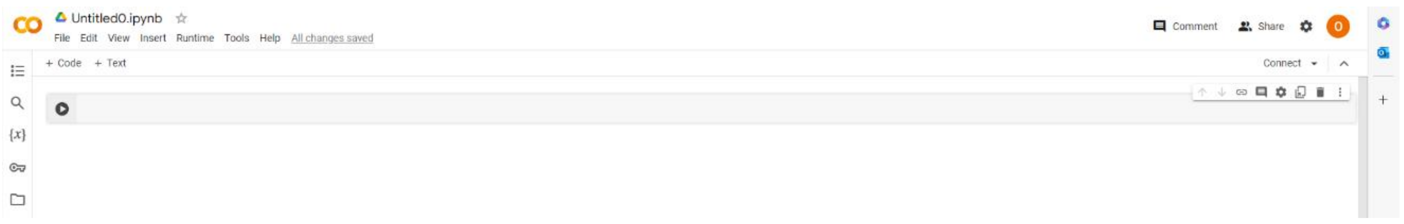
Introduction to Machine Learning

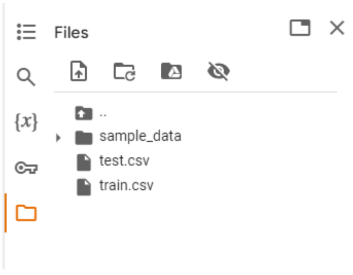
Machine Learning (ML) is a branch of Artificial Intelligence which focuses on the use of algorithms to learn patterns from datasets. We can use Machine Learning for a wide variety of applications, from predicting house prices to identifying cancer tumours in CT scans.

In this worksheet, we will be looking at how to get started with your first Machine Learning project by attempting to develop a model capable of predicting which passengers survived the sinking of the Titanic.

- To get started open https://colab.google/
- Sign into your Google account and select 'New Notebook'



- Download the following files train.csv, test.csv.
- Click on the folder icon on the left-hand side of the screen. Drag train.csv and test.csv into the window to upload them to your notebook environment. For the moment, we will be focusing on **train.csv** which contains data from 891 passengers. This is the data our model will learn from in order to make predictions.



There are lots of different types of Machine Learning models, each of which serve different purposes. For example, **Regression** models are used to predict a specific number (continuous variables), such as predicting future house prices, whereas **Classification** models predict discrete variables (meaning different categories), such as whether an image is of a cat or a dog.

As we will be predicting whether a passenger survives the sinking of the Titanic, we will be creating a **Binary Classification** model where there are only two possible outcomes: 1 = Survived and 0 = Not Survived.

- The Google colab notebook is made up of 'cells'. Within each cell you can insert blocks of code to perform specific tasks. Unlike a traditional programming where you run your entire program, you can run each cell independently by pressing the **play button**, or by pressing **Shift** and **Enter** together.
    - After running a cell, you will see a new cell is automatically generated below. You can also manually create cells by selecting **Insert** and **Code Cell** from the navigation menu.
- Copy the following code into your first cell and run it.
  The code performs two functions. It imports two libraries *numpy* and *pandas* into our notebook and creates a dataframe (similar to a spreadsheet) called **train_df** from the train.csv file.

```
import numpy as np #commonly used library used for scientific computing
import pandas as pd #pandas dataframes are what we will use to hold our
data - similar to an excel spreadsheet
train_df = pd.read_csv("train.csv")
```

- You can preview a sample of the dataset by placing the following line of code in the next cell. Remember to press the play button (or Shift and Enter) to run the code.

```
train_df.head()
```

The dataset has the following fields:

PassengerId – unique identifier of each passenger (not relevant to the models we will be producing)

Survived – this is what we will try and predict. 1 = Survived, 0 = Not survived

Pclass – Passenger class which can be used to represent the socio-economic status of the passengers, i.e., passengers in upper class are likely to be more wealthy than those in lower class.
1 = Upper Class, 2 = Middle Class, 3 = Lower Class

Name –The passenger's name.

Sex – Whether a passenger is male or female.

Age – the age of the passenger at the date of the journey (in years old).

SibSp – the total number of passengers' siblings and spouse.

Parch – the total number of the passengers' parents and children. Like SibSP, this helps us to understand whether a specific passenger was travelling alone or as part of a family.

Ticket – the passenger's individual ticket number.

Fare – how much the passenger paid for their ticket.

Cabin – the passenger's cabin.

Embarked – the port from which the passenger joined Titanic.
C = Cherbourg, Q = Queenstown, S = Southampton

- In nearly all real-world datasets, there are likely going to be missing pieces of information – especially in a dataset as old as this one. The machine learning models we are going to be using require any missing values to be accounted for before we can begin training.
- Place the code below in the next empty cell. Missing information is represented by 'NaN', therefore, this piece of code is going to count all occurrences of NaN for each feature.

```
train_df.isna().sum()
```

- Run the code in the cell (Shift + Enter) and you should receive the following output:

```
PassengerId 0
Survived    0
Pclass      0
Name        0
Sex         0
Age         177
SibSp       0
Parch       0
Ticket      0
Fare        0
Cabin       687
Embarked    2
dtype: int64
```

- From the output we can see three features have missing values – **Cabin, Age** and **Embarked**.
    - Cabin information is missing from over 75% of passengers. Our dataset would become extremely constrained If we were to remove all of these passengers. Instead, we should drop the Cabin feature from our dataset entirely, as the amount of missing information limits its usefulness to our model. We can drop a feature using the following code:

    ```
    col_to_drop = ['Cabin']
    train_df.drop(col_to_drop, axis = 1, inplace = True)
    ```

    - Around 20% of passengers are missing data relating to their age, meaning that there is still enough data for us to work with, but we must handle the missing value.
    We can view the distribution of passenger's ages by plotting a histogram:

    ```
    train_df['Age'].plot.hist(bins = 40)
    ```

    As the age is approximately normally distributed we impute the missing ages with the mean value of all passengers' ages. This approach avoids adding any bias to our modules, but its simplistic approach could potentially hamper our model's performance as it makes any true relationships with this feature harder to establish.

```
train_df['Age'].fillna(train_df['Age'].mean(), inplace =
True)
```

- Similarly, only two passengers missing information regarding where they embarked from. As this is a categorical variable, we can use the most common value (the mode). The small number of missing values for this feature are not likely to have a significant impact on the performance of our model.

```
train_df['Embarked'].fillna(train_df['Embarked'].mode()[0],
inplace = True)
```

You should use the example of dropping the Cabin column to also drop Name and Ticket as these are unique pieces of information to each passenger and as such, will not be useful to our models. **Note.** Do not drop the Passenger ID Column.

All remaining columns (a.k.a. features) are now numeric, with the exception of Sex and Embarked. In order for these features to be used within our model we must transform these features from text to numbers. This is achieved through the use of **Dummy Variables** which replace textual categories with numbers.
For example: male = 1, female = 0.

```
from sklearn.preprocessing import LabelEncoder
le = LabelEncoder()
cols_to_change = ['Sex', 'Embarked']
train_df[cols_to_change] =
train_df[cols_to_change].apply(le.fit_transform)
```

You are now ready to train our first Machine Learning model!
The first model we are going to try is called a **Decision Tree Classifier**. Decision trees make predictions using a series of decision rules which can only be answered using true or false. These rules are combined together resulting in a tree-like structure, with the final output for a binary classification model being true or false. For our scenario this refers to whether a passenger survived or not.

Copy the code on the next page into a new cell. Read through the comments in green in order to ensure you understand each step being taken.

```
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score
#Creating a list of the features (columns) of information to be
inputted into the model.
#Consideration for later - how can we choose what to include here? Some
features are more important than others.
#Research 'Feature Selection' methods as extension work.
features = ['Pclass', 'Sex', 'Age', 'SibSp', 'Parch', 'Fare',
'Embarked']
```

```
#X represents our input features into our model
X = train_df[features]
#y represents what we are trying to predict - our target variable.
y = train_df['Survived']

#Splitting the data. 80% used for training. 20% used for testing
X_train, X_test, y_train, y_test = train_test_split(X,y ,
                                   random_state = 24,
                                   test_size=0.20,shuffle=True)

#Training the model. Random state used for consistency between runs.
model = DecisionTreeClassifier(random_state = 24)
model.fit(X_train, y_train)
#Outputting how well the model fits the training data.
print(model.score(X_train, y_train))

#Making predictions for the test set
y_predicted = model.predict(X_test)
#Output the accuracy of the predictions on the test set.
accuracy_score(y_test, y_predicted)
```

A Decision Tree is a type of **supervised learning** algorithm as it uses the true values of what we are trying to predict within the training data in order to establish the patterns and trends within the data which can be used to make predictions.

The output from the code above should give you two numbers:

```
0.9845505617977528
0.7597765363128491
```

The first shows us how well our model fits the training data – i.e. how well it fits data it has been told the answer too. This equates to an accuracy of approximately 98% which is very high. Actually, this is suspiciously high!

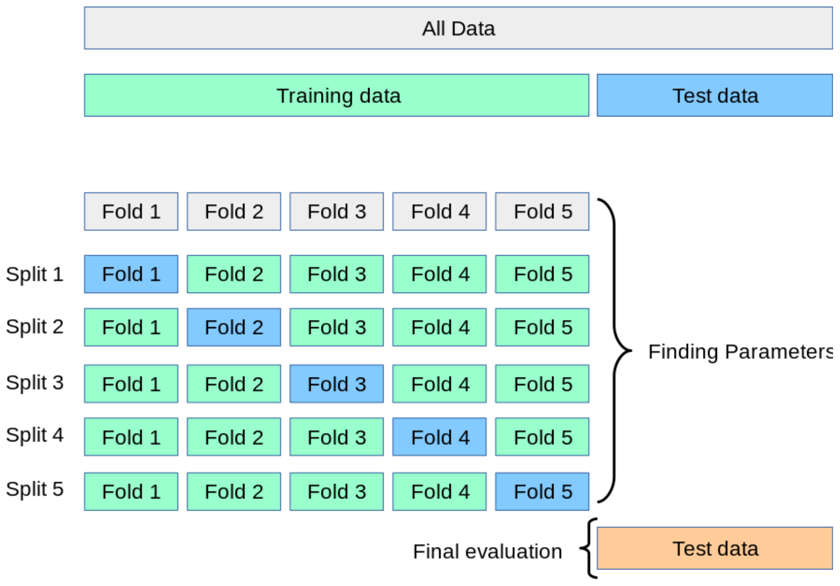
The second number is how well our model performed on data it has not seen before. This time, the accuracy is quite a bit lower, at around 76%. These results mean our model might be **overfitting**, where it understands the data it has been trained on too well, so does not work well on unseen data. This would mean that in real-world usage, our model would not perform very well.

One of the unique advantages of using Decision Trees is that we can create a plot of the tree in order to understand how it is making its predictions. Copy the code below into a new cell and run it. You will see the tree which has been produced is very large and complex, supporting the theory that our model is overfitting.

```
from sklearn import tree
tree.plot_tree(model)
```

Looking back at the code, we can see we separated out 20% of our overall training data to act as a test set. This is quite a small amount, so the results may not actually be representative of real-world performance.

In order gain a more reliable estimate of performance we can use **K-Fold Cross Validation** to partition our dataset into *k* sections (folds). For example, where *k* = 5, our dataset will be partitioned into 5 equal sections, and our model will be trained 5 times, with a different section being used to test rather than train the model.



See: https://scikit-learn.org/stable/modules/cross_validation.html

Copy the code below into a new cell and run it.

```
from sklearn.model_selection import cross_val_score
import numpy as np
cv_model = DecisionTreeClassifier(random_state = 24)
cv_scores = cross_val_score(cv_model, X_test, y_test, cv = 5)
avg_cv = np.mean(cv_scores)
print(cv_scores)
print("Average CV accuracy:", avg_cv)
```

This average gives us a more realistic estimate of how our model performs on underseen data. However, we want to try and improve the performance of our model. To do this we use **Hyperparameters**, which allow us to alter the inner-workings of the model to best fit our scenario. In order to find the optimal value for a hyperparameter we can perform a **Grid Search**, wherein the cross validation process described earlier is repeated several times, with a different value for the hyperparameter under evaluation being used each time.

For example, **ccp_alpha** is used to remove parts of the tree through a process known as 'pruning' in order to reduce the potential for overfitting. In order for this process to work correctly, we need to choose an appropriate value for ccp_alpha. If the value is too large, our tree would lose important information and struggle to make predictions. If it is too small, pruning would not be effective, and the tree would overfit the training data.

Copy the code below into a new cell and run it.
ccp_alpha_list contains a list of different values to be trialed.

```
from sklearn.model_selection import GridSearchCV
model = DecisionTreeClassifier()
ccp_alpha_list = [0.001, 0.01, 0.1, 1.0, 10]
grid = GridSearchCV(estimator = model, param_grid = {'ccp_alpha' :
ccp_alpha_list}, cv = 5)
grid.fit(X_train, y_train)
print(grid.best_score_, grid.best_params_)
```

The output indicated the average accuracy (established through cross validation) for the best performing model, along with the value for the hyperparameter. For example:

```
0.8075642667191962 {'ccp_alpha': 0.01}
```

You should experiment changing the values in the ccp_alpha_list to include more values between 0.001 and 0.01. **Note:** Adding lots of values will increase the total training time exponentially, so you should try to limit the number you are trialing. Feel free to remove any values > 0.01.

When you are happy with your model copy the code below into a new cell. You should replace 0.01 with your chosen value for ccp_alpha.

Your tree plot should now be much smaller, meaning you have produced a model which should be more generalisable, meaning it works better with unseen data.

```
model = DecisionTreeClassifier(ccp_alpha=0.01)
model.fit(X_train, y_train)
print(model.score(X_train, y_train))

y_predicted = model.predict(X_test)
accuracy_score(y_test, y_predicted)
tree.plot_tree(model)
```

Whilst Decision Trees are very explainable, their simplistic structure still makes them susceptible to overfitting. The code below demonstrates using a different type of model called a Random Forest. A Random Forest is an ensemble model, which uses a large number of Decision Trees each of which have been trained on a random subset of the training data and also a random subset of the features. By having a wide variety of trees which have all been constructed differently, Random Forests are much less suspectable to overfitting than individual Decision Trees.

Copy the code below into a new cell. You will notice it is very similar to the Decision Tree code, only this time we are using a Random Forest Classifier. Like the Decision Tree, we can optimise the ccp_alpha hyperparameter. However, we can also optimise the n_estimators hyperparameter which controls how many Decision Trees will be used within the Random Forest.

Experiment inputting different values for each of the hyperparameters in order to optimise the model. You can also try adding additional hyperparameters listed in the SciKit-Learn Random Forest Classifier documentation.

```
from sklearn.ensemble import RandomForestClassifier
model = RandomForestClassifier()
ccp_alpha_list = [0.001,0.1]
n_estimators_list = [100, 500]
grid = GridSearchCV(estimator = model, param_grid = {'ccp_alpha' :
ccp_alpha_list, 'n_estimators' : n_estimators_list}, cv = 5)
grid.fit(X_train, y_train)
print(grid.best_score_, grid.best_params_)
```

Record your best performing model details in the table below:

Model Name	**Hyperparameters and values**	**Accuracy**

Next Steps

Producing an optimal model which works well on unseen data is a difficult task. In order to get a true estimate of a model's real-world performance we use a hold out test set. This is a portion of our dataset (usually at least 20%) which is completely isolated from the training process, i.e., everything we have been doing so far.

At the start of the exercise, you downloaded a section file called test.csv. We are going to use this file to test our models, however, we cannot do this over and over as this effectively makes our hold out test set part of our training data. Instead, we should only ever evaluate our best performing model on our hold-out test set.

Important the test data using the following code:

```
test_df = pd.read_csv("test.csv")
```

You should now complete the same data processing steps as was carried out on the training data.

Next, you should train your chosen model with the hyperparameters values which were identified as giving the best performance. For example:

```
features = ['Pclass', 'Sex', 'Age', 'SibSp', 'Parch', 'Fare',
'Embarked']
X_test = test_df[features]
model = RandomForestClassifier(n_estimators = 500, ccp_alpha = 0.001)
model.fit(X_train, y_train)
```

Finally, you need to make predictions for each passenger in the hold out test set. We will also be exporting the results into submission.csv.

```
final_pred = model.predict(X_test)

output = pd.DataFrame({'PassengerId': test_df.PassengerId, 'Survived':
final_pred})
output.to_csv('submission.csv', index=False)
```

You can now upload your file and find out how well your model actually performs by clicking here. You will need to sign up to Kaggle if you do not already have an account.

You should continue to explore how to improve your models.
Continue to experiment with different types of models and different hyperparameters.
You should try and gain a better understanding of the data. For example, we have previously discounted the name of a passenger, however, their title "Mr, Mrs, Miss, etc." could be useful. You could also research into automated feature selection methods which would help establish the best data to input into the model.