1. The Elements of Machine Learning

Gholamreza Haffari

1. The Elements of Machine Learning

Gholamreza Haffari

Contents

1 An Introduction to Machine Learning

The general problem of *Statistical Learning* is *learning* a *functional* relationship between a set of *attribute (or input) variables* and the associated *response or target variables*. The purpose of this process can be either *prediction* or *inference.* For prediction, which is the main focus of *Machine Learning*, the learned model is used to *predict* the target/response value for any (possibly new) values of the attribute variables. Whereas in inference, the models used to *understand* the way that the target variable is affected as the input variables change.

For example, consider Figure **1.1.1** which shows the gold medal winning time for the men's 100m at each of the Olympics Games held since 1896. Our aim is to use this data to *learn* a model of the functional dependence (if one exists) between Olympics year and 100m winning time, and use this model to make *predictions* about the winning times in *future* games.

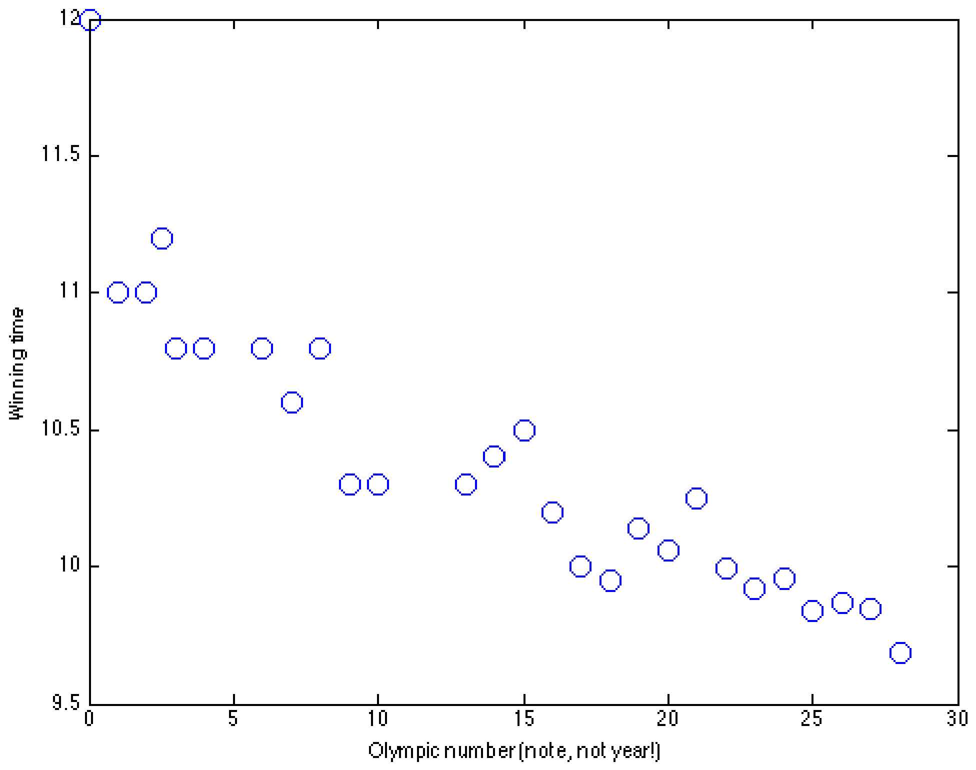


Figure 1.1.1: Winning men's 100m times at the Summer Olympics since 1896.

Clearly, the year is not the only factor that affects the winning time, but by examining Figure **1.1.1**, we can see that there is, at least, a *statistical dependence* between year and winning time (it may not be a *casual dependence* though - elapsing years are not directly causing the drop in winning times).

There are many *functions* that can be used to define the mapping between the year x (input variable) and the winning time t (target variable); mathematically, we write this as $t = f(x)$. For instance, we may consider a degree M polynomial as our function (or model):

$$f(x) := w_0 + w_1 x + w_2 x^2 + \ldots + w_M x^M = \sum_{j=0}^{M} w_j x^j$$

where x^j denotes x raised to the power of j, and $\mathbf{w} := (w_0, \ldots, w_M)$ are the model parameters which need to be *determined somehow*. In machine learning, we *learn* the model parameters from a suitably given dataset, which is called the *training set*. For our Olympics problem, we learn the parameters based on the training data that we have collected for the past Olympic years $\{x_1, \ldots, x_N\}$ and the corresponding winning times for men's 100m.

Once the model is *trained* (e.g. its parameters are learned), it can then predict the winning time for new Olympic years, which are said to comprise a *test set*. The ability to predict accurately the target for new examples that differ from those used in the training set is known as *generalization. The holy grail of machine learning is to build models which can generalize well to unseen examples.*

Applications in which the training data comprises examples of the input variable along with their corresponding target variable are known as *supervised learning* problems. Cases such as the Olympic winning time prediction, where the desired output is *real-valued* and *continuous*, is called *regression*. If the desired output consists of *a finite number of discrete categories*, then the task is called *classification*. An example of a classification task is to predict whether a given image includes a human face or not, i.e. the target values are 'YES' or 'NO'.

In other machine learning problems, the training data consists of a set of inputs without any corresponding target value. This is then called *unsupervised learning* problems. The goal in such problems may be to discover groups of similar examples within the data, where is called *clustering*, or to project the data from high-dimensional space down to two or three dimensions for the purpose of *visualization*.

2 The Fundamental Concepts

The Setup

We begin by introducing a simple regression problem, which we use as a running example to introduce the key concepts. Suppose we have observed a real-valued input variable x and we wish to use this observation to predict the value of a real-valued target variable t. For now, we consider an artificial example using syntactically generated data because we then know the precise process that generated the data for comparison against any learned model. The data for this example is generated from the function $sin(2\pi x)$ with random noise included in the target values.

Training Set. Now suppose that we are given a *training set* comprising N data points $\{(x_1, t_1), \dots, (x_N, t_N)\}$ where each tuple (x_n, t_n) consists of the input observation x_n and its corresponding target value t_n. Figure **1.2.1** shows a plot of a training set comprising $N = 10$ data points, where we have placed x_n uniformly in the range $[0, 1]$. The corresponding target value t_n for each x_n was obtained by first computing the corresponding value of the function $sin(2\pi x_n)$ and then adding a small level of random noise. By generating data in this way, we are capturing a property of many real world learning problems: The measurements are corrupted by noise but still we would like to recover (or learn) the *unknown* underlying functional relationship between the input and output variables based on our noisy training set.

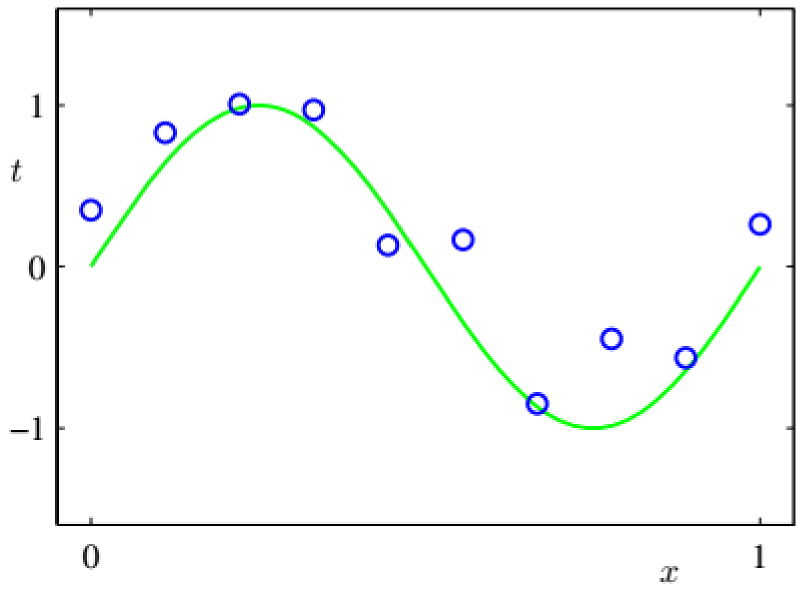


Figure 1.2.1: The plot of a training data set of N = 10 points, shown as blue circles, each comprising an observation of the input variable x along with the corresponding target variable t. The green curve shows the function sin(2πx) used to generate the data. Our goal is to predict the value of t for some new value of x, without knowledge of the green curve.

Test Set. Our goal is to exploit the training set in order to build a model which is able to predict the target value for a new input accurately. In other words, the goal is to achieve good *generalisation* by making accurate predictions for new data. We can obtain some *quantitative* insight into the generalisation of our model by considering a separate *test set* comprising 100 data points generated using exactly the same procedure used to generate the training set points. When model training is finished, we use the test set to assess the goodness of our model by *comparing* the predicted value $\hat{t}$ with the correct (or gold) value t for each new input x in the test set. Note that we are *not* allowed to use the test set while the model is trained; otherwise, it would be cheating.

Assuming a Model Class

To learn the unknown function relating the input x to output y, we need to assume a *model* for this relationship. In general, the model can be *parametric* or *non-parametric*. In a parametric model, the number of parameters is *fixed* and does not depend on the size of the training set. Whereas in a non-parametric model, the number of parameters can grow as the size of the training set increases, i.e. potentially there are an infinite number of parameters. *K-nearest neighbour classifiers* are examples of non-parametric models, which we will see later in this module.

For our regression problem, we consider a parametric model which is the class of degree M polynomials*:*

$$y(x, \mathbf{w}) := w_0 + w_1 x + w_2 x^2 + \ldots + w_M x^M = \sum_{j=0}^{M} w_j x^j$$

where the polynomial coefficients $\mathbf{w} := (w_0, \ldots, w_M)$ are the model parameters. Note that, although the polynomial function $y(x, \mathbf{w})$ is a nonlinear function of x, it is a linear function of the coefficients $\mathbf{w}$. Functions, such as the polynomial, which are linear in the unknown parameters have important properties and are called *linear models* and will be discussed extensively in Modules 2 and 3. We now need to learn the parameters of our model based on the training data.

Training the Model

The values of the coefficients will be determined by fitting the polynomial to the training data. This can be done by *minimising* an *error function* that measures the misfit between the function $y(x, \mathbf{w})$, for any given value of $\mathbf{w}$, and the training set data points. This means that *learning,* as we approach it, is closely related to *mathematical optimisation.*

Training Objective. For our polynomial curve fitting example, one simple choice of error function is given by the sum of the squares of the errors between the predictions $y(x_n, \mathbf{w})$ for each training data point x_n and the corresponding target value t_n. So we minimise the following objective function to train the model:

$$E(\mathbf{w}) := \frac{1}{2} \sum_{n=1}^{N} [y(x_n, \mathbf{w}) - t_n]^2$$

where the factor of 1/2 is included for later convenience. To motivate this error function, note that it is a nonnegative quantity that would be zero if, and only if, the function $y(x, \mathbf{w})$ were to pass exactly through each training data point.

Optimisation Algorithm. We can solve the learning problem by choosing the value of $\mathbf{w}^*$ for which $E(\mathbf{w}^*)$ is as small as possible:

$$\mathbf{w}^* := \arg\min_{\mathbf{w}} E(\mathbf{w})$$

We know from the calculus that to solve such continuous optimisation problem, we need to take the derivative of the error function $E(\mathbf{w})$ with respect to $\mathbf{w}$ and find $\mathbf{w}$ for which the derivative is zero; such values are called stationary points of the error function. We may be able to find a closed form solution for the stationary points of the error function, or we may resort to iterative algorithms to find them (we will see these algorithms in details in Module 2).

For our example curve fitting problem, the error function is a quadratic function of the coefficients $\mathbf{w}$, so its derivatives with respect to the coefficients will be linear in the elements of $\mathbf{w}$. Therefore, the minimization of the error function has a unique solution, denoted by $\mathbf{w}^*$, which can be found in closed form. The resulting polynomial is denoted by the function $y(x, \mathbf{w}^*)$.

Model Complexity vs Generalisation

There remains the problem of choosing the order M of the polynomial, and as we shall see this will turn out to be an example of an important concept called *model selection*. Intuitively, the higher the degree, the more complex the model; since it has more parameters. So we take M as the complexity measure for our polynomial model. In Figure **1.2.2**, we show four examples of the results of fitting polynomials having orders M = 0, 1, 3, and 9 to the data set shown in Figure **1.2.1**.

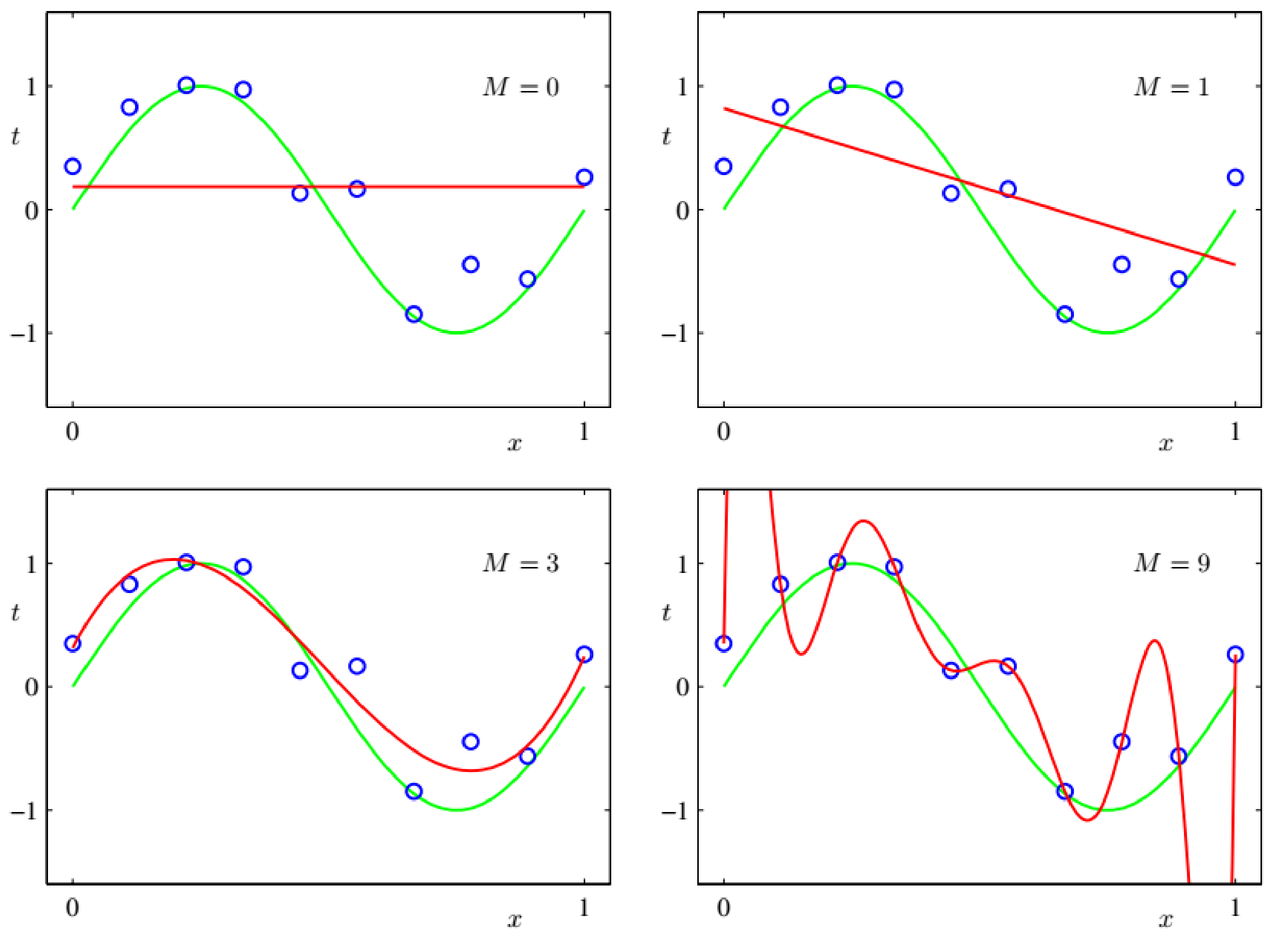


Figure 1.2.2: The plots of polynomials having various orders M, shown as red curves, fitted to the data set shown in Figure 2.1. The green curve indicates the actual (generative) curve while the red curve is the fitted one.

Over-fitting and Under-fitting. Note that the constant ($M = 0$) and first order ($M = 1$) polynomials give rather poor fits to the data and consequently rather poor representations of the function $sin(2\pi x)$. This behaviour is known as *under-fitting*. The third order ($M = 3$) polynomial seems to give the best fit to the function $sin(2\pi x)$ of the examples shown in Figure **1.2.2.** When we go to a much higher order polynomial ($M = 9$), we obtain an excellent fit to the training data. In fact, the polynomial passes exactly through each data point and $E(\mathbf{w}) = 0$. However, the fitted curve oscillates wildly and gives a very poor representation of the function $sin(2\pi x)$. This latter behaviour is known as *over-fitting*.

Generalisation Performance. As noted earlier, the goal is to achieve good generalisation by making accurate predictions for new data. We obtain quantitative insight into the generalisation performance of a fitted model by computing its error on the *test set*. For our curve fitting example, we use root-mean-square (RMS) error defined by $E_{RMS} := \sqrt{2E(\mathbf{w}^*)/N}$ where the division by N allows us to compare different sizes of data sets on an equal footing.

Recall that the test set was not used when the model was trained, so the model would have a low error on the test set only if it captures generalisable aspects of the functional dependence between the input and output. In other words, the test error would be high if the model mostly captures the particularities specific to the training set (such as noise) which are not generalisable.
Model Complexity vs Generalisation. Graphs of the training and test set RMS errors are shown, for

various values of M, in Figure **1.2.3**. The test set error is a measure of how well we are doing in predicting the values of t for new data observations of x. We note from Figure **1.2.3** that small values of M give relatively large values of the test set error, and this can be attributed to the fact that the corresponding polynomials are rather inflexible and are incapable of capturing the oscillations in the function $sin(2\pi x)$ (i.e. under-fitting). Values of M in the range $3 \leq M \leq 8$ give small values for the test set error, and these also give reasonable representations of the generating function $sin(2\pi x)$, as can be seen, for the case of $M = 3$, from Figure **1. 2.2**. For $M = 9$, the training set error goes to zero; however, the test set error has become very large (i.e. over-fitting). As we saw in Figure **1.2.2**, the corresponding function $y(x, \mathbf{w})$ exhibits wild oscillations.

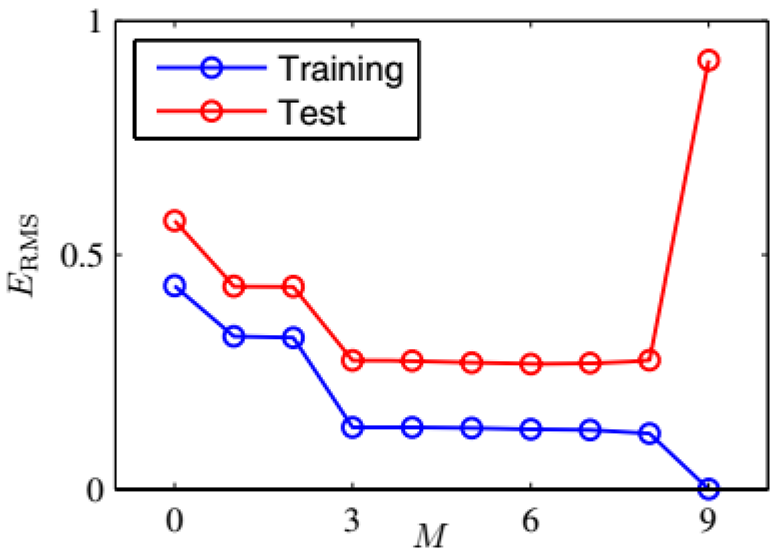


Figure 1.2.3: Graphs of the root-mean-square error evaluated on the training set and on an independent test set for various values of M.

Regularisation

A power series expansion of the function $sin(2\pi x)$ contains terms of all orders, so we might expect that the test error should decrease monotonically as we increase the degree M. We can gain some insight into this paradox by examining the values of the coefficients $\mathbf{w}$ obtained from polynomials of various order, as shown in Table **1.2.1**. We see that, as M increases, the *magnitude* of the coefficients typically gets larger; look particularly the coefficients for $M = 9$. Intuitively, large weights in high-degree polynomials make the model too flexible so that it is increasingly tuned to the random noise on the target values. Hence, the main issue is not the degree, it is, in fact, the magnitude of the coefficients.

	$M = 0$	$M = 1$	$M = 6$	$M = 9$
$w_0^\star$	0.19	0.82	0.31	0.35
$w_1^\star$		-1.27	7.99	232.37
$w_2^\star$			-25.43	-5321.83
$w_3^\star$			17.37	48568.31
$w_4^\star$				-231639.30
$w_5^\star$				640042.26
$w_6^\star$				-1061800.52
$w_7^\star$				1042400.18
$w_8^\star$				-557682.99
$w_9^\star$				125201.43

Table 1.2.1 Coefficients for polynomials of various order.

Also, there is something rather unsatisfying about having to limit the number of parameters in a model according to the size of the available training set. It would seem more reasonable to choose the complexity of the model according to the complexity of the problem being solved. Figure **1.2.4.** examines the behaviour of a given model as the size of the data set is varied. We see that, for a given model complexity, the over-fitting problem become less severe as the size of the data set increases. Another way to say this is that the larger the data set, the more complex (in other words more flexible) the model that we can afford to fit to the data.

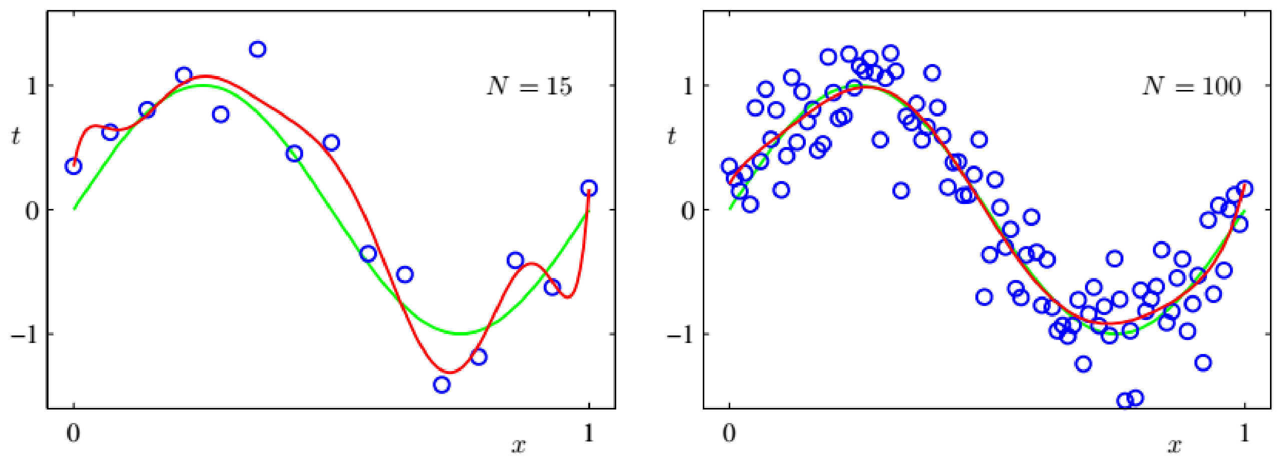


Figure 1.2.4: Plots of the solutions obtained by minimising the sum-of-squares error function using the M = 9 polynomial for N = 15 data points (left plot) and N = 100 data points (right plot). We see that increasing the size of the data set reduces the over-fitting problem.

Our desiderata, therefore, is to use high-degree polynomials as our model but to have a training mechanism which prevents overfitting on a given (possibly small) training set. One technique that is often used to control the over-fitting phenomenon in such cases is that of *regularisation*, which involves adding a *penalty* term to the error function in order to discourage the coefficients from reaching large values. The simplest such penalty term takes the form of a sum of squares of all of the coefficients, leading to a modified error function of the form:

$$E(\mathbf{w}) := \frac{1}{2}\sum_{n=1}^{N}[y(x_n,\mathbf{w})-t_n]^2 + \frac{\lambda}{2}\|\mathbf{w}\|^2$$

where $\|\mathbf{w}\|^2 := w_0^2 + w_1^2 + \ldots + w_M^2$, and the *regularisation parameter* λ governs the relative importance of the regularisation term compared with the sum-of-squares error term. Looking more closely at this training objective, *it is a tradeoff between the empirical error (the left term) and a measure of the model complexity (the right term)*. Such training objectives are commonplace in Machine Learning, and we will see them in different Modules in future.

The impact of the regularisation term on the generalisation error can be seen by plotting the value of the RMS error for both training and test sets against $\ln\lambda$, as shown in Figure **1.2.5**. We see that in effect λ now controls the *effective complexity* of the model and hence determines the degree of over-fitting.

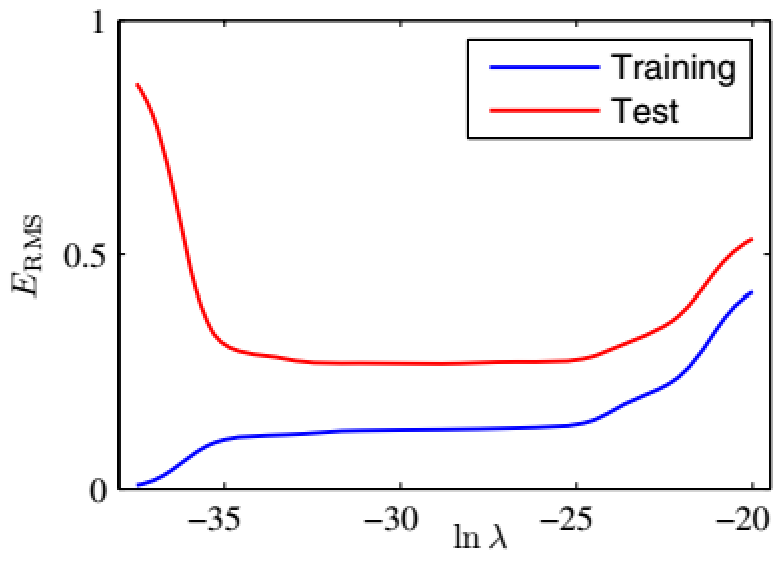


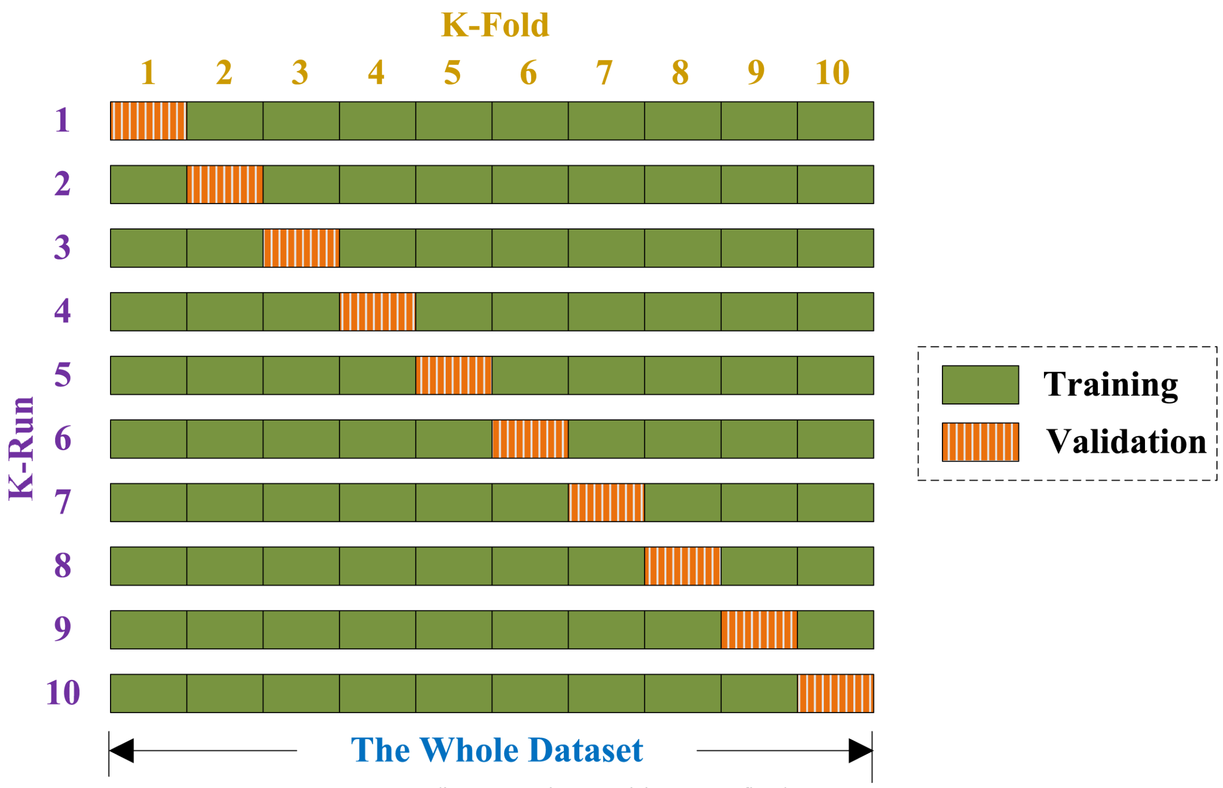
Figure 1.2.5: Graph of the root-mean-square error versus ln λ for the M = 9 polynomial.

Model Selection

In our example of polynomial curve fitting using least squares, we saw that there was an optimal order of polynomial that gave the best generalization. The order of the polynomial controls the number of free parameters in the model and thereby governs the model complexity. With regularised least squares, the regularisation coefficient λ also controls the effective complexity of the model, whereas for more complex models, such as mixture distributions or neural networks there may be multiple parameters governing complexity. In a practical application, we need to determine the values of such parameters, and the principal objective in doing so is usually to achieve the best predictive performance on new data. Furthermore, as well as finding the appropriate values for complexity parameters within a given model, we may wish to consider a range of different types of model in order to find the best one for our particular application.

Validation Set. If data is plentiful, then one approach is simply to use some of the available data to train a range of models, or a given model with a range of values for its complexity parameters, and then to compare them on independent data, sometimes called a *validation set*, and select the one having the best predictive performance. For example, one may reserve 10% of the available data (that are randomly selected) for the evaluation process and train his model using the other 90% of the dataset. This process is usually referred as *hold-out validation technique*.

K-Fold Cross-Validation. For a better analysis, one may repeat the above process a number of times. For example, one may divide the available dataset into K equal-size distinct subsets, and each time use one of these subsets ($1/K$ of all samples) as validation set and the combination of all other $K-1$ subsets as the training set. This procedure is repeated K times to ensure all samples are used for both training ($K-1$ times) and validation (only once). The average of the obtained validation errors is used as an estimation of the testing error. In machine learning, this procedure is widely known as K-Fold cross-validation (https://en.wikipedia.org/wiki/Cross-validation_%28statistics%29#k-fold_cross-validation). Figure **1.2.6.** illustrates K-Fold CV procedure.



(https://peerj.com/articles/1251/#fig-5)

Figure 1.2.6. The K-Fold Cross-Validation Process. Source: https://peerj.com/articles/1251/

One major drawback of K-fold cross-validation is that the number of training runs that must be performed is increased by a factor of K, and this can prove problematic for models in which the training is itself computationally expensive. A further problem arises from the fact that we might have multiple complexity parameters for a single model (for instance, there might be several regularisation parameters). Exploring combinations of settings for such parameters could, in the worst case, require a number of training runs that is exponential in the number of parameters, which is prohibitive.

Leave-One-Out Cross Validation. It is a special case of K-Fold cross-validation where K (i.e., the number of folds/subsets) is equal to the size of the training dataset. Therefore, in each iteration, one training datapoint is left out as the validation set and all the others are used to build the model. This procedure is repeated K times. This is to make sure that all datapoints are selected exactly once as in the validation phase.

3 Activity 1.1: K-Nearest Neighbour

In this activity we study model complexity and generalisation using KNN classifier. This activity helps you to complete Assignment 1.

Please download this (https://www.alexandriarepository.org/wp-content/uploads/20160704112304/Activity11.zip) zip file (right click and then choose "save link as") this contains the R scripts (.r), Jupyter notebooks (.ipynb) and output files (.html). If you have access to a Jupyter instance, the notebook is enough to run the experiments. Otherwise, you may read the instructions and discussions from the HTML file and execute the R script.

For detailed discussion about setting your programming environment up, please refer to Appendix B: Setting up Your Programming Environment (https://www.alexandriarepository.org/reader/1-the-elements-of-machine-learning/97654).

4
Probabilistic Machine Learning

A key concept in machine learning is that of *uncertainty*. It arises both through the noise in measurements, as well as through the finite size of data sets. Probability theory provides a consistent framework for the quantification and manipulation of uncertainty and forms one of the central foundations for machine learning. In what follows, we first review the basics of probability theory. We then show how it helps in capturing our uncertainty when learning models and making predictions within two schools of thought, i.e. frequentist and Bayesian statistics.

Probability Theory Concepts

A *random variable* is a variable whose value is subject to variation due to chance. For example, we can denote the outcome of tossing a coin by the random variable X whose values can be either H (for Heads) or T (for Tails); technically we say that the *domain* of our random variable is $D_X := \{H,T\}$. To begin with, we shall define the *probability* of an event to be the *fraction* of times that event occurs out of the total number of trials, in the *limit* that the total number of trials goes to infinity. For example, If we toss the coin many times and 20% of times it lands tails, we say that the probability of tails is 20%. The *probability distribution* for random variable X is a function which maps each value of that random variable to its probability. For our coin example, we can denote the probability distribution by $p(.)$, hence $p(X=H)=.2$ and $p(X=T)=.8$. Note that, based on the definition of probability, we have

$$\sum_{a \in D_X} p(X=a) = 1 \text{ and } \forall a \in D_X : p(X=a) \geq 0$$

When there is no confusion about the domain of the random variable, we overloaded the notation and write $\sum_x p(x) = 1$ where the lowercase letter denotes the values.

We can extend the definition of probability from one variable to *multiple* variables, which results in the notion of *joint* probability distribution. For example, given two random variables X and Y, the joint probability distribution $p(X,Y)$ gives the probability of possible combination of values for these two random variables, hence $\sum_x \sum_y p(x,y) = 1$. We can now define the *marginal* and *conditional* probability distributions, and extend the definitions from *discrete* valued random variables to continuous-valued random variables via *probability density function* where $\int_{-\infty}^{\infty} p(x)dx = 1$. Appendix A contains the basic concepts of probability theory; please read through it if you feel that you need a refreshment. We now turn our attention to some important concepts from probability theory.

The Sum and product Rules. Given two random variables X and Y, based on the definitions we have had so far, we can write the two fundamental rules of probability theory:

$$\text{Sum Rule: } p(x) = \sum_y p(x,y)$$

$$\text{Product Rule: } p(x,y) = p(y|x)p(x)$$

Bayes Theorem. From the product rule, we immediately obtain the following relationship between conditional probabilities which is called the Bayes' theorem:

$$p(y|x) = \frac{p(x|y)p(y)}{p(x)}$$

Using the sum rule, the denominator in Bayes' theorem can be expressed in terms of the quantities appearing in the numerator:

$$p(x) = \sum_y p(x|y)p(y)$$

We can view the denominator in Bayes' theorem as being the normalisation constant required to ensure that the sum of the conditional probability on the left-hand side of the above equation over all values of Y equals one.

Independence. Two random variables are called independent if and only if

$$p(x,y) = p(x)p(y)$$

For example, if we toss a coin independently two times, the outcome of the first toss X and the outcome of the second toss Y are independent, hence, their joint probability will be the product of their marginal probabilities.

The Coin Example

Suppose we are given a dataset which contains the outcomes of 10 *independent* tosses of the same coin $\mathcal{D} := \{H,T,T,T,T,H,T,T,H,T\}$, where H and T stand for Head and Tail, respectively. We are asked to build a model for this coin tossing and determine the best value for our model parameters.

We can imagine that this is a damaged coin so that the probability of landing heads is not necessarily the same as that of landing tails. We take a simple probabilistic model for our coin, in which the probability of coming head is denoted by the *parameter* w where $0 \leq w \leq 1$, in other words, $p(x=H) = w$. Since the sum of the probabilities over all outcomes must be one, we know that $p(x=T) = 1-w$.

Fixing our parametric model for the coin, we can now calculate the probability of having the dataset D generated from the coin. Why may we want to do that? Because we can then choose a value for our coin parameter which maximises the probability of the generated dataset D, which intuitively seems a reasonable thing to do. The probability of the dataset given the model is called the *likelihood*, which plays a central role in machine learning.

Now let's compute the probability of the dataset based on our parametric model for the coin. Since the coin tosses are *independent*, the probability of the dataset is the product of the probability of individual data points:

$$p(\mathcal{D}|\text{coin model}) = \prod_{i=1}^{10} p(x_i|\text{coin model}) = w^3(1-w)^7$$

Note that the likelihood is a function of w, so we need to set its derivative to zero to maximise it. Instead of maximising the likelihood, we can maximise its log denoted by $\mathcal{L}(w)$, which does not change the solution (since log is a monotone function) but is mathematically more convenient:

$$\frac{d\mathcal{L}(w)}{dw} = 0 \Rightarrow \frac{d[3\log w + 7\log(1-w)]}{dw} = 0 \Rightarrow \frac{3}{w} - \frac{7}{1-w} = 0 \Rightarrow w = \frac{3}{3+7}$$

You may now think that it was obvious from the beginning that a reasonable value for the coin parameter should have been $3/10$. However, this example is meant to give a rigorous reasoning for this reasonable intuition. Shortly, we will see that *maximum likelihood* is a central parameter estimation technique in machine learning, and we will apply it to more complex probabilistic models in future.

A Probabilistic Approach to Machine Learning

In probabilistic machine learning, we would like to capture our uncertainty about the data using probability theory. As mentioned earlier, the uncertainty arises both through the noise in measurements, as well as through finite size of data sets. We may wish to represent our uncertainty about model prediction, or even the estimated values for model parameters. In the following, we first see a principle, rooted in probability theory, to estimate model parameters. We then see a technique which can be used to formalise our uncertainties about predictions and estimated model parameters.

Maximum Likelihood Principle. The *maximum likelihood* is a widely used parameter estimation technique in which the model parameters is set to the value that maximises the likelihood function $p(\mathcal{D}|\text{model})$. This corresponds to choosing the value of parameters for which the probability of the observed data set is maximised (as seen in the coin example). In the machine learning literature, the negative log of the likelihood function is called an *error function*. Because the negative logarithm is a monotonically decreasing function, maximising the likelihood is equivalent to minimising the error.

Bootstrap for Quantifying Uncertainty. One approach to quantifying the uncertainty is the *bootstrap* technique, which works as follows. Suppose we are given a data set D, and we would like to fit a model with parameters w to this data. We can resort to maximum likelihood, and estimate a value for w which clearly depends on D. Now, we may wonder to some extent our estimated parameters would change if we have an alternate dataset D'. If we do this exercise for several alternate datasets, then we will a *distribution* over our maximum likelihood estimates for the parameters. The higher the variance of this distribution, the more uncertainty we have about our parameter estimates. The bootstrap technique helps us to quantify our uncertainty not only about the parameter estimates but also about the predictions of the model.

Now the question is how to get the alternate datasets. We are given only one initial dataset, and in most cases, we are not allowed to request more datasets. The idea is to generate the alternate datasets from the original one, as follows. Suppose our original data set consists of N data points $\mathcal{D} = \{x_1, \ldots, x_N\}$. We can create a new dataset $\mathcal{D}'$ by drawing N points at *random* from $\mathcal{D}$ with *replacement.* Therefore, some points in $\mathcal{D}$ may be replicated in $\mathcal{D}'$, whereas other points in $\mathcal{D}$ may be absent from $\mathcal{D}'$. This process can be repeated L times to generate L data sets each of size N and each obtained by sampling from the original data set $\mathcal{D}$. The statistical accuracy of parameter estimates can then be evaluated by looking at the variability of predictions between the different bootstrap data sets.

Bayesian Approach to Machine Learning

So far in this chapter, we have viewed probabilities in terms of the frequencies of random, repeatable events. We shall refer to this as the *classical* or *frequentist* interpretation of probability. Now we turn to the more general *Bayesian* view, in which probabilities provide a quantification of uncertainty.

Consider an uncertain event, for example, whether the moon was once in its own orbit around the sun, or whether the Arctic ice cap will have disappeared by the end of the century. These are not events that can be repeated numerous times in order to define a notion of probability as we did earlier in the context of boxes of fruit. Nevertheless, we will generally have some idea, for example, of how quickly we think the polar ice is melting. If we now obtain fresh evidence, for instance from a new Earth observation satellite gathering novel forms of diagnostic information, we may revise our opinion on the rate of ice loss.

In the field of machine learning, too, it is helpful to have a more general notion of probability. Consider the example of polynomial curve fitting discussed in Section 2. It seems reasonable to apply the frequentist notion of probability to the random values of the observed variables t_n. However, we would like to address and quantify the uncertainty that surrounds the appropriate choice for the model parameters $\mathbf{w}$. We shall see that, from a Bayesian perspective, we can use the machinery of probability theory to describe the uncertainty in model parameters such as $\mathbf{w}$, or indeed in the choice of the model itself.

Prior and Posterior. We capture our assumptions about model parameters $\mathbf{w}$, *before* observing the data, in the form of a *prior* probability distribution $p(\mathbf{w})$. The effect of the observed data $\mathcal{D}$is expressed through the conditional probability $p(\mathcal{D}|\mathbf{w})$. Bayes' theorem, which takes the form:

$$p(\mathbf{w}|\mathcal{D}) = p(\mathcal{D}|\mathbf{w})p(\mathbf{w})/p(\mathcal{D}) \quad (1.3.1)$$

then allows us to evaluate the uncertainty in $\mathbf{w}$ after we have observed $\mathcal{D}$ in the form of the *posterior* probability $p(\mathbf{w}|\mathcal{D})$. We can state Bayes' theorem in words

$$\text{posterior} \propto \text{likelihood} \times \text{prior}$$

where all of these quantities are viewed as functions of $\mathbf{w}$. The denominator in **(1.3.1)** is the normalisation constant, which ensures that the posterior distribution on the left-hand side is a valid probability density and integrates to one. Indeed, integrating both sides of **(3.1)** with respect to $\mathbf{w}$, we can express the denominator in Bayes' theorem in terms of the prior distribution and the likelihood function

$$p(\mathcal{D}) = \int p(\mathcal{D}|\mathbf{w})p(\mathbf{w})d\mathbf{w}$$

Frequentist Vs Bayesian. In both the Bayesian and frequentist paradigms, the likelihood function $p(\mathcal{D}|\mathbf{w})$ plays a central role. However, the manner in which it is used is fundamentally different in the two approaches. In a frequentist setting, $\mathbf{w}$ is considered to be a fixed parameter, whose value is determined by some form of *estimator*, and error bars on this estimate are obtained by considering the distribution of possible data sets $\mathcal{D}$ (c.f. the bootstrapping technique). By contrast, from the Bayesian viewpoint, there is only a single data set $\mathcal{D}$ (namely the one that is actually observed), and the uncertainty in the parameters is expressed through a probability distribution over $\mathbf{w}$.

Bayesian Machine Learning. We are now ready to see the fundamental rules of Bayesian machine learning:

$$\text{Prediction: } p(x|\mathcal{D}, M) = \int p(x|\mathbf{w}, \mathcal{D}, M)p(\mathbf{w}|\mathcal{D}, M)d\mathbf{w}$$

$$\text{Model Comparison: } p(M|\mathcal{D}) = \frac{p(\mathcal{D}|M)p(M)}{p(\mathcal{D})}$$

where $p(\mathcal{D}|M) = \int p(\mathcal{D}|\mathbf{w}, M)p(\mathbf{w}|M)d\mathbf{w}$.

The Coin Example: A Bayesian View

Now let us revisit our coin example, and assume that we are given a dataset of the outcome of ten coin tosses where all of which turn out to be heads. If we use the maximum likelihood principle, then the parameter of the coin w (i.e. probability of coming heads) turns out to be one! This means that, we would predict the probability of coming tails for a future coin toss is zero, which may seem unreasonable! The fact is that we have a very small dataset consisting of only ten coin tosses and may have been unlucky to get all of them to be heads, so we should not only rely on the data and instead need to capture our *prior belief* as well.

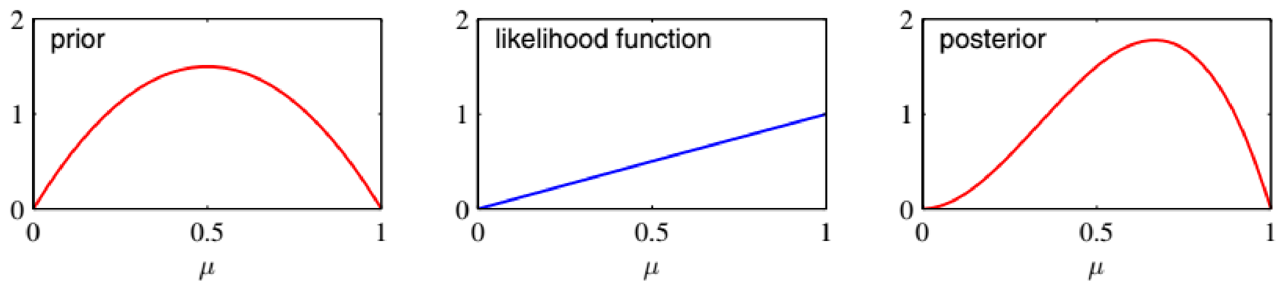


Figure 1.3.1: Illustration of one step of sequential Bayesian inference. The prior is given by a beta distribution with parameters a = 2, b = 2, and the likelihood function with N = m = 1, corresponds to a single observation of x = 1, so that the posterior is given by a beta distribution with parameters a = 3, b = 2.

A priori, before seeing any dataset, we may believe that a given coin is most probably fair, i.e. the probability of coming heads and tails are the same and equal .5. Let us assume we use the Beta distribution to represent our prior belief:

$$\text{Beta}(w|a, b) \propto w^{a-1}(1-w)^{b-1}$$

The Beta distribution has two parameters (a, b), and puts a distribution over $w \in [0, 1]$ where the mean of the distribution is $\frac{a}{a+b}$. In our case, we may choose to set $(a = 2, b = 2)$ to represent the fact that a priori we believe that most probably a coin is fair, i.e. our prior belief is concentrated around

$\frac{2}{2+2}$. Now we can combine the prior with the likelihood to come up with our updated belief (namely the posterior) about the parameter of the coin (See Figure **1.3.1**):

$$\begin{aligned}\text{Posterior} &= \text{Beta}(a,b) \times \text{Likelihood}\\ &\propto w^{a-1}(1-w)^{b-1} \times w^{10}\\ &= w^{a+10-1}(1-w)^{b-1}\\ &= \text{Beta}(a+10,b)\end{aligned}$$

As seen in the last step, the posterior turns out to be a Beta distribution with parameters $(a+10,b)$. So if we set $\text{Beta}(2,2)$ as our prior, the posterior would be $\text{Beta}(12,2)$. This means in our updated belief, we mostly believe that the parameter of the coin is $\frac{12}{12+2}$. So in our Bayesian viewpoint we give every value a chance to be the parameter of the coin, where 12/14 is the most probable value.

Interestingly, if we get a new batch of data containing the outcomes of new tosses of the coin, we can start with our current belief $\text{Beta}(10+a,b)$ as the prior, and convert it to the posterior using the likelihood of the new batch of data. This is called the *Bayesian updating*.

5
Activity 2: Bootstrapping

In Chapter 3, we have learned that the Bootstrapping is a powerful statistical tool that helps us to measure the uncertainty in the prediction of model. In this activity, we implement this technique to assess the variation in the prediction of KNN classifier for fixed values of K. This activity helps you to complete Assignment 1.

Please download this (https://www.alexandriarepository.org/wp-content/uploads/20160704112304/Activity11.zip) zip file (right click and then choose "save link as") that contains the R scripts (.r), Jupyter notebooks (.ipynb) and output file (.html). If you have access to a Jupyter instance, the notebook is enough to run the experiments. Otherwise, you may read the instructions and discussions from the HTML file and execute the R script.

For detailed discussion about setting your programming environment up, please refer to Appendix B: Setting up Your Programming Environment (https://www.alexandriarepository.org/reader/1-the-elements-of-machine-learning/97654).

6 Appendix A: Probability Background

Basics of Probability

We will introduce the basic concepts of probability theory by considering a simple example. Imagine we have two boxes, one red and one blue, and in the red box we have 2 apples and 6 oranges, and in the blue box, we have 3 apples and 1 orange. This is illustrated in Figure **A.1**. Now suppose we *randomly* pick one of the boxes and from that box we *randomly* select an item of fruit, and having observed which sort of fruit it is we replace it in the box from which it came. We could imagine repeating this process many times. Let us suppose that in so doing we pick the red box 40% of the time and we pick the blue box 60% of the time, and that when we remove an item of fruit from a box we are equally likely to select any of the pieces of fruit in the box.

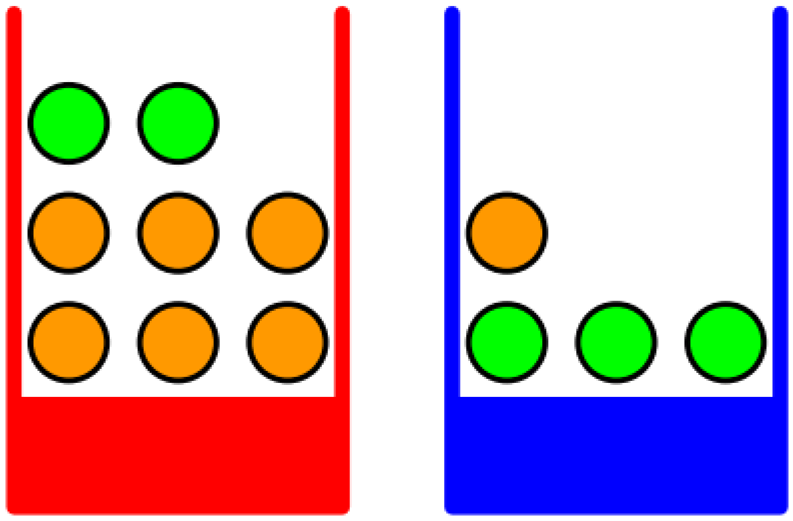


Figure A.1: We use a simple example of two coloured boxes each containing fruit (apples are shown in green and oranges are shown in orange) to introduce the basic ideas of probability.

Random Variable. In this example, the identity of the box that will be chosen is a *random variable*, which we shall denote by B. This random variable can take one of the two possible *values*, namely r (corresponding to the red box) or b (corresponding to the blue box). Similarly, the identity of the fruit is also a random variable and will be denoted by F. It can take either of the values a (for apple) or o (for orange).

Probability. We define the *probability* of an *event* to be the *fraction* of times that event occurs out of the total number of trials, in the limit that the total number of trials goes to infinity. Thus, the probability of selecting the red box is 4/10, and the probability of selecting the blue box is 6/10. We write these probabilities as $p(B=r)=4/10$ and $p(B=b)=6/10$. Note that, by definition, probabilities must lie in the interval $[0,1]$. Also, if the events are mutually exclusive and if they include all possible outcomes (for instance, in this example the box must be either red or blue), then we see that the probabilities for those events must sum to one.

Joint/Marginal/Conditional Probability. Consider the slightly more general example shown in Figure **A.2** involving two random variables X and Y (which could for instance be the Box and Fruit variables considered above). We shall suppose that X can take any of the values x_i where $i = 1, \ldots, M$, and Y can take the values y_j where $j = 1, \ldots, L$. Consider a total of N trials in which we sample both of the variables X and Y, and let the number of such trials in which $X = x_i$ and $Y = y_j$ be denoted by n_{ij}. The probability that X will take the value x_i and Y will take the value y_j is written $p(X = x_i, Y = y_j)$ and is called the *joint* probability of $X = x_i$ *and* $Y = y_j$. It is defined by the number of points falling in the cell i, j as a fraction of the total number of points:

$$p(X = x_i | Y = y_j) := \frac{n_{ij}}{N}$$

Here we are implicitly considering the limit $N \rightarrow \infty$. Similarly, the probability that X takes the value x_i irrespective of the value of Y, called the *marginal probability* $p(X = x_i)$, is defined by the fraction of the total number of points that fall in column i:

$$p(X = x_i) := \frac{c_i}{N}$$

where c_i is the number of trials in which X takes the value x_i (irrespective of the value that Y takes).

If we consider only those instances for which $X = x_i$, then the fraction of such instances for which $Y = y_j$ is written $p(Y = y_j | X = x_i)$ and is called the *conditional probability*. It is defined by finding the fraction of those points in column i that fall in cell i, j:

$$p(X = x_i, Y = y_j) := \frac{n_{ij}}{c_i}$$

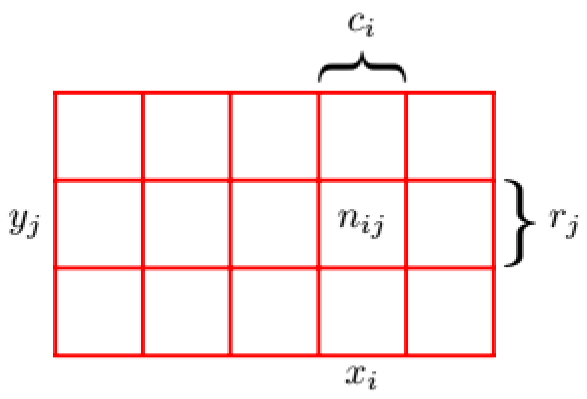


Figure A.2: We can derive the sum and product rules of probability by considering two random variables, X, which takes the values {xi} where i = 1, . . . , M, and Y , which takes the values {yj} where j = 1, . . . , L. In this

illustration we have M = 5 and L = 3. If we consider a total number N of instances of these variables, then we denote the number of instances where X = xi and Y = yj by nij, which is the number of points in the corresponding cell of the array. The number of points in column i, corresponding to X = xi, is denoted by ci, and the number of points in row j, corresponding to Y = yj, is denoted by rj.

Expectation, Variance, Covariance. One of the most important operations involving probabilities is that of finding weighted averages of functions. The average value of some function $f(x)$ under a probability distribution $p(x)$ is called the *expectation*: $\mathbb{E}[f] := \sum_x p(x)f(x)$.

The variance of $f(x)$ is defined by $var[f] := \mathbb{E}[(f(x) - E[f])^2]$. It can be shown that the variance can be re-written as $\mathbb{E}[f(x)^2] - \mathbb{E}[f]^2$.

For two random variables x and y, the covariance is defined by $cov[x,y] := \mathbb{E}_{x,y}[(x - \mathbb{E}[x])(y - \mathbb{E}[y])]$.

Continuous Random Variables. So far we have seen random variables with discrete values, and the definition of *probability distribution* for them. However, there is another type of random variables which accepts continuous values. We define the *probability density function* for continuous random variables which must satisfy the following conditions

$$p(x) \geq 0 \qquad \text{and} \qquad \int_{-\infty}^{\infty} p(x)dx = 1$$

The probability that x lies in the interval $(-\infty, z)$ is given by the *cumulative distribution function* defined by

$$P(z) := \int_{-\infty}^{z} p(x)dx$$

If we have several continuous variables $x_1, \ldots, x_D$, denoted collectively by the vector $\mathbf{x}$, then we can define a *joint* probability density $p(\mathbf{x}) = p(x_1, \ldots, x_D)$. Accordingly, we can define the conditional and marginal probability density functions, Bayes' theorem, expectation etc for this case.

An example of a famous probability density function for continuous random variables is the *normal* or *Gaussian* distribution. For the case of a single real-valued variable x, the Gaussian distribution is defined by

$$\mathcal{N}(x|\mu,\sigma) := \frac{1}{(2\pi\sigma^2)^{\frac{1}{2}}} \exp\{-\frac{1}{2\sigma^2}(x-\mu)^2\}$$

where μ and σ are the parameters of the distribution. It can be shown that μ is the expectation and σ^2 is the variance of the Gaussian distribution.

7 Appendix B: Setting up Your Programming Environment

This unit includes a number of programming exercises and assignments that you will need to finish in order to complete the unit. The primary programming language we will use is R, except for last module where we switch to Python (as SparkR is in beta version at the time of unit development). We will employ Jupyter Notebooks, an interactive computing and development environment. We expect that you already know at least the basics of R and Python programming, but may not have encountered Jupyter Notebooks.

This chapter provides a path to JuPyteR installation and configuration. All of these steps are well-documented on the Internet. We expect that you will be well on your way after you go through the reference materials.

While you can develop and execute your code using R/Python terminals we recommend you use JuPyteR notebooks as they provide a clean interactive environment for practice and reporting. For this reason, you need to have a local JuPyteR setup unless you intend to use online or cloud resources such as Microsoft Azure ML Studio (https://studio.azureml.net/), IBM powered Data Scientist's Workbench (DSW) (https://datascientistworkbench.com/), Data Bricks Notebook (https://databricks.com/) etc. Note that we recommend:

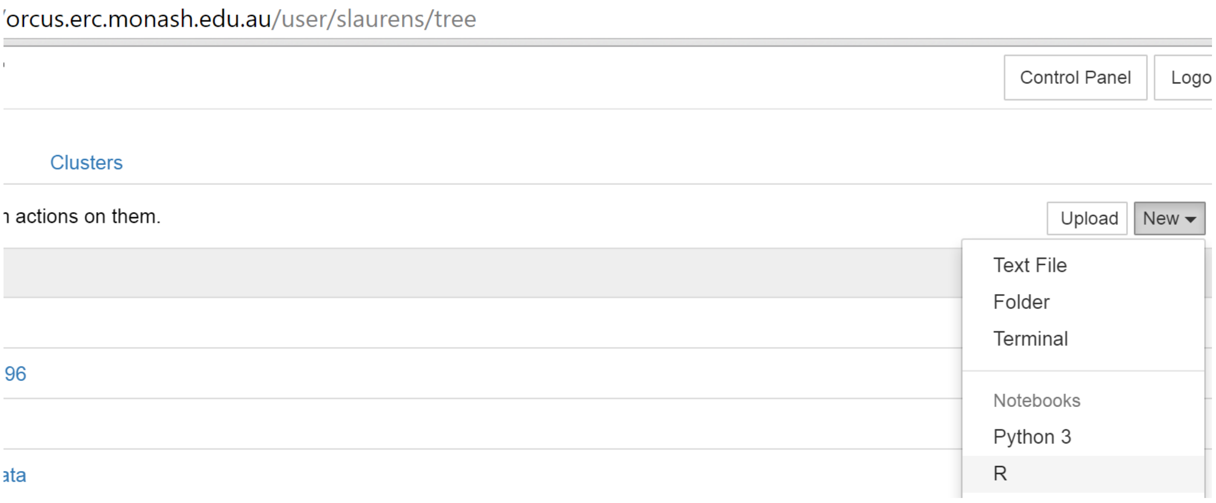
- using JuPyteR locally for Activity 2 of Module 5 which requires 'H2O' installation (otherwise you can use H2O Flow (http://www.h2o.ai/product/flow/) instead of H2O for R (http://www.h2o.ai/download/))
- using IBM DSW for Module 6 activities (and Assignment 4) as you need a SPARK cluster.

Installing JuPyteR

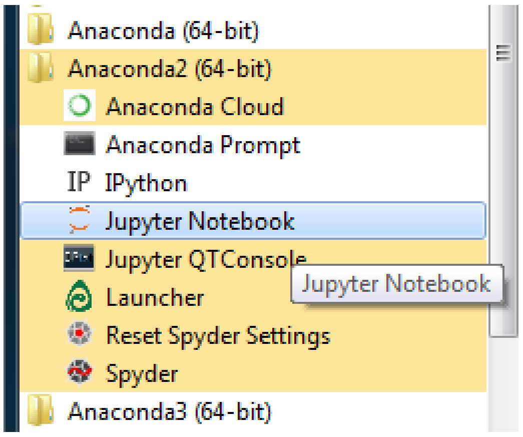
The easiest way to run JuPyteR notebooks is to follow the instructions here (https://www.continuum.io/downloads) to install Anaconda (Python) for Windows, Mac OS X and Linux. Please note that you should always download the distribution for Python 2.7, not Python 3.5. Then, you need to download (https://cran.r-project.org/) and install R Optionally, you may chose to also install RStudio (https://www.rstudio.com/) as it provides a free, open source and popular programming IDE for R. Microsoft R Open (https://mran.microsoft.com/download/) also provides a fast and enhanced alternative. Finally, you should follow these instructions (https://irkernel.github.io/installation/) to install iRkernel. This kernel adds R to your Anaconda Python notebook installation.

Running JuPyteR (IPython) Notebooks on Windows

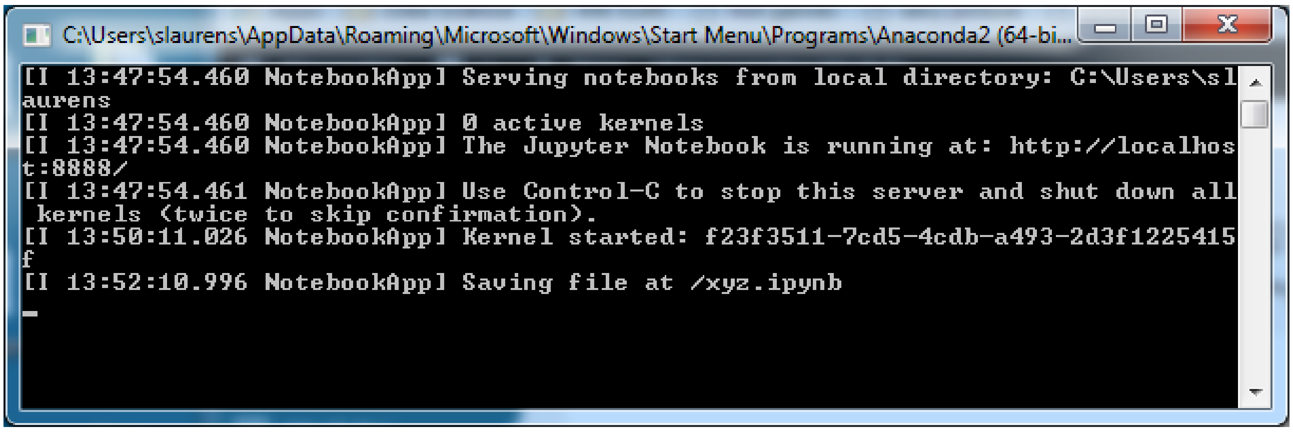
JuPyteR notebooks are also called iPython notebooks (hence the suffix .ipynb) but they're not strictly limited to Python. As the name suggests you can choose from Julia, Python & R (in fact they are becoming 'language agnostic' so the range of language choices is growing). You can install R and Python side by side (as shown below, you can even mix R code with python in the same notebook).



The Jupyter Notebook App can be launched by clicking on the *Jupyter Notebook* icon installed by Anaconda in the start menu, as shown in the following figure.

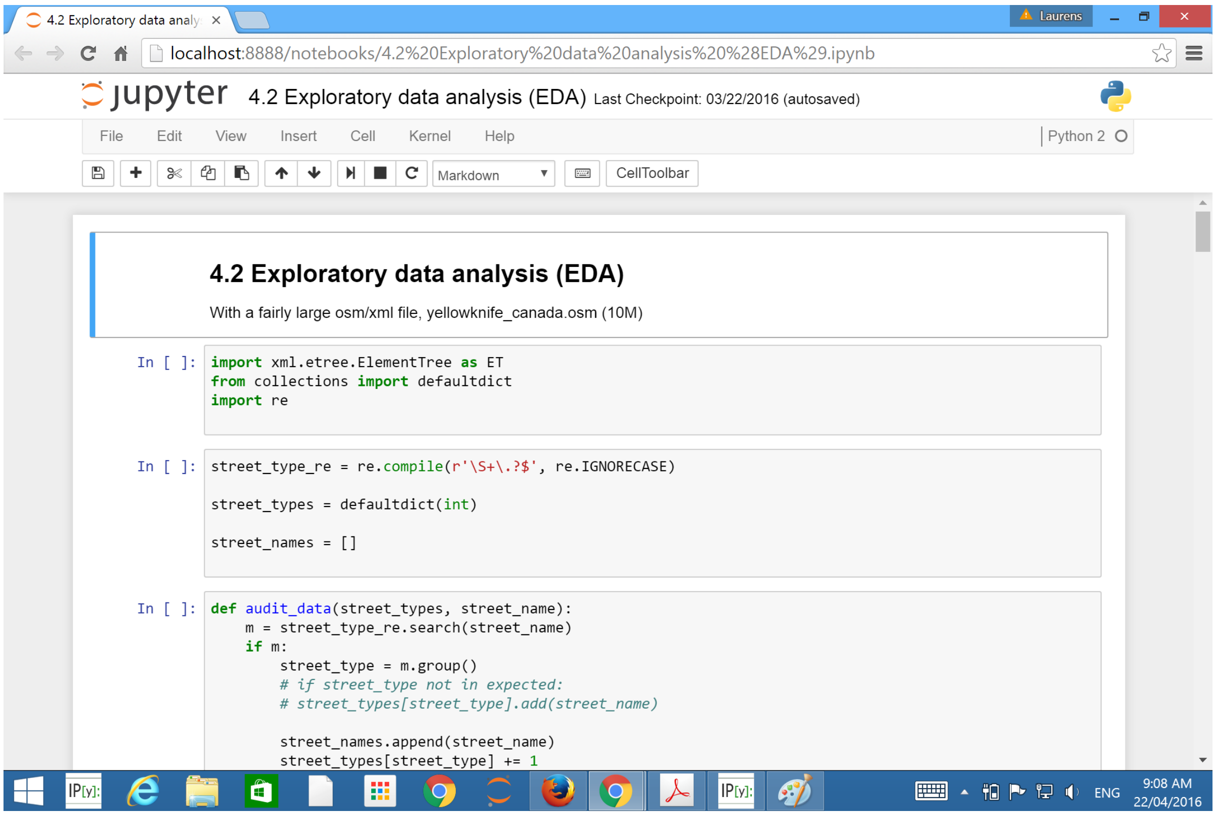


(You may also need the **Anaconda Prompt**, shown above, to install libraries with the commands, 'conda' or 'pip').
After clicking the JuPyteR Notebook icon, it will launch a cmd (command) window as shown in the figure below.


C:\Users\slaurens\AppData\Roaming\Microsoft\Windows\Start Menu\Programs\Anaconda2 (64-bi...

```
[I 13:47:54.460 NotebookApp] Serving notebooks from local directory: C:\Users\sl
aurens
[I 13:47:54.460 NotebookApp] 0 active kernels
[I 13:47:54.460 NotebookApp] The Jupyter Notebook is running at: http://localhos
t:8888/
[I 13:47:54.461 NotebookApp] Use Control-C to stop this server and shut down all
 kernels (twice to skip confirmation).
[I 13:50:11.026 NotebookApp] Kernel started: f23f3511-7cd5-4cdb-a493-2d3f1225415
f
[I 13:52:10.996 NotebookApp] Saving file at /xyz.ipynb
_
```

Which will then launch a new browser window (or tab if browser is already running), which should look like this:


4.2 Exploratory data analy
localhost:8888/notebooks/4.2%20Exploratory%20data%20analysis%20%28EDA%29.ipynb
jupyter
4.2 Exploratory data analysis (EDA) Last Checkpoint: 03/22/2016 (autosaved)
File
Edit
View
Insert
Cell
Kernel
Help
Python 2
Markdown
CellToolbar

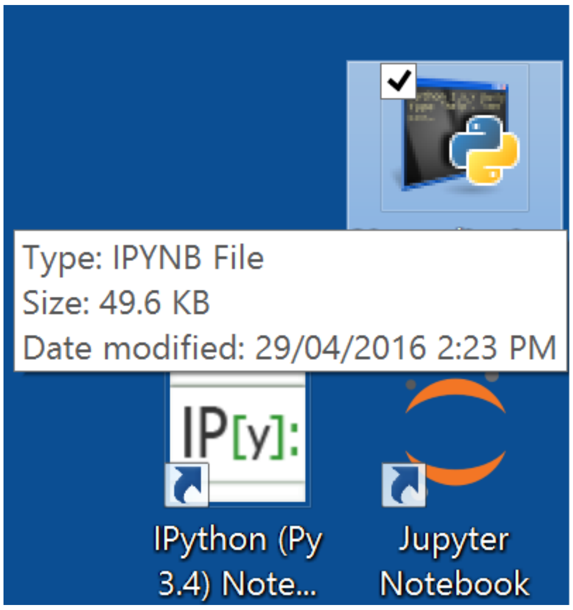
4.2 Exploratory data analysis (EDA)

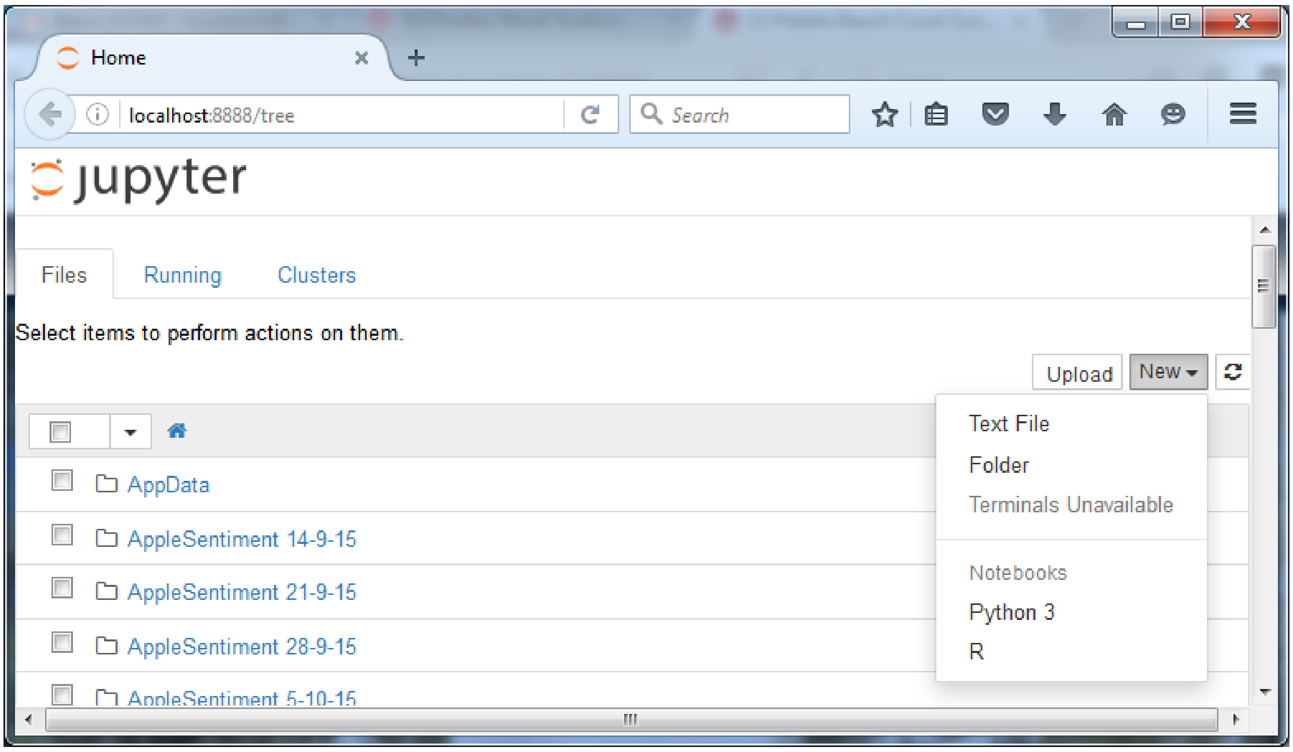
With a fairly large osm/xml file, yellowknife_canada.osm (10M)

```
In [ ]: import xml.etree.ElementTree as ET
        from collections import defaultdict
        import re

In [ ]: street_type_re = re.compile(r'\S+\.?$', re.IGNORECASE)

        street_types = defaultdict(int)

        street_names = []

In [ ]: def audit_data(street_types, street_name):
            m = street_type_re.search(street_name)
            if m:
                street_type = m.group()
                # if street_type not in expected:
                # street_types[street_type].add(street_name)

                street_names.append(street_name)
                street_types[street_type] += 1
```

When started, the JuPyteR Notebook app can access only files within its start-up folder (including any sub-folder). If you store the notebook documents in a subfolder of your user folder no configuration is necessary. Otherwise, you need to choose a folder which will contain all the notebooks and set this as the start-up folder. To do so, you need first to copy the JuPyteR Notebook launcher from the menu to the desktop, and then right-click on the new launcher and change the "Start in" field to the directory where you store your notebook files. Now, double-click on the launcher to start the Jupyter Notebook App. In the launched browser window you should see your notebook files.

Here's another way to open a notebook file (recommended, for Windows): drag a .ipynb notebook (from anywhere) onto the 'Jupyter Notebook' icon (e.g. keep a shortcut on the desktop) to launch. Shown below is a .ipynb being dragged towards two shortcuts (iPython3.4 at left, Jupyter/iPython2.7 at right). In this case iRKernel has been installed on top of Anaconda Python 3.4 so R notebooks can be created, saved and loaded (see following screen shot, 'New' > Python 3, R).





Select items to perform actions on them.

AppData
AppleSentiment 14-9-15
AppleSentiment 21-9-15
AppleSentiment 28-9-15
AppleSentiment 5-10-15

Note that closing the browser (or the tab) will not close the Jupter Notebook App. In order to completely shut it down you should close the associated cmd window.

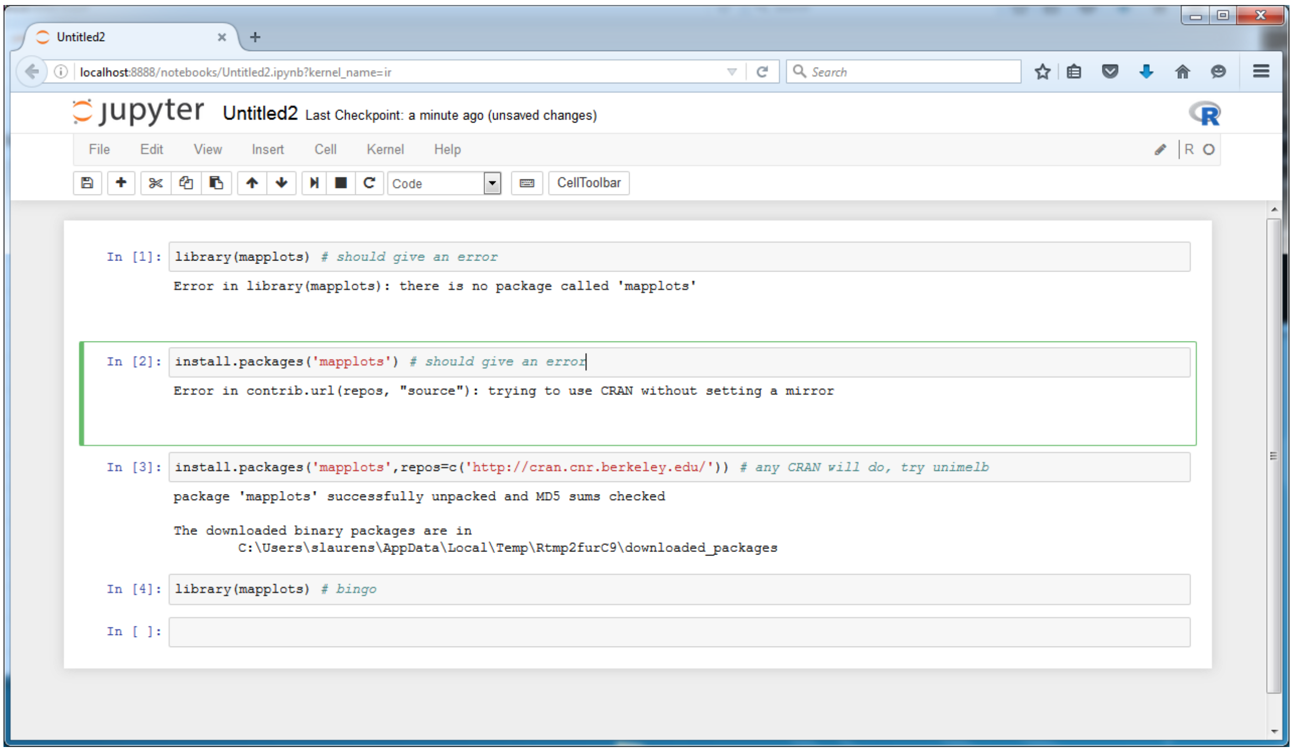
iRkernel - Installing additional packages.

because **iRkernel is sitting on top of Anaconda the usual R command to add packages/libraries *may* not work**

Currently, installing additional packages without any repo (repository) setup may fail with the following error:
Error in contrib.url(repos, "source"): trying to use CRAN without setting a mirror

The default is install.packages(..., repos = getOption('repos')), which evaluates toinstall.packages(..., repos = c(CRAN = '@CRAN@'))

'@CRAN@' means "show a selector", but iRkernel doesn't identify as "interactive" yet, so interactive functions like popping up a repo selector doesn't work. Therefore, you either have to call install.packages(..., repos = c('https://...', ...)) every time, or options(repos = c('https://...', ...)) once in your notebook (before the install.packages call). e.g. (in an R notebook)


Untitled2
localhost:8888/notebooks/Untitled2.ipynb?kernel_name=ir
Search
jupyter
Untitled2 Last Checkpoint: a minute ago (unsaved changes)
File
Edit
View
Insert
Cell
Kernel
Help
R
Code
CellToolbar

```
In [1]: library(mapplots) # should give an error

        Error in library(mapplots): there is no package called 'mapplots'

In [2]: install.packages('mapplots') # should give an error

        Error in contrib.url(repos, "source"): trying to use CRAN without setting a mirror

In [3]: install.packages('mapplots',repos=c('http://cran.cnr.berkeley.edu/')) # any CRAN will do, try unimelb

        package 'mapplots' successfully unpacked and MD5 sums checked

        The downloaded binary packages are in
                C:\Users\slaurens\AppData\Local\Temp\Rtmp2furC9\downloaded_packages

In [4]: library(mapplots) # bingo

In [ ]:
```

Running JuPyteR Notebook on Mac OS

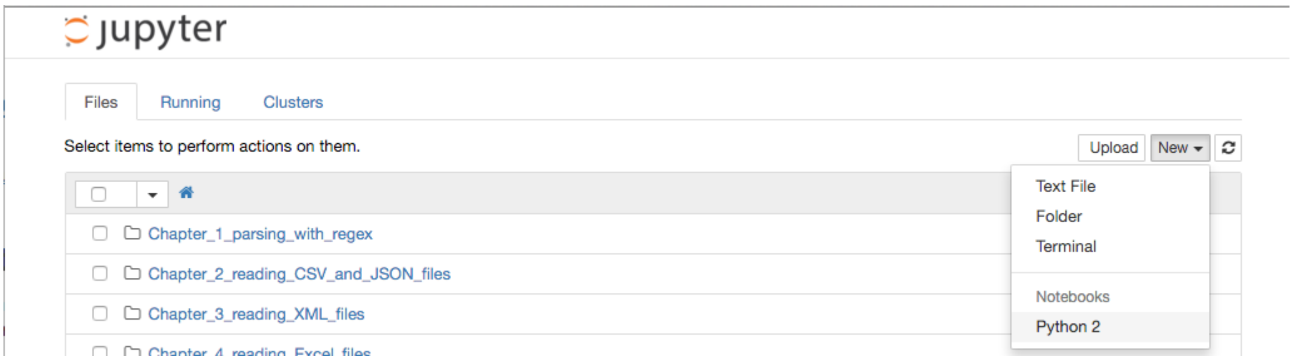
Compared with running JuPyteR Notebook on Windows, it is easier to launch JuPyteR Notebook in Mac OS. You need the following three steps:

1. Open a terminal window. You can find the terminal icon in "Applications/Utilities" .
2. Enter the folder where you store the notebook file(s) by typing "cd /some_folder_name". For example, I have stored all my notebook in "*./MonashTeaching/DataWrangling/module_2_Handling_Machine_Generated_Data*", I then type "*cd ./MonashTeaching/DataWrangling/module_2_Handling_Machine_Generated_Data* ".
3. Type either "JuPyteR notebook" or "ipython notebook" to launch the JuPyteR Notebook App. It will appear in a new browser window or a new tab.

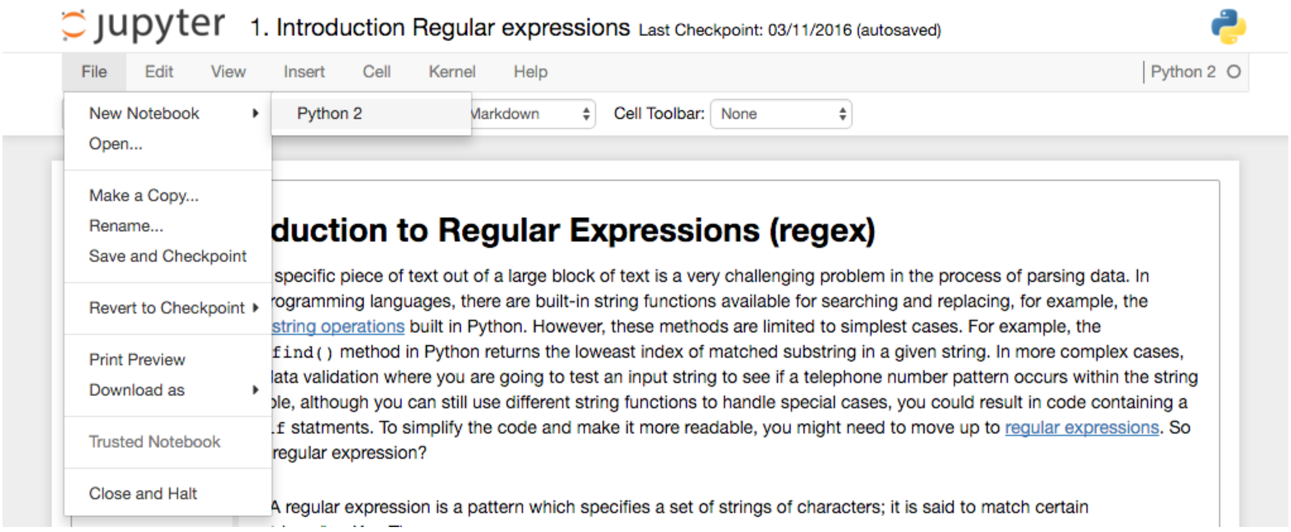
More information about running the JuPyteR Notebook can be found here (https://jupyter-notebook-beginner-guide.readthedocs.org/en/latest/execute.html).

Creating a new JuPyteR Notebook Document

There are two ways of creating a new Jupyter Notebook document. One way is to click "New" on the dashboard. You will then see a drop-down list, where you click "Python 2" or "R", as shown below. The new notebook is created within the same directory and will open in a new browser tab. It will also be reflected as a new entry in the notebook list on the dashboard.


jupyter
Files
Running
Clusters
Select items to perform actions on them.
Upload
New
Chapter_1_parsing_with_regex
Chapter_2_reading_CSV_and_JSON_files
Chapter_3_reading_XML_files
Text File
Folder
Terminal
Notebooks
Python 2

Another way is to use "File > New > Python 2" menu option from within an active notebook as shown below.

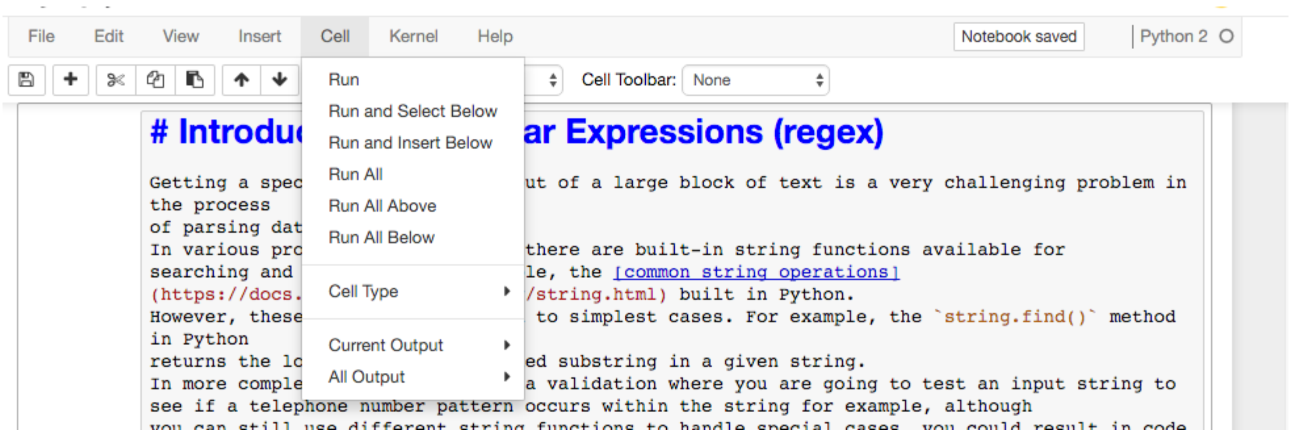


duction to Regular Expressions (regex)

specific piece of text out of a large block of text is a very challenging problem in the process of parsing data. In rogramming languages, there are built-in string functions available for searching and replacing, for example, the string operations built in Python. However, these methods are limited to simplest cases. For example, the `find()` method in Python returns the loweast index of matched substring in a given string. In more complex cases, lata validation where you are going to test an input string to see if a telephone number pattern occurs within the string ple, although you can still use different string functions to handle special cases, you could result in code containing a .f statments. To simplify the code and make it more readable, you might need to move up to regular expressions. So regular expression?

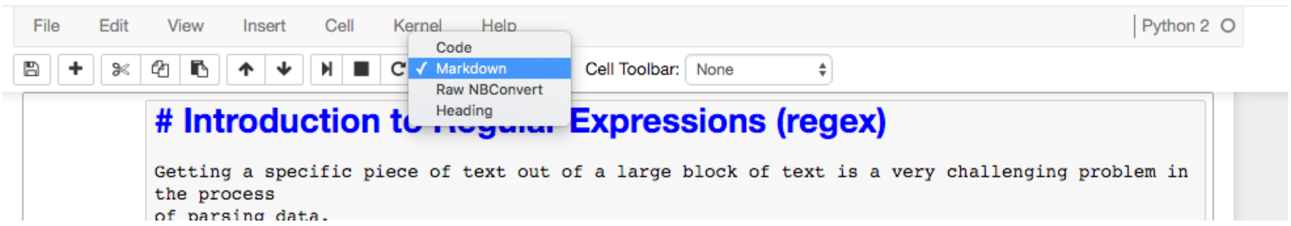
A regular expression is a pattern which specifies a set of strings of characters; it is said to match certain

A notebook usually consists of a sequence of cells. A cell is a multi-line text input field, and its contents can be executed by using `Shift-Enter`, or "`Cell>Run`" in the menu bar.



```
# Introduc                ar Expressions (regex)

Getting a spec            ut of a large block of text is a very challenging problem in
the process
of parsing dat
In various pro            there are built-in string functions available for
searching and             le, the [common string operations]
(https://docs.            /string.html) built in Python.
However, these             to simplest cases. For example, the `string.find()` method
in Python
returns the lo            ed substring in a given string.
In more comple            a validation where you are going to test an input string to
see if a telephone number pattern occurs within the string for example, although
```

The execution behavior of a cell is determined the cell's type. There are four types of cells: code cells, markdown cells, raw cells and heading cells. Every cell starts off being a code cell, but its type can be changed by using a dropdown on the toolbar.



```
# Introduction to Regular Expressions (regex)

Getting a specific piece of text out of a large block of text is a very challenging problem in
the process
of parsing data.
```

The two major cell types that we are going to use are *code cells* and *Markdown cells*. A *code cell* allows you to edit and write new code, with full syntax highlighting and tab completion. With *Markdown cells*, you can document the computational process in a literate way, alternating descriptive text with code, using *rich text*. When a Markdown cell is executed, the Markdown code is converted into the corresponding formatted rich text. Markdown allows arbitrary HTML code for formatting. Some tutorial on using Jupyter Notebook will be introduced in next section.

Introduction to R, Python and JuPyteR Notebook

Congratulations, you have completed your initial setup. Now you need to get yourselves familiar with Python, R and Jupyter Notebook. We will introduce other major libraries that are specific for specific tasks in subsequent modules.

Python and the libraries that we will used usually have very good online documentation, which provides detailed and practitioner-oriented guidelines on how to use them. To save you time, here we suggest the following materials.

Firstly, I would suggest **one** of the following tutorials for learning Python:

- The Python Tutorial (https://docs.python.org/2.7/tutorial/index.html), an official tutorial provided by the Python Organisation. It provides the basics, and offer a gentle tour of the Python language and its standard libraries.
- Google Developers Python Course (https://www.youtube.com/playlist?list=PLfZeRfzhgQzTMgwFVezQbnpc1ck0I6CQl) (recommended for visual learners): it consists of 7 YouTube videos.
- Introduction to data analysis with python (http://www.lynda.com/Numpy-tutorials/Introduction-Data-Analysis-Python/419162-2.html?srchtrk=index%3a1%0alinktypeid%3a2%0aq%3apython%0apage%3a1%0as%3arelevance%0asa%3atrue%0aproducttypeid%3a2) from Lynda.com. To get an access to the videos, please refer to how to use Lynda.com (http://moodle.vle.monash.edu/mod/page/view.php?id=3011058).

There are many free resources available online to learn R with your pace. The followings are just a few examples:

- R tutorials on Tutorials Point (http://www.tutorialspoint.com/r/) and R-Turor (http://www.r-tutor.com/r-introduction)
- R Cookbook (http://www.cookbook-r.com/)for everyone
- An interactive tutorial on Code School (http://tryr.codeschool.com/levels/1/challenges/1)
- he most extensive and updated blog (http://www.r-bloggers.com/) for R developers.

JuPyteR notebook is a web-based application for interactive data science and scientific computing with kernels that provide an interface that combines live-code with narrative text and visualisation. It is particularly useful for interactively working with data. The following online tutorials are a good place to get started with IPython/JuPyteR notebook:

- IPython Notebook Overview (http://cs231n.github.io/ipython-tutorial/) from Stanford
- Running code in Jupyter Notebook

(http://jupyter-notebook.readthedocs.org/en/latest/examples/Notebook/Running%20Code.html): the basics of how to run Jupyter Notebook, which you should know.

- Jupyter Notebook Users Manual (https://athena.brynmawr.edu/jupyter/hub/dblank/public/Jupyter%20Notebook%20Users%20Manual.ipynb) which shows how to view and execute computer programs and to create executable documents or documents with visualisation. It is a good **reference material** while you are writing your notebook.