.0	569.0	569.0	569.0	569.0
mean	14.1	19.3	92.0	654.9	0.1
std	3.5	4.3	24.3	351.9	0.0
min	7.0	9.7	43.8	143.5	0.1
25%	11.7	16.2	75.2	420.3	0.1
50%	13.4	18.8	86.2	551.1	0.1
75%	15.8	21.8	104.1	782.7	0.1
max	28.1	39.3	188.5	2501.0	0.2

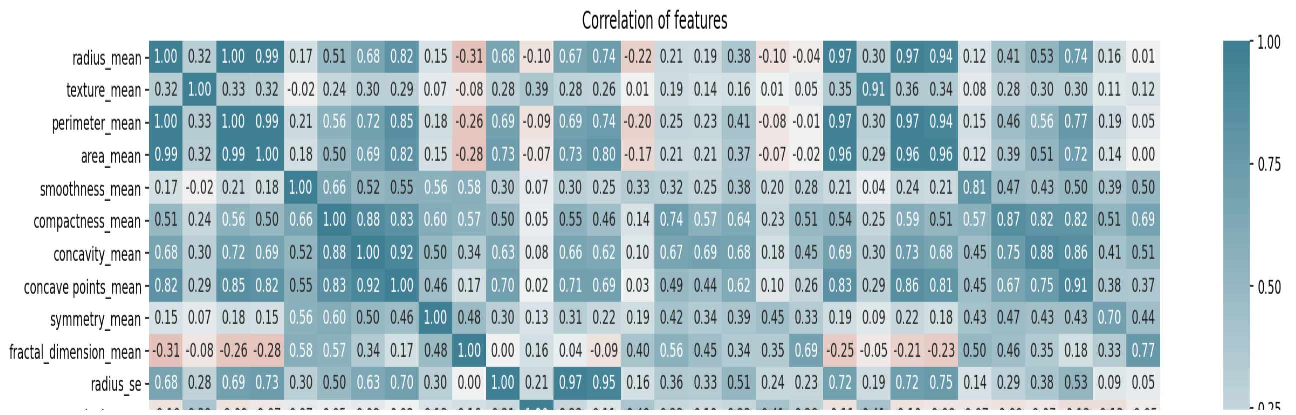
All the feature variables are numerical variables (for the kNN classifier). We can generate a correlation matrix to investigate the correlation between all pairs of feature variables, using the `corr` method.

```python
correlation = df.corr()
np.round(correlation, 2) # Rounding to two decimal places
```

	radius_mean	texture_mean	perimeter_mean	area_mean	sm
radius_mean	1.00	0.32	1.00	0.99	
texture_mean	0.32	1.00	0.33	0.32	
perimeter_mean	1.00	0.33	1.00	0.99	
smoothness_mean	0.17	-0.02	0.21	0.18	

We can visualize these correlations with a heatmap using the matplotlib package.

```
plt.figure(
    figsize=(21, 9)
)
plt.title('Correlation of features')
ax = sns.heatmap(
    correlation,
    vmin=-1,
    vmax=1,
    center=0,
    cmap=sns.diverging_palette(20, 220, n=200),
    annot=True,
    fmt='.2f',
    linecolor='white'
)
ax.set_xticklabels(
    ax.get_xticklabels(),
    rotation=90
)
plt.show();
```



DATA SPLITTING

Before we split the data into a training and a test set, we need to separate the features variables from the target variable.

```
# Generate the computer variable X from df with removal of the diagnosis column
y = df.diagnosis # The target variable

X = df.drop(
    ['diagnosis'],
     axis=1
)
```

We use the `train_test_split` function again to split 20% of the data as a test set.

```
X_train, X_test, y_train, y_test = train_test_split(
    X,
    y,
    test_size=0.2,
    random_state=12
)
```

We review the dimensions of the training and test sets.

```
X_train.shape, X_test.shape
```

```
((455, 30), (114, 30))
```

```
y_train.shape, y_test.shape
```

```
((455,), (114,))
```

DATA SCALING

Scaling data is an important step in ML. It puts all the variables within a similar numerical interval. This has advantages for the training step of many ML algorithms. There are various ways to scale data. Here, we will use **standard scaling**, where the mean of a variable is subtracted from each value in that variable and this difference individed by the standard deviation of that variable, shown in (3), where z is the scaled value, x_i is each value for the variable, $\bar{x}$ is the mean and s_X the standard deviation of the variable.

$$z = \frac{x_i - \bar{X}}{s_X} \tag{3}$$

To use this scaler, we instantiate the class.

```
# Generating an instance of the StandardScaler class with default argument value
scaler = StandardScaler(
    copy=True,
    with_mean=True,
    with_std=True
)
```

The training set is first fitted and then transformed (in one step) to the scaler using the `fit_transform` method.

```
X_train = scaler.fit_transform(X_train)
```

The test data is transformed with the attributes of the scaling of the training set. This is very important. The test set must not be scaled using its own mean and standard deviation.

```
X_test = scaler.transform(X_test)
```

TRAINING

We follow the same steps as used in our initial introduction to the kNN classifier. We will use $k = 3$ as hyperparameter value.

```
# Instantiating the classifier with k=3
knn = KNeighborsClassifier(
    n_neighbors=3
)
```

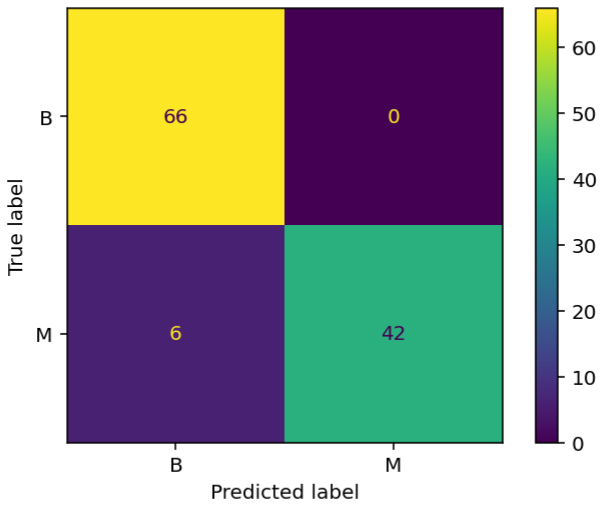
```
# Fit the training set
knn.fit(
    X_train,
    y_train
)
```

```
KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski',
                     metric_params=None, n_jobs=None, n_neighbors=3, p=2,
                     weights='uniform')
```

METRICS

The confusion matrix plot shows how well the model faired when using the unseen test data.

```
metrics.plot_confusion_matrix(
    knn,
    X_test,
    y_test
);
```



The accuracy is calculated below using the `accuracy_score` function.

```
metrics.accuracy_score(
    y_test,
    knn.predict(X_test)
)
```

```
0.9473684210526315
```

We compare this to the accuracy of the model using the training set, this time using the alternative approach of the `score` method of the model.

```
knn.score(
    X_train,
    y_train
)
```

```
0.978021978021978
```

This gives the same result as the `accuracy_score` function.

```
metrics.accuracy_score(
    y_train,
    knn.predict(X_train)
)
```

```
0.978021978021978
```

There is definitely some overfitting (high variance) as the model does better on the training data than on the test data.

We can also look at the balanced accuracy score using the `balanced_accuracy_score` function. This metric allows for class imbalance.

```
metrics.balanced_accuracy_score(
    y_test,
    knn.predict(X_test)
)
```

```
0.9375
```

This score is still much better than the **null score**, which is the fraction of the majority class. Since `y_test` is a pandas series object, we can use the `value_counts` method with the `normalize` argument set to `True`.

```
# Null or baseline score based on proportion majority class
y_test.value_counts(
    normalize=True
)
```

```
B    0.578947
M    0.421053
Name: diagnosis, dtype: float64
```

The majority class in the test set is `B` (benign disease), with a fraction of 0.58. If we simply use the majority class as predictor, we would be correct 58% of the time. Our model therefor improves our prediction.

There are other more important metrics such as the sensitivity (recall), the specificity, the positive predictive value (precision), and the negative predictive value. All of these required us to understand the concepts of true and false positive and negatives.

The two classes in our target variable for the current example is `B` for benign and `M` for malignant. As data scientists, we choose one of these classes as our *class of interest*, i.e. the one that we want to predict. In this case, it can be `M`. Given an unknown new observation (set of values for all the feature variables), the model will predict `M` with some probability. We can set a threshold on the interval $[0, 1]$. If the probability for `M` is above the threshold, the model predicts `M`, else it predicts `B`. We can set this threshold depending on how costly mistakes (in either direction) are given the cicumstances in which the model is used. By default, the threshold is 0.5.

If we look at the first observation in the test set, we note that its true class was `M`.

```
y_test.iloc[0]
```

```
'M'
```

We can use the first row in the feature set to see what the model predicts.

```
knn.predict(X_test[0].reshape(1, -1))
```

```
array(['M'], dtype=object)
```

So, if our class of interest was `M`, then the model's prediction would be termed a **true positive**. Here positive refers to the class of interest. If the predicted class was `B`, this would be a **false negative**. Here negative is assigned purely on our research approach and which class we are interested in.

If the true class was `B` and the model predicted a `B`, then this would be a **true negative**. If it predicted `M`, though, it would a `false positive`.

The following abbreviations are often used. The values from our last confusion matrix plot are added under the assumption of `M` being our positive class.

Metric	Abbreviation	Example value
True positive	TP	42
True negative	TN	66
False positive	FP	0
False negative	FN	6

The equations for our four new metrics are shown in (4), where PPV is positive predictive value and NPV is negative predictive value.

$$
\begin{aligned}
\text{sentitivity (recall)} &= \frac{TP}{TP + FN} \\
\text{specificity} &= \frac{TN}{TN + FP} \\
\text{PPV (precision)} &= \frac{TP}{TP + FP} \\
\text{NPV} &= \frac{TN}{TN + FN}
\end{aligned}
\tag{4}
$$

Recall is then the fraction of true positive cases predicted as such by the model. **Specificity** is the fraction of true negative cases predicted as such by the model. **Precision** is the fraction of cases that were correctly predicted as positive over all the cases that were predicted as positive. **Negative predictive value** is the fraction of cases that were truely negative over all the cases that were predicted to be negative.

> These metrics are domain specific. In healthcare for instance, the sensitivity (more often used in this setting than the Data Science term recall) is the ability of a test to return a positive result given all truely positive cases. Sensitivity is the ability of a test to correctly identity truely negative cases. Positive predictive value (more often used in this setting than the Data Science term precision) is used after the test returns a positive result and expresses the probability of the actual result being positive. Finally, negative predictive value is also used after a test is done and expresses the probability of a negative result actually being negative.

Another metric is the **f1 score**. This reflects the *balance* between the precision and recall, as shown in (5).

$$
f_1 = 2 \times \frac{\text{precision} \times \text{recall}}{\text{precision} + \text{recall}} = \frac{\text{TP}}{TP + \frac{1}{2}(FP + FN)} \tag{5}
$$

Some of the metrics are returned using the `classification_report` function.

```
print(metrics.classification_report(
    y_test,
    knn.predict(X_test)
))
```

```
              precision    recall  f1-score   support

           B       0.92      1.00      0.96        66
           M       1.00      0.88      0.93        48
```

```
      accuracy                           0.95       114
     macro avg       0.96      0.94      0.94       114
  weighted avg       0.95      0.95      0.95       114
```

We have raised the question as to when a value is predicted as a certain class. The kNN algorithm produces a probability for each class (the fraction of actual k classes in the neighbourhood of an observation. Since we are dealing with an odd number of neigbours, the majority class in this neigbourhood *rules*. Below, we look at the first 10 probabilities predicted from the test set feature variables.

```
knn.predict_proba(X_test)[0:10]
```

```
array([[0.        , 1.        ],
       [1.        , 0.        ],
       [1.        , 0.        ],
       [1.        , 0.        ],
       [1.        , 0.        ],
       [0.66666667, 0.33333333],
       [0.66666667, 0.33333333],
       [1.        , 0.        ],
       [0.66666667, 0.33333333],
       [0.        , 1.        ]])
```

For most cases all three nearest neighbours were of the same class, but in three of them only two of the neighbours were of the same class.

Given larger values of k we will see different probabilities. Below, we choose $k = 7$ and look at the accuracy metric and at the first 10 observation probabilities again.

```
# Instantiate the classifier
knn_7 = KNeighborsClassifier(
    n_neighbors=7
)
```

```
# Train the model
knn_7.fit(
    X_train,
    y_train
)
```

```
KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski',
                     metric_params=None, n_jobs=None, n_neighbors=7, p=2,
                     weights='uniform')
```

```
# Accuracy on the test set
knn_7.score(
    X_test,
    y_test
)
```

```
0.956140350877193
```

We get an improved accuracy.

```
knn_7.predict_proba(X_test)[0:10]
```

```
array([[0.        , 1.        ],
       [1.        , 0.        ],
       [1.        , 0.        ],
       [1.        , 0.        ],
       [1.        , 0.        ],
       [0.85714286, 0.14285714],
       [0.71428571, 0.28571429],
       [1.        , 0.        ],
       [0.57142857, 0.42857143],
       [0.        , 1.        ]])
```

Below, we generate a DataFrame object from these probabilities (using the test set).

```
probabilities = DataFrame(
    knn_7.predict_proba(X_test),
    columns=['B', 'M']
)

probabilities[:10]
```

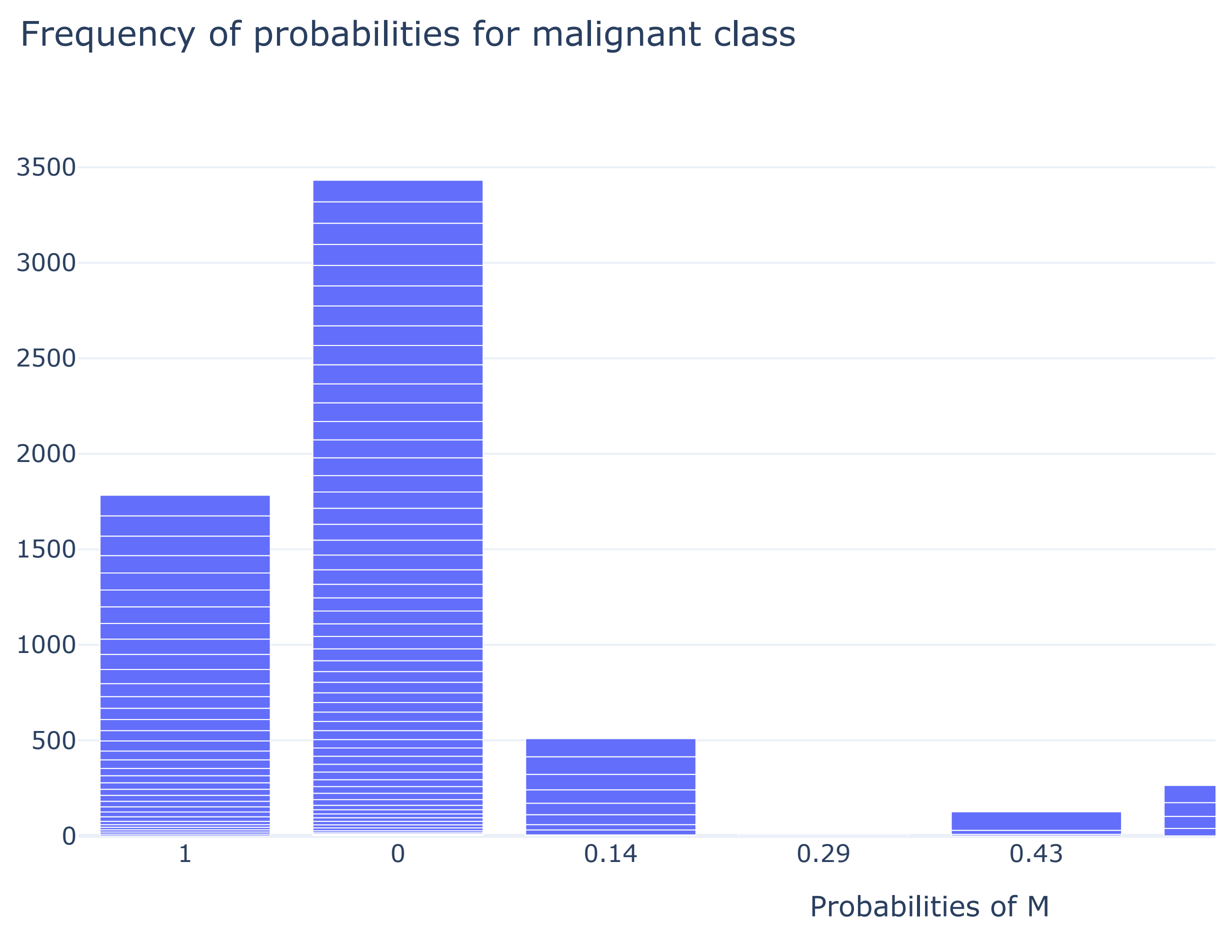
1 to 10 of 10 entries Filter ?

index	B	M
0	0.0	1.0
1	1.0	0.0
2	1.0	0.0
3	1.0	0.0
4	1.0	0.0
5	0.8571428571428571	0.14285714285714285
6	0.7142857142857143	0.2857142857142857
7	1.0	0.0
8	0.5714285714285714	0.42857142857142855
9	0.0	1.0

Show 25 per page

We can now create a bar chart to show the frequency of each probability for the `M` class.

```
px.bar(
    np.round(probabilities, 2),
    x='M',
    title='Frequency of probabilities for malignant class',
    labels={'M':'Probabilities of M'}
).update_xaxes(
    type= category
)
```



All the observations that had a probability of `M` greater than 50% (0.5) is predicted to be `M` by the model. What if we change this threshold, though? This can be done depending on how *expensive* mistakes are. If it is costly to miss a positive results, then we can set the threshold lower so that more observations are predicted to be positive. The meaning of the term *expensive* is determined by the setting of the Data Science project.

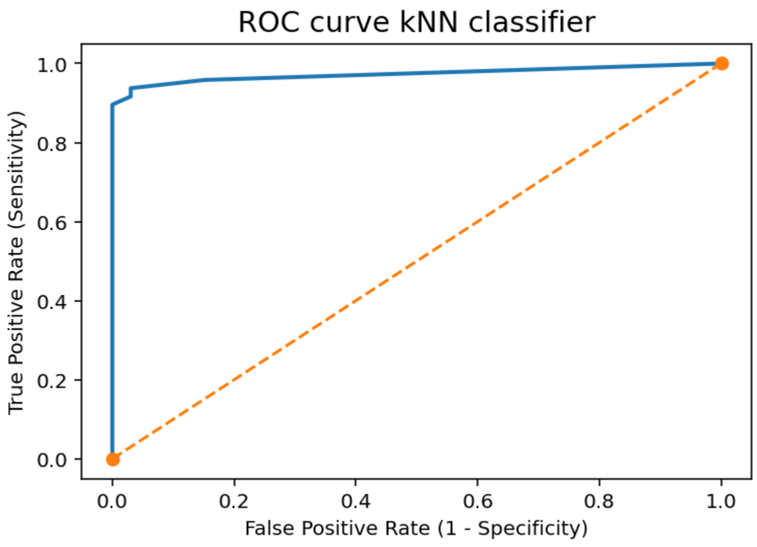
A **receiver operator characteristic** (ROC) curve presents a visual representation of different thresholds. The *x* axis of this plot is $1 - \text{specificity}$ and the *y* axis is the recall. To use this plot, we first calculate the required values using the `roc_curve` function.

```
fpr, tpr, thresholds = metrics.roc_curve(
    y_test.replace(['B', 'M'], [0, 1]),
    knn_7.predict_proba(
        X_test
    )[:, 1]
)
```

Below, we use the matplotlib package to generate the curve.

```
plt.plot(fpr, tpr, linewidth=2)
plt.plot([0,1], [0,1], 'o--')
plt.rcParams['font.size'] = 12
plt.title('ROC curve kNN classifier')
plt.xlabel('False Positive Rate (1 - Specificity)')
plt.ylabel('True Positive Rate (Sensitivity)')

plt.show();
```



The orange dotted line represents a $50:50$ *chance*. We want our curve to be higher than this line, which indeed it is. For a specificity of just under 100% (to the left of the *x* axis), we get almost 90% recall.

The area under the curve(ROC AUC) represents the ROC score. The closer the ROC AUC is to 1.0 the better the model performance. The `roc_auc_score` function calucates this area. Note that we use the numpy `where` function to replace `B` with 0 and `M` with 1 since we need numerical values.

```
metrics.roc_auc_score(
    y_test.replace(['B', 'M'], [0, 1]),
    np.where(
        knn_7.predict(X_test) == 'B', 0, 1
    )
)
```

```
0.9479166666666667
```

The ROC score or ROC AUC is 0.95, which is very good.

For both kNN classifiers ($k=3$ and $k=7$) we have only performed a single training step. We might have been very *lucky* or *unlucky* in the random split of a training and a test set. It is better to repeat this process many times over. This is termed cross-validation.

CROSS VALIDATION

With **cross-validation** we split the data repeatedly and measure performance metrics, over which we average in the end. In k fold cross validation, we choose a number of folds. As an example, we might choose $k=5$. Note that this is not the k of kNN. For a 5 fold cross validation, the data is split into training and test sets five times, such that all the data is used for training and testing.

When the `scoring` argument is set to `accuracy`, the `cross_val_score` function from the model_selection module of the scikit-learn package returns the accuracy k number of times.

```
scores = cross_val_score(
    knn_7,
    X,
    y,
    cv=5,
    scoring='accuracy'
)
```

```
scores
```

```
array([0.87719298, 0.93859649, 0.94736842, 0.94736842, 0.92035398])
```

The average of these scores gives a better understanding of model performance on unseen data.

```
np.mean(scores)
```

```
0.9261760596180716
```

```
np.min(scores), np.max(scores)
```

```
(0.8771929824561403, 0.9473684210526315)
```

There is quite a large range for these scores, indicating that the accuracy is very dependent on which observations are in the training and the test sets. In this case, it is a function of the small number of observations in the data set.

The final question in this section is which value of k for the kNN classifier is best. One method for finding the optimal value is to perform a grid search.

GRID SEARCH FOR BEST HYPERPARAMETER VALUES

With a grid search we can explore the solution space for hyperparameter values. This process is known as **hyperparameter tuning**.

The hyperparameters we will tune for the kNN classifier here are the leaf count, the number of neighbours, and the distance metric. While we used Euclidean distance in this notebook, there are other distance metrics too. This argument is set using the `p` value when instatiating the classifier. The leaf count pertains to the search algorithm used for determining the closest neighbours. The kNN classifier used in scikit-learn can use a few of these algorithms such as the KD-tree algorithm or the ball-tree algorithm, or even a brute force approach.

To use a grid search, we set values to explore.

```
# Generate a Python list of leaf size values on the closed interval 1 through 4
leaf_size = list(range(1, 50))

# Generate a Python list of neigbour numbers on the closed interval 1 through 1
n_neighbors = list(range(1, 20))

p = [1, 2]
```

We convert these lists to a dictionary.

```
hyperparams = {
    'leaf_size':leaf_size,
    'n_neighbors':n_neighbors,
    'p':p
}
```

Next we instantiate a new kNN classifier with default argument values.

```
knn = KNeighborsClassifier()
```

Now we perform the grid search, which will go through all the hyperparameter values when we fit the data. We also use 5 fold cross validation. All of this can be computationally expensive (consuming a lot a computer resources and taking a long time).

```
knn_grid_search = GridSearchCV(
    knn,
    hyperparams,
    cv=5
)
```

```
best_params = knn_grid_search.fit(
best_params = knn_grid_search.fit(
    X,
    y
)
```

The process took over 90 seconds on Colab.

Now we can print the best hyperparameter values using the `best_estimator.get_params` method.

```
# Best leaf size
best_params.best_estimator_.get_params()['leaf_size']
```

```
1
```

```
# Best number of neighbours
best_params.best_estimator_.get_params()['n_neighbors']
```

```
9
```

```
# Best distance metric
best_params.best_estimator_.get_params()['p']
```

```
1
```

Below, we use these hyperparameter values and 5 fold cross validation. Note that these may be different every time you run the code. Below, we see the best parameter values generated during a previous run.

```
knn_best = KNeighborsClassifier(
    n_neighbors=9,
    leaf_size=9,
    p=1
)
```

```
scores = cross_val_score(
    knn_best,
    X,
    y,
    cv=5,
    scoring='accuracy'
)
```

The average accuracy is now better than before.

```
np.mean(scores)
```

```
0.9385188635305077
```

k NEAREST NEIGHBOUR REGRESSION

The k nearest neighbour (kNN) algorithm can also be used for regression. Here the target variable is a continuous numerical variable.

GENERATING A DATA SET

To understand the basic concept of building a kNN regression model, we start by generating a data set, with a single feaure variable, and then visualise the data.

```
X = np.arange(
    start=1,
    stop=11
)

X # A 10 element array
```

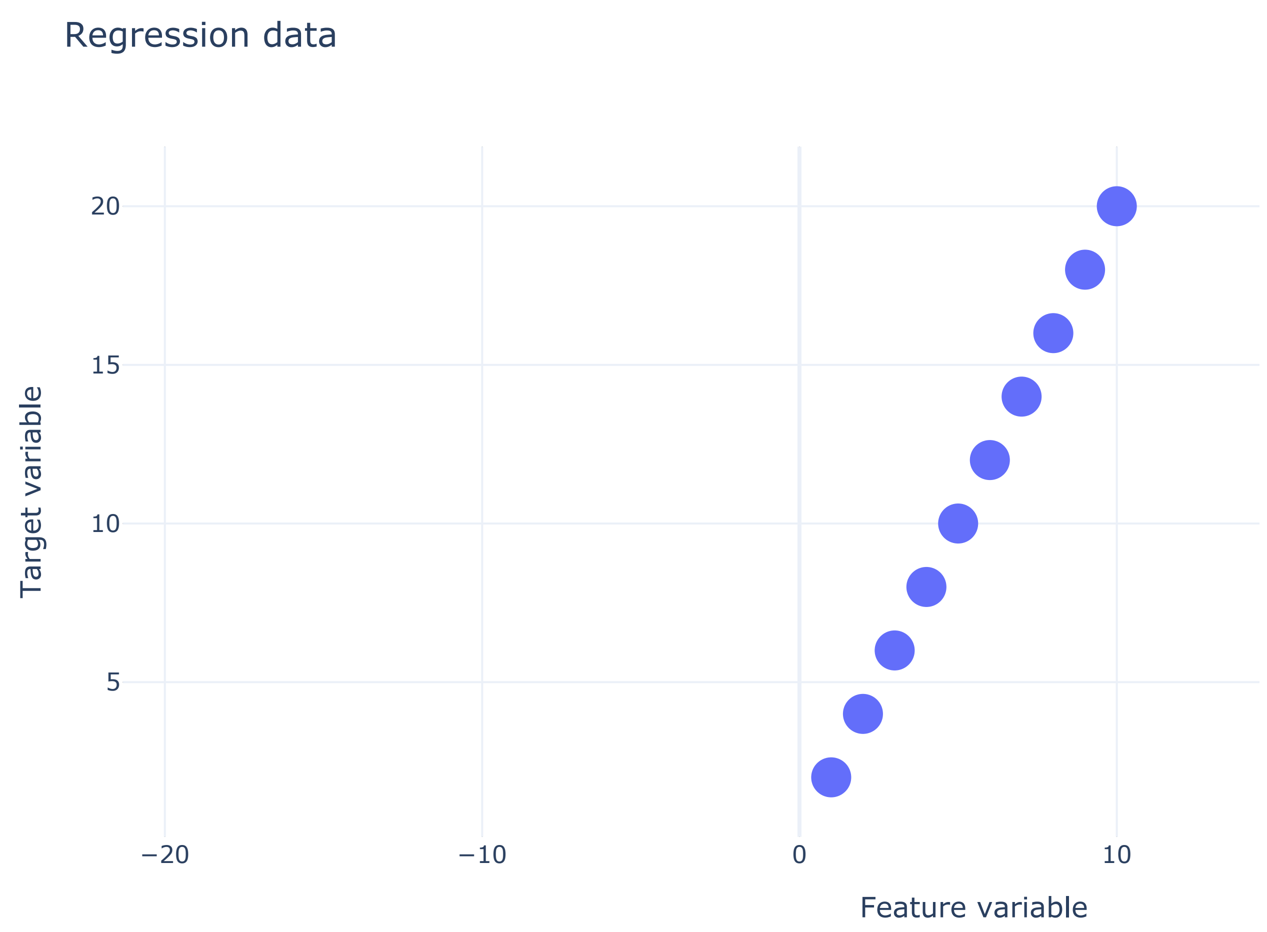
```
array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10])
```

```
y = 2 * X

y
```

```
array([ 2,  4,  6,  8, 10, 12, 14, 16, 18, 20])
```

```
go.Figure(
    go.Scatter(
        x=X,
        y=y,
        name='Data',
        mode='markers',
        marker={
            'size':20
        }
    )
).update_yaxes(
    scaleanchor='x',
    scaleratio=1
).update_layout(
    title='Regression data',
    xaxis={'title':'Feature variable'},
    yaxis={'title':'Target variable'}
)
```



CREATING A MODEL

We instantiate a kNN regressor with $k = 3$ nearest neighbours. The leaf size, search method, and distance measure arguments are left at their default values.

```
# Instantiating a kNN regressor with k=3 neighbours
knn = KNeighborsRegressor(
    n_neighbors=3
)
```

TRAINING THE MODEL

Next, we fit the data to the model for training.

```
knn.fit(
    X.reshape(-1, 1),
    y
)
```

```
KNeighborsRegressor(algorithm='auto', leaf_size=30, metric='minkowski',
                    metric_params=None, n_jobs=None, n_neighbors=3, p=2,
                    weights='uniform')
```

TESTING THE MODEL

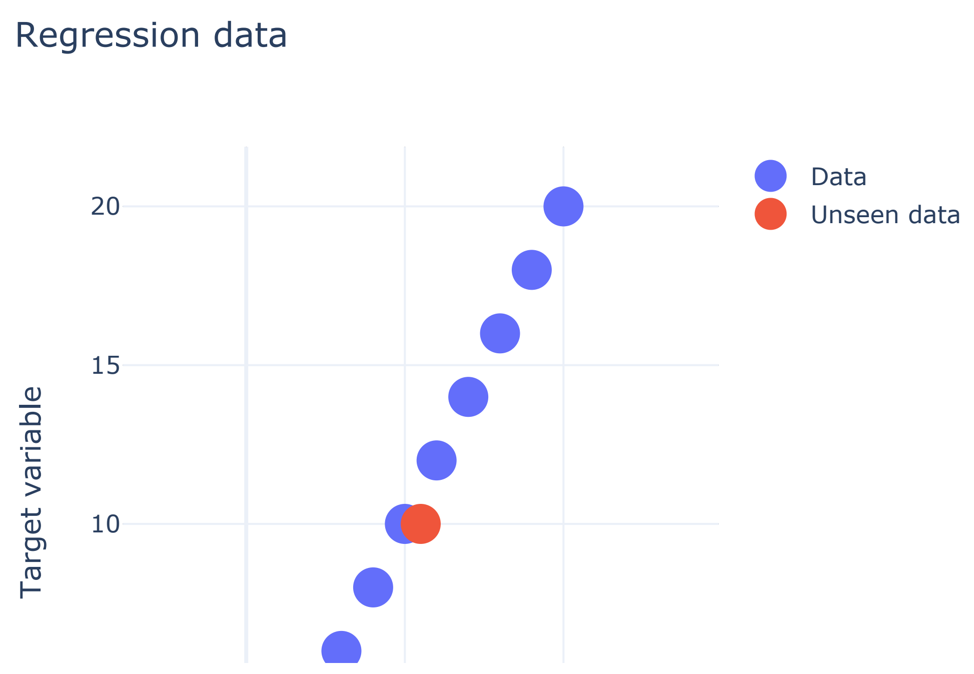
We can now pass a value to the model to see what it predicts and then try to understand the result.

```
knn.predict(np.array([5.5]).reshape(1, -1))
```

```
array([10.])
```

We see a result of 10. We can plot this to visualize the prediction in view of the data.

```
go.Figure(
    go.Scatter(
        x=X,
        y=y,
        name='Data',
        mode='markers',
        marker={
            'size':20
        }
    )
).add_trace(
    go.Scatter(
        x=[5.5],
        y=[10],
        name='Unseen data',
        mode='markers',
        marker={
            'size':20
        }
    )
).update_yaxes(
    scaleanchor='x',
    scaleratio=1
).update_layout(
    title='Regression data',
    xaxis={'title':'Feature variable'},
    yaxis={'title':'Target variable'}
)
```



The three nearest observations have target variable values of 9, 10, and 11. The average of this is 10.

CONCLUSION

This notebook was an introduction to the world of machine learning using one of the most interpretable algorithms. Through the examples, we have gained valueble knowledge of terms used in machine learning, the construction of these models, and how to evaluate them.

```
1
```

✓ 0s completed at 14:09 ● ×