Classification and Regression: In a Weekend

By

Ajit Jaokar

Dan Howarth

With contributions from

Ayse Mutlu

Contents

Introduction and approach _____ 5

Background _____ 5

Tools _____ 6

Philosophy _____ 8

What you will learn from this book? _____ 9

Components for book _____ 11

Big Picture Diagram _____ 13

Code outline _____ 15

Regression code outline _____ 15

Classification Code Outline _____ 16

Exploratory data analysis _____ 17

Numeric Descriptive statistics _____ 17

Graphical descriptive statistics _____ 19

Analysing the target variable _____ 22

Pre-processing data _____ 23

Dealing with missing values _____ 23

Treatment of categorical values _____ 23

Normalise the data _____ 23

Split the data 27

Choose a Baseline algorithm 29

Defining / instantiating the baseline model 29

Fitting the model we have developed to our training set 29

Define the evaluation metric 30

Predict scores against our test set and assess how good it is 32

Evaluation metrics for classification 33

Improving a model – from baseline models to final models 37

Understanding cross validation 38

Feature engineering 41

Regularization to prevent overfitting 41

Ensembles – typically for classification 43

Test alternative models 45

Hyperparameter tuning 45

Conclusion 47

Appendix 49

Regression Code 49

Classification Code 60

Introduction and approach

Background

This book began as a series of weekend workshops created by Ajit Jaokar and Dan Howarth in the "Data Science for Internet of Things" meetup in London. The idea was to work with a specific (longish) program such that we explore as much of it as possible in one weekend. This book is an attempt to take this idea online. We first experimented on Data Science Central in a small way and continued to expand and learn from our experience. The best way to use this book is to work with the code as much as you can. The code has comments. But you can extend the comments by the concepts explained here.

The code is

Regression
https://colab.research.google.com/drive/14m95e5A3AtzM_3e7IZLs2dd0M4Gr1y1W

Classification
https://colab.research.google.com/drive/1qrj5B5XkI-PkDN-S8XOddlvqOBEggnA9

This document also includes the code in a plain text format in the appendix. The book also includes an online forum where you are free to post questions relating to this book

link of forum

Community for the book

https://www.datasciencecentral.com/group/ai-deep-learning-machine-learning-coding-in-a-week

Finally, the book is part of a series. Future books planned in the same style are

"AI as a service: An introduction through Azure in a weekend"
"AI as a service: An introduction through Google Cloud Platform in a weekend"

Tools

We use Colab from Google. The code should also work on Anaconda. There are four main Python libraries that you should know: numpy, pandas, mathplotlib and sklearn

NumPy

The Python built-in list type does not allow for efficient array manipulation. The NumPy package is concerned with manipulation of multi-dimensional arrays. NumPy is at the foundation of almost all the other packages covering the Data Science aspects of Python.

From a Data Science perspective, collections of Data types like Documents, Images, Sound etc. can be represented as an array of numbers. Hence, the first step in analysing data is to transform data into an array of numbers. NumPy functions are used for transformation and manipulation of data as numbers – especially before the model building stage – but also in the overall process of data science.

Pandas

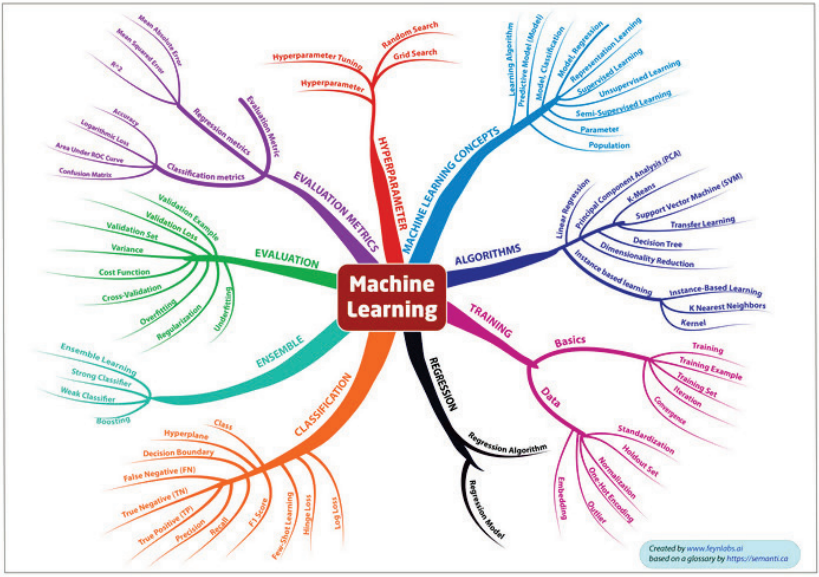
The Pandas library in Python provides two data structures: The **DataFrame and the Series** object. The Pandas Series Object is a one-dimensional array of indexed data which can be created from a list or array. The Pandas DataFrames objects are essentially multidimensional arrays with attached row and column labels. A DataFrame is roughly equivalent to a 'Table' in SQL or a spreadsheet. Through the Pandas library, Python implements a number of powerful data operations similar to database frameworks and spreadsheets. While the NumPy's ndarray data structure provides features for numerical computing tasks, it does not provide flexibility that we see in Tale structures (such as attaching labels to data, working with missing data, etc.). The Pandas library thus provides features for data manipulation tasks.

Matplotlib

The Matplotlib library is used for data visualization in Python built on numpy. Matplotlib works with multiple operating systems and graphics backends.

Scikit-Learn

The Scikit-Learn package provides efficient implementations of a number of common machine learning algorithms. It also includes modules for cross validation, grid search and feature engineering



(original pdf in attached zip)

Philosophy

The book is based on the philosophy of deliberate practise to learn coding. This concept originated in the old Soviet Union athletes. It is also associated with a diverse range of people including Golf (Ben Hogan), Shaolin Monks, Benjamin Franklin etc. For the purposes of learning coding for machine learning, we apply the following elements of deliberate practice

- Break down key ideas in simple, small steps. In this case, using a mindmap and a glossary
- Work with micro steps
- Keep the big picture in mind
- Encourage reflection/feedback

What you will learn from this book?

This book covers regression and classification in an end-to-end mode. We first start with explaining specific elements of regression. Then we move to classification where we cover elements of classification which differ (for example evaluation metrics). We then discuss a set of techniques that help to improve a baseline model for both regression and classification.

Components for book

The book comprises of the following components as part of the online zip

Regression:
https://colab.research.google.com/drive/14m95e5A3AtzM_3e7IZLs2dd0M4Gr1y1W

Classification:
https://colab.research.google.com/drive/1qrj5B5XkI-PkDN-S8XOddlvqOBEggnA9

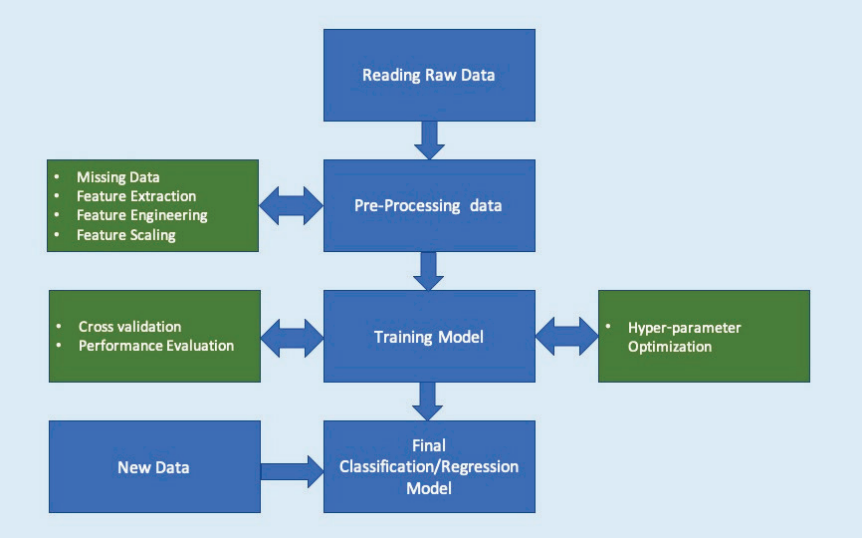
Community for book:
https://www.datasciencecentral.com/group/ai-deep-learning-machine-learning-coding-in-a-week

Glossary: Attached as part of zip also HERE

Mindmap: Attached as part of the zip also HERE

Big Picture Diagram

As below



Code outline

Regression code outline

https://colab.research.google.com/drive/14m95e5A3AtzM_3e7IZLs2dd0M4Gr1y1W

The steps for the code are
Load and describe the data
Exploratory Data Analysis
    Exploratory data analysis – numerical
    Exploratory data analysis - visual
    Analyse the target variable
    compute the correlation
Pre-process the data
    Dealing with missing values
    Treatment of categorical values
    Remove the outliers
    Normalise the data
Split the data
Choose a Baseline algorithm
    defining / instantiating the baseline model
    fitting the model we have developed to our training set
    Define the evaluation metric
    predict scores against our test set and assess how good it is
Refine our dataset with additional columns

Test Alternative Models
Choose the best model and optimise its parameters
    Gridsearch

Classification Code Outline

https://colab.research.google.com/drive/1qrj5B5XkI-PkDN-S8XOddlvqOBEggnA9

```
Load the data
Exploratory data analysis
  Analyse the target variable
  Check if the data is balanced
  Check the co-relations
Split the data
Choose a Baseline algorithm
Train and Test the Model
Choose an evaluation metric
Refine our dataset
Feature engineering
Test Alternative Models
Ensemble models
Choose the best model and optimise its parameters
```

Exploratory data analysis

Numeric Descriptive statistics

Overview

The pandas dataframe structure is a way of storing and operating on tabular data. Pandas has a lot of functionality to assist with exploratory data analysis. describe() provides summary statistics on all numeric columns. describe() function gives descriptive statistics for any numeric columns using describe. For each feature, we can see the `count`, or number of data entries, the `mean` value, and the `standard deviation`, `min`, `max` and `quartile` values. describe() function excludes the character columns. To include both numeric and character columns, we add include='all'. We can also see the shape of the data using the .shape attribute. Keys() method in Python Dictionary, returns a view object that displays a list of all the keys in the dictionary

Numeric descriptive statistics

Standard deviation represents how measurements for a group are spread out from the average (mean). A low standard deviation implies that most of numbers are close to the average. A high standard deviation means that the numbers are spread out. The standard deviation is affected by outliers because the standard deviation is

based on the distance from the mean. The mean is also affected by outliers.

Interpreting descriptive statistics

What actions can you take from the output of the describe function at regression problem?

For each feature, we can see the count, or number of data entries, the mean value, and the standard deviation, min, max and quartile values. We can see that the range of values for each feature differs quite a lot, so we can start to think about whether to apply normalization to the data. We can also see that the CHAS feature is either a (1,0) value. If we look back at our description, we can see that this is an example of a categorical variable. These are values used to describe non-numeric data. In this case, a 1 indicates the house borders near the river, and a 0 that it doesn't.

Source:

- http://www.datasciencemadesimple.com/descriptive-summary-statistics-python-pandas/
- https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.Series.describe.htmlSource
- https://www.dataz.io/display/Public/2013/03/20/Describing+Data%3A+Why+median+and+IQR+are+often+better+than+mean+and+standard+deviation
- https://www.quora.com/What-is-the-relation-between-the-Range-IQR-and-standard-deviation

We can build on this analysis by plotting the distribution and boxplots for each column

Graphical descriptive statistics

Histogram and Boxplots – understanding the distribution

Histograms are used to represent data which is in groups. X-axis represents bin ranges. The Y-axis represents the frequency of the bins. For example, to represent age-wise population in form of graph, then the histogram represents the number of people in age buckets. The bins parameter represents the number of buckets that your data will be divided into. You can specify it as an integer or as a list of bin edges. Interpretation of histograms and box plots and the action taken from it A `histogram` tells is the number of times, or frequency, a value occurs within a `bin`, or bucket, that splits the data (and which we defined). A histogram shows the frequency with which values occur within each of these bins, and can tell us about the distribution of data. A `boxplot` captures within the box the `interquartile range`, the range of values from Q1/25th percentile to Q3/75th percentile, and the median value. It also captures the `min` and `max` values of each feature. Together, these charts show us the distribution of values for each feature. We can start to make judgements about how to treat the data, for example whether we want to deal with outliers; or whether we want to normalize the data. The subplot is a utility wrapper that makes it convenient to create common layouts in a single call.

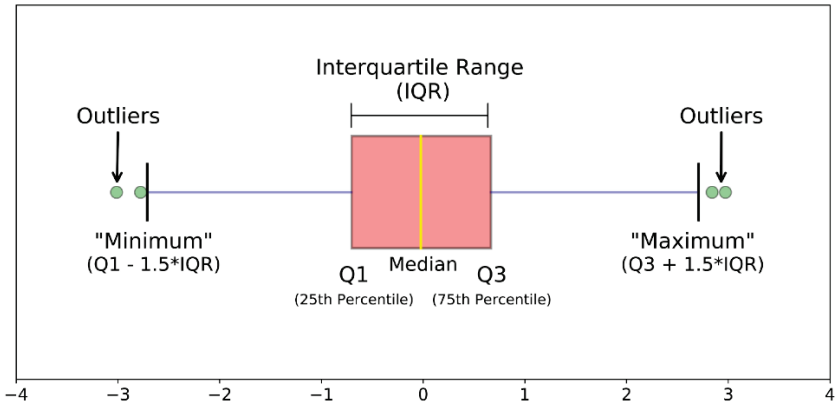
References:

https://matplotlib.org/api/pyplot_api.html#matplotlib.pyplot.subplots

https://towardsdatascience.com/understanding-boxplots-5e2df7bcbd51

Boxplots and IQR

An alternative to mean and standard deviation are median and interquartile range (IQR). IQR is the difference between the third and first quartiles (75th and 25th quantiles). IQR is often reported using the "five-number summary," which includes: minimum, first quartile, median, third quartile and maximum. IQR tells you where the middle 50% of the data is located while Standard Deviation tells you about the spread of the data. Median and IQR measure the central tendency and spread, respectively, but are robust against outliers and non-normal data. IQR makes outlier identification easy to do an initial estimate of outliers by looking at values more than one-and-a-half times the IQR distance below the first quartile or above the third quartile. Skewness: Comparing the median to the quartile values shows whether data is skewed.



https://towardsdatascience.com/understanding-boxplots-5e2df7bcbd51?gi=730efa1b7da5

Correlation

Correlation is a statistical measure that describes the association between random variables. There are several methods for calculating the correlation coefficient, each measuring different types of strength of association. Correlation values range between -1 and 1. Pandas dataframe.corr() gives the pairwise correlation of all columns in the dataframe. Three of the most widely used methods.

1. Pearson Correlation Coefficient
2. Spearman's Correlation
3. Kendall's Tau

Pearson is the most widely used correlation coefficient. Pearson correlation measures the linear association between continuous variables. In other words, this coefficient quantifies the degree to which a relationship between two variables can be described by a line.

$$r = \frac{Covariance\ (x,y)}{S.D.(x)S.D.(y)}$$

$$r = \frac{n\sum xy - (\sum x)(\sum y)}{\sqrt{[n\sum x^2 - (\sum x)^2][n\sum y^2 - (\sum y)^2]}}$$

In this formulation, raw observations are centered by subtracting their means and re-scaled by a measure of standard deviations.

Source:

- https://www.datascience.com/blog/introduction-to-correlation-learn-data-science-tutorials
- https://www.geeksforgeeks.org/python-pandas-dataframe-corr/

heatmaps for co-relation

A heatmap is a two-dimensional graphical representation of data where the individual values are represented as colors. The seaborn python package enables the creation of annotated heatmaps. This heat map works by correlation. This shows you which variables are correlated to each other from a scale of 1 being the most correlated and -1 is not correlated at all. However, you cannot correlate strings. You can only correlate numerical features.

Range from -1 to 1:

- +1.00 means perfect positive relationship (Both variables are moving in the same direction)
- 0.00 means no relationship
- -1.00 means perfect negative relationship (As one variable increases the other decreases)

Source:

- https://seaborn.pydata.org/generated/seaborn.heatmap.html
- https://statisticsbyjim.com/basics/correlations/

Source:

- https://www.datascience.com/blog/introduction-to-correlation-learn-data-science-tutorials

Analysing the target variable

There are a number of ways to analyse the target variable we can plot a histogram using binning to find the grouping of the house prices we can plot a boxplot of the target variable we can do is plot a boxplot of one variable against the target variable we can extend the analysis by creating a heatmap this shows the correlation between the features and target

Pre-processing data

Dealing with missing values

Dealing with missing values, where we identify what, if, any missing data we have and how to deal with it. For example, we may replace missing values with the mean value for that feature, or by the average of the neighbouring values. pandas` has a number of options for filling in missing data that is worth exploring. We can also use `k-nearest neighbour`to help us predict what the missing values should be, or `sklearn Imputer` function (amongst other ways)

Treatment of categorical values

Treat categorical values, by converting them into a numerical representation that can be modelled. There are a number of different ways to do this in `sklearn` and `pandas`

Normalise the data

The terms normalization and standardization are sometimes used interchangeably, but they usually refer to different things. Normalization usually means to scale a variable to have a value between 0 and 1, while standardization transforms data to have a mean of

zero and a standard deviation of 1. (source: statisticshowto). Rescaling data in this way is a common pre-processing task in machine learning because many of algorithms assume that all features are on the same scale, typically 0 to 1 or -1 to 1. We need to rescale the values of numerical feature to be between two values. We have several methods to do that. In skicit learn, the commonly used methods are MinMaxScaler and StandardScaler.

MinMaxScaler: Normalization shrinks the range of the data such that the range is fixed between 0 and 1. It works better for cases in which the standardization might not work so well. If the distribution is not Gaussian or the standard deviation is very small, the min-max scaler works better. Normalization makes training less sensitive to the scale of features, so we can better solve for coefficients.

Normalization is typically done via the following equation:

$$z = \frac{x \ - \min(x)}{\max(x) - \min(x)}$$

The StandardScaler: Standardization is used to transform the data such that it has a mean of 0 and a standard deviation of 1. Specifically, each element in the feature is transformed. The mean and standard deviation are separately calculated for the feature, and the feature is then scaled based on:

$$\frac{x_i - mean(x)}{stdev(x)}$$

Source:

- https://www.statisticshowto.datasciencecentral.com/normalized/
- https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html
- https://datascience.stackexchange.com/questions/12321/difference-between-fit-and-fit-transform-in-scikit-learn-models
- https://medium.com/@zaidalissa/standardization-vs-normalization-da7a3a308c64
- https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.drop.html
- https://docs.scipy.org/doc/numpy/reference/generated/numpy.ravel.html
- https://jovianlin.io/feature-scaling/
- https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html
- https://jovianlin.io/feature-scaling/
- https://scikitlearn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html
- https://scikitlearn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html
- https://datascience.stackexchange.com/questions/12321/difference-between-fit-and-fit-transform-in-scikit-learn-models
- https://medium.com/@zaidalissa/standardization-vs-normalization-da7a3a308c64

Split the data

The original dataset should be split up into training and testing data. **Training:** This data is used to build your model. E.g. finding the optimal coefficients in a Linear Regression model; or using the CART algorithm to create a Decision Tree. **Testing:** This data is used to see how the model performs on unseen data, as it would in a real-world situation. This data should be left completely unseen until you would like to test your model to evaluate performance.

Model Selection contains 4 groups of lists. You can check the links (https://scikit-learn.org/stable/modules/classes.html#module-sklearn.model_selection) for details. **Splitter Classes, Splitter Functions, Hyper-parameter optimizers and Model validation.** The module is mainly used for splitting the dataset. It includes 14 different classes and two functions for that purpose. It also provides some functions for model validation and hyper-parameter optimization.

Source
https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.ShuffleSplit.html#sklearn.model_selection.ShuffleSplit

Choose a Baseline algorithm

Defining / instantiating the baseline model

A baseline is a method that uses heuristics, simple summary statistics, randomness, or machine learning to create predictions for a dataset. You can use these predictions to measure the baseline's performance (e.g., accuracy). This metric will then become what you compare any other machine learning algorithm against. For example, your algorithm may be 75% accurate. You would want your 75% accuracy to be higher than any baseline you have run on the same data.

Source:
https://datascience.stackexchange.com/questions/30912/what-does-baseline-mean-in-the-context-of-machine-learning

Fitting the model we have developed to our training set

Linear models are among the oldest and most interpretable modelling methods. A linear model uses a linear function to map a set of values to a set of normal distributions. Linear models are widely useful because the normal distribution occurs frequently in the

natural world and any continuous function can be approximated well with a straight line over a short distance.

Fitting your model to (i.e. using the fit() method on the training data is the training part of the modelling process. After it is trained, the model can be used to make predictions, with a predict() method call. Model fitting is a procedure that takes three steps:

1. First you need a function that takes in a set of parameters and returns a predicted data set.
2. Second you need an 'error function' that provides a number representing the difference between your data and the model's prediction for any given set of model parameters. This is usually either the sums of squared error (SSE) or maximum likelihood.
3. Third you need to find the parameters that minimize this difference.

Source:
https://courses.washington.edu/matlab1/ModelFitting.html
http://garrettgman.github.io/model-fitting/
Source:
https://courses.washington.edu/matlab1/ModelFitting.html

Define the evaluation metric

The most commonly used metric for regression tasks is RMSE (root-mean-square error). This is defined as the square root of the average squared distance between the actual score and the predicted score:

$$\text{RMSE} = \sqrt{\frac{\sum_i (y_i - \hat{y}_i)^2}{n}}$$

Here, yi denotes the true score for the ith data point, and yi denotes the predicted value. One intuitive way to understand this formula is that it is the Euclidean distance between the vector of the true scores and the vector of the predicted scores, averaged by n, where n is the number of data points.

Mean Squared Error is difference between of the estimated values and what you get as a result. The predicted value is based on some equation and tell what you will expect as an average but the result you get might differ from this prediction which is a slight error from the estimated value. This difference is called MSE. This determines how good is the estimation based on your equation.

Mean Absolute Error is the measure of the difference between the two continuous variables. The MAE is the average vertical distance between each actual value and the line that best matches the data. MAE is also the average horizontal distance between each data point and the best matching line.

R^2 is (coefficient of determination) regression score function. It is also called as coefficient of determination. R^2 gives us a measure of how well the actual outcomes are replicated by the model or the regression line. This is based on the total variation of prediction explained by the model. R^2 is always between 0 and 1 or between 0% to 100%.

Mean squared error	MSE	=	$\frac{1}{n}\sum_{t=1}^{n} e_t^2$
Root mean squared error	RMSE	=	$\sqrt{\frac{1}{n}\sum_{t=1}^{n} e_t^2}$
Mean absolute error	MAE	=	$\frac{1}{n}\sum_{t=1}^{n} \lvert e_t \rvert$
Mean absolute percentage error	MAPE	=	$\frac{100\%}{n}\sum_{t=1}^{n} \lvert \frac{e_t}{y_t} \rvert$

Source:

- https://stats.stackexchange.com/questions/131267/how-to-interpret-error-measures
- https://stats.stackexchange.com/questions/118/why-square-the-difference-instead-of-taking-the-absolute-value-in-standard-devia

Predict scores against our test set and assess how good it is

as above

Evaluation metrics for classification

Previously, we considered evaluation metrics for Regression. In this section, we consider the evaluation metrics for Classification. Evaluating the performance of a machine learning model is a fundamental requirement. Essentially, we are exploring two questions: How can I measure the success of this algorithm and when do I know that I have succeeded i.e. should not improve the algorithm more. Different machine learning algorithms have varying evaluation metrics. We have seen evaluation metrics for regression – we now explore the evaluation metrics for classification

For classification, the most common metric is **Accuracy**. Accuracy simply measures how often the classifier makes the correct prediction. It's the ratio between the number of correct predictions and the total number of predictions

$$\text{accuracy} = \frac{\#\ \text{correct predictions}}{\#\ \text{total data points}}$$

While accuracy is easy to understand, the accuracy metric is not suited for unbalanced classes. Hence, we also need to explore other metrics for classification. A **confusion matrix** is a structure to represent classification and it forms the basis of many classification metrics.

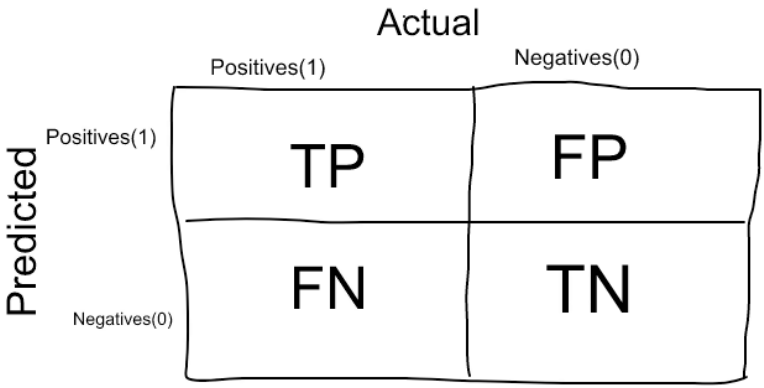


Image source: thalus-ai

There are 4 important terms:
True Positives: The cases in which we predicted YES and the actual output was also YES.
True Negatives: The cases in which we predicted NO and the actual output was NO.
False Positives: The cases in which we predicted YES and the actual output was NO.
False Negatives: The cases in which we predicted NO and the actual output was YES.
Accuracy for the matrix can be calculated by taking average of the values lying across the "main diagonal" i.e.

$$Accuracy = \frac{TruePositives + FalseNegatives}{TotalNumberofSamples}$$

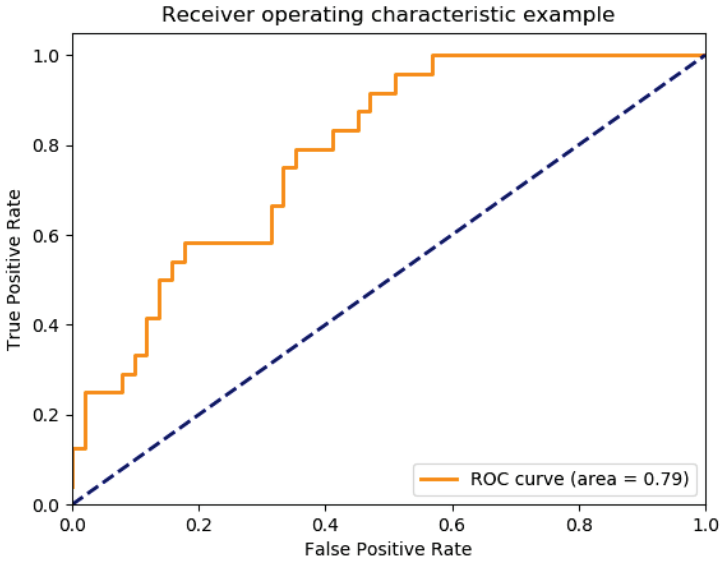
Area Under Curve

One of the widely used metrics for binary classification is the **Area Under Curve(AUC)** AUC represents the probability that the classifier will rank a randomly chosen positive example higher than a randomly chosen negative example. The AUC is based on a plot of the false positive rate vs the true positive rate which are defined as:

$$TruePositiveRate = \frac{TruePositive}{FalseNegative + TruePositive}$$

$$FalsePositiveRate = \frac{FalsePositive}{FalsePositive + TrueNegative}$$

The area under the curve represents the area under the curve when the false positive rate is plotted against the True positive rate as below.



AUC has a range of [0, 1]. The greater the value, the better is the performance of the model because the closer the curve is towards the True positive rate. The AUC shows the correct positive classifications can be gained as a trade-off between more false positives. The advantage of considering the AUC i.e. the area under a curve .. as opposed to the whole curve is that – it is easier to compare the area (a number) with other similar scenarios. Another metric commonly used is Precision-Recall. The Precision metric represents "Out of the items that the classifier predicted to be relevant, how many are truly relevant? The recall answers the question, "Out of all the items that are truly relevant, how many are found by the ranker/classifier?". Similar to the AUC, we need a numeric value to compare similar scenarios. A single number that combines the precision and recall is the F1 score which is represented by the harmonic mean of the precision and recall.

$$F_1 = 2\frac{\text{precision}*\text{recall}}{\text{precision} + \text{recall}}$$

For unbalanced classes and outliers, we need other considerations which are explained HERE

Source:

1. Evaluating machine learning models by Alice Zheng - https://www.oreilly.com/ideas/evaluating-machine-learning-models
2. https://towardsdatascience.com/metrics-to-evaluate-your-machine-learning-algorithm-f10ba6e38234
3. https://medium.com/thalus-ai/performance-metrics-for-classification-problems-in-machine-learning-part-i-b085d432082b

Improving a model – from baseline models to final models

Once we have a baseline model, we can enhance it further. There are a number of steps we could take to achieve this. The baseline model represents the simplest possible prediction. From this point on, you employ a series of techniques to improve the algorithm evaluation metrics. The baseline may be a poor result but it should be seen as a starting point for improvement.

In this document, the strategies we use to improve the baseline model are:

a) Feature engineering – by adding extra columns and trying to understand if it improves the model
b) Regularization to prevent overfitting
c) Ensembles – typically for classification
d) Test alternative models
e) Hyperparameter tuning

References:

https://machinelearningmastery.com/how-to-get-baseline-results-and-why-they-matter/

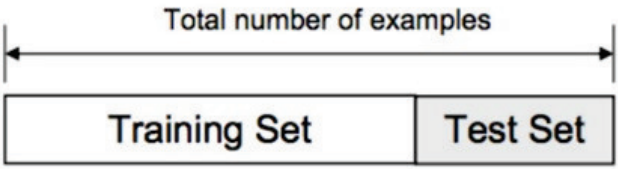
Understanding cross validation

Cross validation is a technique commonly used In Data Science. Most people think that it plays a small part in the data science pipeline, i.e. while training the model. However, it has a broader application in model selection and hyperparameter tuning.

Let us first explore the process of cross validation itself and then see how it applies to different parts of the data science pipeline.

Cross-validation is a resampling procedure used to evaluate machine learning models on a limited data sample. In **k-fold cross-validation**, the original sample is randomly partitioned into **k** equal sized subsamples. Of the **k** subsamples, a single subsample is retained as the **validation** data for testing the model, and the remaining **k** − 1 subsamples are used as training data.

In the model training phase, Cross-validation is primarily used in applied machine learning to estimate the skill of a machine learning model on unseen data to overcome situations like overfitting. The choice of k is usually 5 or 10, but there is no formal rule. Cross validation is implemented through KFold() scikit-learn class. Taken to one extreme, for k = 1, we get a single train/test split is created to evaluate the model. There are also other forms of cross validation ex stratified cross validation



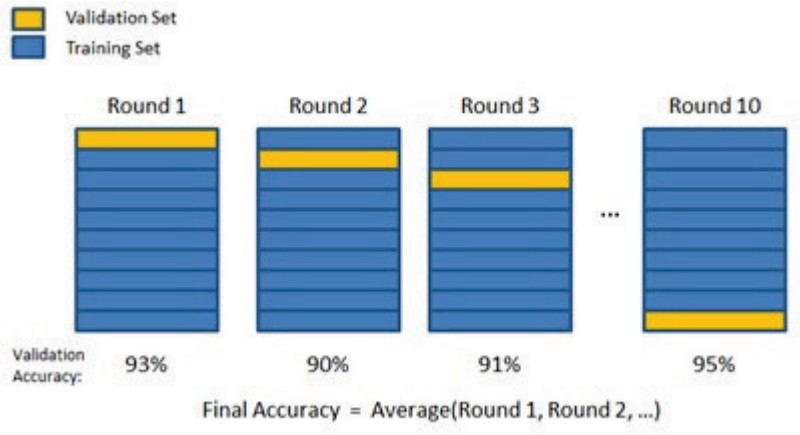


Image source
https://towardsdatascience.com/cross-validation-70289113a072

Now, let's recap the end to end steps for classification based on THIS classification code which we use in the – learn machinelearning coding basics in a weekend

Classification code outline

Load the data
Exploratory data analysis
    Analyse the target variable
    Check if the data is balanced
    Check the co-relations
Split the data
Choose a Baseline algorithm
Train and Test the Model
Choose an evaluation metric
Refine our dataset
Feature engineering
Test Alternative Models

Ensemble models
Choose the best model and optimise its parameters

In this context, we outline below two more cases where we can use cross validation

1. In choice of alternate models and
2. In hyperparameter tuning

we explain these below

1) Choosing alternate models:
If we have two models, and we want to see which one is better, we can use cross validation to compare the two for a given dataset. For the code listed above, this is shown in the following section.

```
"""### Test Alternative Models
logistic = LogisticRegression()
cross_val_score(logistic, X, y, cv=5, scoring="accuracy").mean()
rnd_clf = RandomForestClassifier()
cross_val_score(rnd_clf, X, y, cv=5, scoring="accuracy").mean()
```

2) hyperparameter tuning
Finally, cross validation is also used in hyperparameter tuning
As per cross validation parameter tuning grid search

> *"In machine learning, two tasks are commonly done at the same time in data pipelines: cross validation and (hyper)parameter tuning. Cross validation is the process of training learners using one set of data and testing it using a different set. Parameter tuning is the process to selecting the values for a model's parameters that maximize the accuracy of the model."*

So, to conclude, cross validation is a technique used in multiple parts of the data science pipeline

Feature engineering

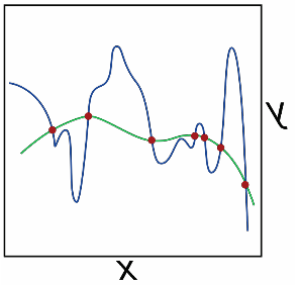
Feature engineering is a key part of the machine learning pipeline. We refine our dataset with additional columns with the objective that some combination of features better represents the problems space and so is an indicator of the target variable. we are using pandas functionality to add a new column called LSTAT_2, which will feature values that are the square of LSTAT values
boston_X['LSTAT_2'] = boston_X['LSTAT'].map(lambda x: x**2)
we can now run the same code as before on our refined dataset to see if things have improved

```
lm.fit(X_train, Y_train)
Y_pred = lm.predict(X_test)
evaluate(Y_test, Y_pred)
```

lambda operator or lambda function is used for creating small, one-time and anonymous function objects in Python.

Regularization to prevent overfitting

In machine learning, regularization is the process of adding information in order to solve an ill-posed problem or to prevent overfitting. Regularization applies to objective functions in ill-posed optimization problems. It can be depicted as below.



https://en.wikipedia.org/wiki/Regularization_(mathematics)

The green and blue functions both incur zero loss on the given data points. A learned model can be induced to prefer the green function, which may generalize better to more points drawn from the underlying unknown distribution, by adjusting lambda, the weight of the regularization term.

A regularization term *R(f)* is added to a loss function:

$$\min_f \sum_{i=1}^{n} V(f(x_i), y_i) + \lambda R(f)$$

where V is an underlying loss function that describes the cost of predicting f(x) when the label is y.$^{\lambda}$ is a parameter which controls the importance of the regularization term. The regularization function is typically chosen to impose a penalty on the complexity of f(x). A theoretical justification for regularization is that it attempts to impose Occam's razor (i.e. the simplest feasible solution) (as depicted in the figure above, where the green function, the simpler one, may be preferred). From a Bayesian point of view, many regularization techniques correspond to imposing certain prior distributions on model parameters. Regularization helps in avoiding overfitting and also increasing model interpretability.

Regularization, significantly reduces the variance of the model, without substantial increase in its bias. So, the tuning parameter λ, used in the regularization techniques described above, controls the impact on bias and variance. As the value of λ rises, it reduces the value of coefficients and thus reducing the variance.

Regularization is particularly important in deep learning where we have a large number of parameters to optimise. Ian Goodfellow

describes regularization as "any modification we make to the learning algorithm that is intended to reduce the generalization error, but not its training error".

Generalization in machine learning refers to how well the concepts learned by the model apply to examples which were not seen during training. The goal of most machine learning models is to generalize well from the training data, in order to make good predictions in the future for unseen data. Overfitting happens when the models learns too well the details and the noise from training data, but it doesn't generalize well, so the performance is poor for testing data. A number of regularization techniques are used in deep learning including Dataset augmentation, Early stopping, Dropout layer, Weight penalty L1 and L2

Sources:
https://towardsdatascience.com/regularization-an-important-concept-in-machine-learning-5891628907ea
https://chatbotslife.com/regularization-in-deep-learning-f649a45d6e0
https://en.wikipedia.org/wiki/Regularization_(mathematics)

Ensembles – typically for classification

What is ensemble learning?

Ensemble learning is a machine learning paradigm where multiple models (often called "weak learners") are trained to solve the same problem and combined to get better results. The main hypothesis is that when weak models are correctly combined we can obtain more accurate and/or robust models.

Then, the idea of ensemble methods is to try reducing bias and/or variance of such weak learners by combining several of them together in order to create a strong learner (or ensemble model) that achieves better performances.

In order to set up an ensemble learning method, we first need to select our base models to be aggregated. One important point is that our choice of weak learners should be coherent with the way we aggregate these models. If we choose base models with low bias but high variance, it should be with an aggregating method that tends to reduce variance whereas if we choose base models with low variance but high bias, it should be with an aggregating method that tends to reduce bias.

There are three major kinds of meta-algorithms that aims at combining weak learners:

bagging, that often considers homogeneous weak learners, learns them independently from each other in parallel and combines them following some kind of deterministic averaging process

boosting, that often considers homogeneous weak learners, learns them sequentially in an adaptive way (a base model depends on the previous ones) and combines them following a deterministic strategy

stacking, that often considers heterogeneous weak learners, learns them in parallel and combines them by training a meta-model to output a prediction based on the different weak models predictions

Weak learners can be combined to get a model with better performances. The way to combine base models should be adapted to

their types. Low bias and high variance weak models should be combined in a way that makes the strong model more robust whereas low variance and high bias base models better be combined in a way that makes the ensemble model less biased.

Source:
https://towardsdatascience.com/ensemble-methods-bagging-boosting-and-stacking-c9214a10a205

Test alternative models

As per the section on cross validation, if we have two models, and we want to see which one is better, we can use cross validation to compare the two for a given dataset. For the code listed above, this is shown in the following section.

```
"""### Test Alternative Models
logistic = LogisticRegression()
cross_val_score(logistic, X, y, cv=5,
scoring="accuracy").mean()
rnd_clf = RandomForestClassifier()
cross_val_score(rnd_clf, X, y, cv=5,
scoring="accuracy").mean()
```

Hyperparameter tuning

In this section, we introduce Hyperparameters and how they determine a model's performance. The process of learning Parameters involves taking the input data and using a function to generate a model. In this case, the model parameters tell how to transform input data into desired output whereas, the hyperparameters of the model are used to determine the structure of the model itself. The performance of the model depends heavily on the hyperparameter

values selected. The goal of hyperparameter tuning is to search across various hyperparameter configurations to find a configuration that will result in the best performance. Hyperparameters help answer questions like: the depth of the decision tree or how many layers should a neural network have etc. There are mainly three methods to perform hyperparameter tuning: Grid search, Random search and Bayesian optimisation

Source:
https://www.analyticsindiamag.com/what-are-hyperparameters-and-how-do-they-determine-a-models-performance/
https://towardsdatascience.com/hyperparameter-tuning-the-random-forest-in-python-using-scikit-learn-28d2aa77dd74

Conclusion

We hope you have learnt from this book. Please post your comments at Community for the book:

https://www.datasciencecentral.com/group/ai-deep-learning-machine-learning-coding-in-a-week

Appendix

In this section, the code is provided in a python text format based on a .py file exported and simplified from colab

Regression Code

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

# rather than importing the whole sklearn library,
we will import certain modules
from sklearn.linear_model import LinearRegression
from sklearn.ensemble import RandomForestRegressor
from sklearn.datasets import load_boston
from sklearn import model_selection
from sklearn.preprocessing import StandardScaler
from sklearn import metrics

# we load the dataset and save it as the variable
boston
boston = load_boston()

# if we want to know what sort of detail is
provided with this dataset, we can call .keys()
boston.keys()

# the info at the .DESCR key will tell us more
print(boston.DESCR)
```

```
# we can use pandas to create a dataframe, which is
basically a way of storing and operating on tabular
data
# here we pass in both the data and the column
names as variables
boston_X = pd.DataFrame(boston.data, columns =
boston.feature_names)

# we can then look at the top of the dataframe to
see the sort of values it contains
boston_X.head()

# pandas has a lot of functionality to assist with
exploratory data analysis
# .describe() provide summary statistics on all
numeric columns
print(boston_X.describe())

# we can also see the shape of the data
print(boston_X.shape)

"""* For each feature, we can see the `count`, or
number of data entries, the `mean` value, and the
`standard deviation`, `min`, `max` and `quartile`
values.
* We can see that the range of values for each
feature differs quite a lot, so we can start to
think about whether to apply normalization to the
data.
* We can also see that the `CHAS` faeture is either
a `(1,0)` value. If we look back at our
description, we can see that this is an example of
a `categorical` variable. These are values used to
describe non-numeric data. In this case, a `1`
indicates the house borders near the river, and a
`0` that it doesn't.
"""

# we can build on this analysis by plotting the
distribution and boxplots for each column

# we loop through all the columns
for col in boston_X.columns:
```

```
    # and for each column we create space for one
row with 2 charts
    f, axes = plt.subplots(1, 2, figsize=(12, 6))
    # our first chart is a histogram and we set the
title
    boston_X[col].hist(bins = 30, ax = axes[0])
    axes[0].set_title('Distribution of '+ col)
    # our second column is the boxplot
    boston_X.boxplot(column = col, ax = axes[1])
    # we then use this to command to display the
charts
    plt.show()

"""* A `histogram` tells is the number of times, or
frequency, a value occurs within a `bin`, or
bucket, that splits the data (and which we
defined). A histogram shows the frequency with
which values occur within each of these bins, and
can tell us about the distribution of data.
* A `boxplot` captures within the box the
`interquartile range`, the range of values from
Q1/25th percentile to Q3/75th percentile, and the
median value. It also captures the `min` and `max`
values of each feature.
* Together, these charts show us the distribution
of values for each feature. We can start to make
judgements about how to treat the data, for example
whether we want to deal with outliers; or whether
we want to normalize the data.
"""

# we can now look at our target variable
boston_y = boston.target

# we can plot a histogram in a slightly different
way
plt.hist(boston_y, bins = 40)
plt.title('Housing price distribution, $K')
plt.show()

# and the same for the boxplot
plt.boxplot(boston_y)
plt.title('Box plot for housing price.')
plt.show()
```

```
# another thing we can do is plot a boxplot of one
variable against the target variable
# it is interesting to see how house value
distribution differs by CHAS, the categorical
variable

# here we create a grouped dataframe that includes
the target variable
grouped_df = boston_X.copy()   # note we create a
copy of the data here so that any changes don't
impact the original data
grouped_df['target'] = boston_y.copy()

# we then plot it here
f, axes = plt.subplots(1, 1, figsize=(10, 5))
grouped_df.boxplot(column='target', by = 'CHAS',
ax = axes)
plt.show()

"""* The `interquartile range`for houses next to
the river is higher than for those houses not next
to the river, and the `min` and `max` values differ
too.
* This suggests this could be an important variable
for us to include in our model, given that as it
differs, the target value distribution changes.
"""

# we can extend this sort of analysis by creating a
heatmap
# this shows the correlation between the features
and target

# first we compute the correlation
corr = grouped_df.corr(method='pearson')
# and plot our figure size
plt.figure(figsize = (15, 10))
# and use seaborn to fill this figure with a
heatmap
sns.heatmap(corr, annot = True)
```

```
"""* We will let you review this heatmap to see
what features are important for modelling and
why."""

# OPTIONAL: below is code that generate a pairplot
using seaborn
# look up what a pairplot is and see if you can
interpret the output of the code below

#sns.pairplot(grouped_df)

"""#### Preprocess the data
* We proprocess the data to ensure it is a suitable
state for modelling. The sort of things that we do
to preprocess the data includes:
  * *Dealing with missing values*, where we
identify what, if, any missing data we have and how
to deal with it. For example, we may replace
missing values with the mean value for that
feature, or by the average of the neighbouring
values.
    * `pandas` has a number of options for filling
in missing data that is worth exploring
    * We can also use `k-nearest neighbour`to help
us predict what the missing values should be, or
`sklearn Imputer` function (amongst other ways)
  * *Treat categorical values*, by converting them
into a numerical representation that can be
modelled.
    * There are a number of different ways to do
this in `sklearn` and `pandas`
  * *Normalise the data*, for example by ensuring
the data is, for example all on the scale (such as
within two defined values); normally distributed;
has a zero-mean, etc. This is sometimes necessary
for the ML models to work, and can also help speed
up the time it takes for the models to run.
    * Again, `sklearn` and `pandas` have in-built
functions to help you do this.
* In this notebook, we will look to remove
`outliers`, which are values that might be
erroneous and which can over-influence the model,
and `normalize` the data
"""
```

```
# lets start by removing outliers

# here we define the columns where we have
identified there could be outliers
numeric_columns = ['CRIM', 'ZN', 'INDUS', 'NOX',
'RM', 'AGE', 'DIS', 'RAD', 'TAX', 'PTRATIO', 'B',
'LSTAT']

# this function can be used on any dataset to
return a list of index values for the outliers
def get_outliers(data, columns):
    # we create an empty list
    outlier_idxs = []
    for col in columns:
        elements = data[col]
        # we get the mean value for each column
        mean = elements.mean()
        # and the standard deviation of the column
        sd = elements.std()
        # we then get the index values of all
values higher or lower than the mean +/- 2 standard
deviations
        outliers_mask = data[(data[col] > mean +
3*sd) | (data[col]  < mean  - 3*sd)].index
        # and add those values to our list
        outlier_idxs  += [x for x in outliers_mask]
    return list(set(outlier_idxs))

# we call the function we just created on the
boston dataset
boston_outliers = get_outliers(boston_X,
numeric_columns)

# and drop those values from our feature and target
values
boston_X = boston_X.drop(boston_outliers, axis = 0)
boston_y =
pd.DataFrame(boston_y).drop(boston_outliers, axis =
0).values.ravel()

# we can check that this code has worked by looking
at the shape of our data
print (boston_X.shape)
```

```
print (boston_y.shape)

# we can also create a function to normalize our
data
# first lets look at the data before normalisation
boston_X[0:10]

# this function loops through columns in a data set
and defines a predefined scaler to each
def scale_numeric(data, numeric_columns, scaler):
    for col in numeric_columns:
        data[col] =
scaler.fit_transform(data[col].values.reshape(-1,
1))
    return data

# we can now define the scaler we want to use and
apply it to our dataset

# a good exercise would be to research waht
StandardScaler does - it is from the scikit learn
library
scaler = StandardScaler()
boston_X = scale_numeric(boston_X, numeric_columns,
scaler)

# here we can see the result
boston_X[0:10]

"""### : Split the data
* In order to train our model and see how well it
performs, we need to split our data into training
and testing sets.
* We can then train our model on the training set,
and test how well it has generalised to the data on
the test set.
* There are a number of options for how we can
split the data, and for what proportion of our
original data we set aside for the test set.
"""

# a common way for splitting our dataset is using
train_test_split
```

```
# as an exercise, go to the scikit learn
documentation to learn more about this function and
the parameters available
X_train, X_test, Y_train, Y_test =
model_selection.train_test_split(boston_X,
boston_y, test_size = 0.2, random_state = 5)

# get shape of test and training sets
print('Training Set:')
print('Number of datapoints: ', X_train.shape[0])
print('Number of features: ', X_train.shape[1])
print('\n')
print('Test Set:')
print('Number of datapoints: ', X_test.shape[0])
print('Number of features: ', X_test.shape[1])

"""### Choose a Baseline algorithm
# linear regression is a fairly simple algorithm
compared to more complicate regression options, so
provides a good baseline
lm = LinearRegression()

"""### Train and Test the Model"""

# fitting the model to the data means to train our
model on the data
# the fit function takes both the X and y variables
of the training data
lm.fit(X_train, Y_train)

# from this, we can generate a set of predictions
on our unseen features, X_test
Y_pred = lm.predict(X_test)

"""### : Choose an evaluation metric
* We then need to compare these predictions with
the actual result and measure them in some way.
* This is where the selection of evaluation metric
is important. For regression, we measure the
distance between the predicted and actual answers
in some way. The shorter the distance, the more
correct the model is.
* We cover three common metrics below:
```

```
  * `Mean Absolute Error`: which provides a mean
score for all the predicted versus actual values as
an absolute value
  * `Means Squared Error`: which provides a mean
score for all the predicted versus actual values as
a square of the absolute value
  * `R2`: which we recommend you research as an
exercise to grow your knowledge. WIkipedia and
`sklearn` document are a great place to start!
"""

def evaluate(Y_test, Y_pred):
    # this block of code returns all the metrics we
are interested in
    mse = metrics.mean_squared_error(Y_test,
Y_pred)
    msa = metrics.mean_absolute_error(Y_test,
Y_pred)
    r2 = metrics.r2_score(Y_test, Y_pred)

    print("Mean squared error: ", mse)
    print("Mean absolute error: ", msa)
    print("R^2 : ", r2)

    # this creates a chart plotting predicted and
actual
    plt.scatter(Y_test, Y_pred)
    plt.xlabel("Prices: $Y_i$")
    plt.ylabel("Predicted prices: $\hat{Y}_i$")
    plt.title("Prices vs Predicted prices: $Y_i$ vs
$\hat{Y}_i$")

evaluate(Y_test, Y_pred)

# we can explore how metrics are dervied in a
little more detail by looking at MAE
# here we will implement MAE using numpy, building
it up step by step

# with MAE, we get the absolute values of the error
- as you can see this is of the difference between
the actual and predicted values
np.abs(Y_test - Y_pred)
```

```
# we will then sum them up
np.sum(np.abs(Y_test - Y_pred))

# then divide by the total number of
predictions/actual values
# as you will see, we get to the same score
implemented above
np.sum(np.abs(Y_test - Y_pred))/len(Y_test)

"""### : Refine our dataset
* This step allows us to add or modify features of
the datatset. We might do this if, for example,
some combination of features better represents the
problems space and so is an indicator of the target
variable.
* Here, we create one additional feature as an
example, but you should reflect on our EDA earlier
and see whether there are other features that can
be added to our dataset.
"""

# here we are using pandas functionality to add a
new column called LSTAT_2, which will feature
values that are the square of LSTAT values
boston_X['LSTAT_2'] = boston_X['LSTAT'].map(lambda
x: x**2)

# we can run our train_test_split function and see
that we have an additional features
X_train, X_test, Y_train, Y_test =
model_selection.train_test_split(boston_X,
boston_y, test_size = 0.2, random_state = 5)

print('Number of features after dataset refinement:
', X_train.shape[1])

# we can now run the same code as before on our
refined dataset to see if things have improved
lm.fit(X_train, Y_train)

Y_pred = lm.predict(X_test)

evaluate(Y_test, Y_pred)
```

```
"""### Step 8: Test Alternative Models
* Once we got a nice baseline model working for
this dataset, we also can try something more
sophisticated and rather different, e.g.
RandomForest Regressor. So, let's do so and also
evaluate the result.
"""

# as you can see, its very similar code to
instantiate the model
# we are able to pass in additional parameters as
the model is created, so optionally you can view
the documentation and play with these values

rfr = RandomForestRegressor()
rfr.fit(X_train, Y_train)
Y_pred = rfr.predict(X_test)

evaluate(Y_test, Y_pred)

"""### : Choose the best model and optimise its
parameters
* We can see that we have improved our model as we
have added features and trained new models.
* At the point that we feel comfortable with a good
model, we can start to tune the parameters of the
model.
* There are a number of ways to do this, and a
common way is shown below
"""

## grid search is a 'brute force' search, one that
will explore every possible combination of
parameters that you provide it

# we first define the parameters we want to search
as a dictionary. Explore the documentation to what
other options are avaiable
params = {'n_estimators': [100, 200], 'max_depth' :
[2, 10, 20]}

# we then create a grid search object with our
chosen model and paramters. We also use cross
validation here - explored more in Day 2
```

```
grid = model_selection.GridSearchCV(rfr, params,
cv=5)

# we fit our model to the data as before
grid.fit(X_train, Y_train)

# one output of the grid search function is that we
can get the best_estimator - the model and
parameters that scored best on the training data -
#  and save it as a new a model
best_model = grid.best_estimator_

# and use it to predict and evaluate as before
Y_pred = best_model.predict(X_test)

evaluate(Y_test, Y_pred)
```

Classification Code

```
# -*- coding: utf-8 -*-

# import main data analysis libraries
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
# note we use scipy for generating a uniform
distribution in the model optimization step
from scipy.stats import uniform

# note that because of the different dataset and
algorithms, we use different sklearn libraries from
Day 1
from sklearn.datasets import load_breast_cancer
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import VotingClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import
train_test_split
from sklearn.model_selection import
RandomizedSearchCV
```

```
from sklearn.model_selection import cross_val_score
from sklearn.dummy import DummyClassifier
from sklearn import metrics
from sklearn.metrics import accuracy_score
from sklearn.svm import SVC

# hide warnings
import warnings
warnings.filterwarnings('ignore')

# we load the dataset and save it as the variable
data
data = load_breast_cancer()

# if we want to know what sort of detail is
provided with this dataset, we can call .keys()
data.keys()

# the info at the .DESCR key will tell us more
print (data.DESCR)

#### Analyze the Data
X = pd.DataFrame(data.data, columns =
data.feature_names)

# we can then look at the top of the dataframe to
see the sort of values it contains
X.describe(include = 'all')

# we can now look at our target variable
y = data.target

# we can see that it is a list of 0s and 1s, with
1s matching to the Benign class y

# we can analyse the data in more detail by
understanding how the features and target variables
interact
# we can do this by grouping the features and the
target into the same dataframe
# note we create a copy of the data here so that
any changes don't impact the original data

full_dataset = X.copy()
```

```
full_dataset['target'] = y.copy()

# let's take a look at the first few lines of the
dataset
full_dataset.head()

# lets see how balanced the classes are (and if
that matches to our expectation)
full_dataset['target'].value_counts()

# let's evaluate visually how well our classes are
differentiable on the pairplots
# can see two classes being present on a two
variables charts?
# the pairplot function is an excellent way of
seeing how variables inter-relate, but 30 feature
can make studying each combination difficult!
sns.pairplot(full_dataset, hue = 'target')

"""* We can clearly see the presence of two clouds
with different colors, representing our target
classes.
* Of course, they are still mixed to some extent,
but if we were to visualise the variables in multi-
dimentional space they would become more separable.
* Now let's check the Pearson's correlation between
pairs of our features and also between the features
and our target.
"""

# we can again use seaborn to easily create a
visually interesting chart
plt.figure(figsize = (15, 10))

# we can add the annot=True parameter to the
sns.heatmap arguments if we want to show the
correlation values
sns.heatmap(full_dataset.corr(method='pearson'))

"""* Dark red colours are positilvey correlated
with the corresponding feature, dark blue features
are negatively correlated.
* We can see that some values are negatively
correlated with our target variable.
```

```
* This information could help us with feature
engineering.

###  Split the data
* In order to train our model and see how well it
performs, we need to split our data into training
and testing sets.
* We can then train our model on the training set,
and test how well it has generalised to the data on
the test set.
* There are a number of options for how we can
split the data, and for what proportion of our
original data we set aside for the test set.
"""

# Because our classes are not absolutely equal in
number, we can apply stratification to the split
# and be sure that the ratio of the classes in both
train and test will be the same
# you can learn about the other parameters by
looking at the documentation
X_train, X_test, y_train, y_test =
train_test_split(X, y, test_size = 0.2, stratify =
y, shuffle=True)

# as with Day 1, we can get shape of test and
training sets
print('Training Set:')
print('Number of datapoints: ', X_train.shape[0])
print('Number of features: ', X_train.shape[1])
print('\n')
print('Test Set:')
print('Number of datapoints: ', X_test.shape[0])
print('Number of features: ', X_test.shape[1])

# and we can verify the stratifications using
np.bincount
print('Labels counts in y:', np.bincount(y))
print('Percentage of class zeroes in
class_y',np.round(np.bincount(y)[0]/len(y)*100))

print("\n")
print('Labels counts in y_train:',
np.bincount(y_train))
```

```
print('Percentage of class zeroes in
y_train',np.round(np.bincount(y_train)[0]/len(y_tra
in) *100))

print("\n")
print('Labels counts in y_test:',
np.bincount(y_test))
print('Percentage of class zeroes in
y_test',np.round(np.bincount(y_test)[0]/len(y_test)
*100))

"""### : Choose a Baseline algorithm
* Building a model in `sklearn` involves:
## we can create a baseline model to benchmark our
other estimators against
## this can be a simple estimator or we can use a
dummy estimator to make predictions in a random
manner
# this creates our dummy classifier, and the value
we pass in to the strategy parameter dtermn
dummy = DummyClassifier(strategy='uniform',
random_state=1)

"""### : Train and Test the Model"""
# "Train" model
dummy.fit(X_train, y_train)

# from this, we can generate a set of predictions
on our unseen features, X_test
dummy_predictions = dummy.predict(X_test)

"""### : Choose an evaluation metric
* We then need to compare these predictions with
the actual result and measure them in some way.
This is where the selection of evaluation metric is
important.
* Classification metrics include:
  * `accuracy`: this assess how often the model
selects the best class. This can be more useful
when there are balanced classes (i.e. there are a
similar number of instances of each class we are
trying to predict).
    * There are some limits to this metric. For
example, if we think about something like credit
```

```
card fraud, where the instances of fraudulent
transactions might be 0.5%, then a model that
*always* predicts that a transaction is not
fraudulent will be 99.5% accurate! So we often need
metrics that can tell us how a model performs in
more detail.
  * `f1 score`:
  * `roc_auc`:
  * `recall`:
  * We recommend you research these metrics to
improve your understanding of how they work. Try to
look up an explanation or two (for example on
wikipedia and scikit-learn documentation) and write
a one line summary in the space provided above.
Then, below, when we implement a scoring function,
select these different metrics and try to explain
what is happening. This will help cement you
knowledge.
"""

|def evaluate(y_test, y_pred):
    # this block of code returns all the metrics we
are interested in
    accuracy = metrics.accuracy_score(y_test,
y_pred)
    f1 = metrics.f1_score(y_test, y_pred)
    auc = metrics.roc_auc_score(y_test, y_pred)

    print ("Accuracy", accuracy)
    print ('F1 score: ', f1)
    print ('ROC_AUC: ' , auc)

# we can call the function on the actual results
versus the predictions
# we will see that the metrics are what we'd expect
from a random model
evaluate(y_test, dummy_predictions)

"""### Test Alternative Models

## here we fit a new estimator and use
cross_val_score to get a score based on a defined
metric
```

```
# instantiate logistic regression classifier
logistic = LogisticRegression()

# we pass our estimator and data to the method. we
also specify the number of folds (default is 3)
# the default scoring method is the one associated
with the estimator we pass in
# we can use the scoring parameter to pass in
different scoring methods. Here we use f1.
cross_val_score(logistic, X, y, cv=5, scoring="f1")

# we can see that this returns a score for all the
five folds of the cross_validation
# if we want to return a mean, we can store as a
variable and calculate the mean, or do it directly
on the function
# this time we will use accuracy
cross_val_score(logistic, X, y, cv=5,
scoring="accuracy").mean()

# lets do this again with a different model
rnd_clf = RandomForestClassifier()

# and pass that in
cross_val_score(rnd_clf, X, y, cv=5,
scoring="accuracy").mean()



"""#### Ensemble models

* Let's take this opportunity to explore ensemble
methods.
* The goal of ensemble methods is to combine
different classifiers into a meta-classifier that
has better generalization performance than each
individual classifier alone.
* There are several different approaches to achieve
this, including **majority voting** ensemble
methods, which we select the class label that has
been predicted by the majority of classifiers.
* The ensemble can be built from different
classification algorithms, such as decision trees,
support vector machines, logistic regression
```

```
classifiers, and so on. Alternatively, we can also
use the same base classification algorithm, fitting
different subsets of the training set.
* Indeed, Majority voting will work best if the
classifiers used are different from each other
and/or trained on different datasets (or subsets of
the same data) in order for their errors to be
uncorrelated.
"""

# lets instantiate an additional model to make an
ensemble of three models
dt_clf = DecisionTreeClassifier()

# and an ensemble of them
voting_clf = VotingClassifier(estimators=[('lr',
logistic), ('rf', rnd_clf), ('dc', dt_clf)],
                              # here we select soft
voting, which returns the argmax of the sum of
predicted probabilities
                              voting='soft')

# here we can cycle through the individual
estimators
# for clf in (log_clf, rnd_clf, svm_clf,
voting_clf):

for clf in (log_clf, rnd_clf, dt_clf, voting_clf):

    # fit them to the training data
    clf.fit(X_train, y_train)

    # get a prediction
    y_pred = clf.predict(X_test)

    # and print the prediction
    print(clf.__class__.__name__,
accuracy_score(y_test, y_pred))

"""* We can see that `voting classifier` in this
the case does have a slight edge on the other
models (note that this could vary depending on how
the data is split at training time).
```

```
* This is an interesting approach and one to
consider when you are developing your models.

### Step 9: Choose the best model and optimise its
parameters
* We can see that we have improved our model as we
have added features and trained new models.
* At the point that we feel comfortable with a good
model, we can start to tune the parameters of the
model.
* There are a number of ways to do this. We applied
`GridSearchCV` to identify the best hyperparameters
for our models on Day 1.
* There are other methods available to use that
don't take the brute force approach of
`GridSearchCV`.
* We will cover an implementation of
`RandomizedSearchCV` below, and use the exercise
for you to implement it on the other datatset.
* We use this method to search over defined
hyperparameters, like `GridSearchCV`, however a
fixed number of parameters are sampled, as defined
by `n_iter` parameter.
"""

# we will optimise logistics regression
# we can create hyperparameters as a list, as in
type regularization penalty
penalty = ['l1', 'l2']

# or as a distribution of values to sample from -
'C' is the hyperparameter controlling the size of
the regularisation penelty
C = uniform(loc=0, scale=4)

# we need to pass these parameters as a dictionary
of {param_name: values}
hyperparameters = dict(C=C, penalty=penalty)

# we instantiate our model
randomizedsearch = RandomizedSearchCV(logistic,
hyperparameters, random_state=1,
            n_iter=100, cv=5, verbose=0, n_jobs=-1)
```

```
# and fit it to the data
best_model = randomizedsearch.fit(X, y)

# from this, we can generate a set of predictions
on our unseen features, X_test
best_predictions = best_model.predict(X_test)

# and evaluate model performance
evaluate(y_test, best_predictions)

# and we can call this method to return the best
parameters the search returned
best_model.best_estimator_

# and - we can evaluate the model using the cross
validation method discussed above
cross_val_score(best_model, X, y, cv=5,
scoring="accuracy").mean()

"""* evaluation of the scores
```