Introduction to Machine Learning

COMP60012/70050

Rahul George (rg922)

Written & Maintained Autumn 2024

Imperial College London

October 20, 2024

Disclaimer!

Content may be wrong, inaccurate, or otherwise lacking. Don't treat these as a sole resource for studying. All credit goes to the lecturers teaching the module, this was constructed using their resources.

Contents

0 Administrivia

0.1 Numpy Description and Import

Numpy is pretty simple, it's a package for numerical and scientific computing. It uses vectorisation so we can perform operations with multi-dimensional arrays very easily. As with literally any python library, we use:

```
pip import numpy
```

and it should work fine.

Briefly, we use the `np.array` to basically take over the role of a standard python list. The initalisation for this is shown below, as well as several other useful methods which we can operate on them with.

Cheatsheet

Note: A lot of the following may take `axis` *as a parameter. By default, this is 0 and operations are across columns. If we specify axis = 1, we can operate instead across rows.*

Description	**Code Snippet Examples**
Creation & Assignment	`np_arr = np.array(arr)`
Shape of array, from outer to inner	`np_arr.shape`
Number of total elements	`np_arr.size`
Type of element stored	`np_arr.dtype`
Create an array of 0s with given shape	`arr = np.zeros((2, 3))`
Create an array of 1s with given shape	`arr = np.ones((2, 3))`
Create an array of Ns with given shape	`arr = np.full((2, 3), N)`
Create an n-dim Id matrix	`arr = np.identity(n)`
Vectorisation	`arr_3 = (arr_1 * 2 + 3) * arr_2`
Generate evenly spaced numbers	`arr = np.linspace(1, 5, num=5)`
Dot product	`sum = np.dot(arr)`
Boolean operation	`arr_2 = arr_1  > 5`
Indexing	Same as python
Boolean indexing (filtering)	`filtered_y = y[(y <4) & (y >1)]`
Flattening into 1D array	`1d_arr = arr.flatten()`
Reshaping an $M \times N$ array into an $X \times Y$ array	`re_arr = arr.reshape((x, y))`
Reshaping in column-major order	`re_arr = arr.reshape((x, y), order='F')`
Stacking arrays to get a higher dim	`d_arr = np.stack((arr_1, arr_2))`
Create an array of natural numbers ranging n to m	`arr = np.arange(n, m)`
Sum	`arr.sum()`
Standard Deviation	`arr.std()`
Median	`arr.median()`
Cumulative Sum	`arr.cumsum()`
Checking for total truth	`np.all(arr)`
Checking for at least one truth	`np.any(arr)`
Making array unique and getting counts	`uniq_x,cs = np.unique(x, return_counts=True)`
Sort an array	`s_arr = np.sort(arr)`
Get indices that would sort an array	`inds = np.argsort(arr)`
Randomly shuffle array	`np.random.shuffle(arr)`

Table 1: Numpy Cheatsheet

Other useful functions include `argmax, argmin, max, min, round`.

0.2 Matplotlib

The way matplotlib works is that it first creates an empty `Figure` - this contains some number of axes, after which graphs are plotted.

Usage of plt

```
import matplotlib.pyplot as plt

x = np.array([1, 2, 3, 4, 5])
y1 = x * 2
y2 = x * 5

# We can optionally use this if we intend on having multiple graphs.
# Later on, once we have finished defining our first subplot, we can
# call this again to create a new subplot.
plt.subplot()

# We can plot multiple graphs on the same set of axes.
# We can also specify colours as a 3rd parameter.
plt.plot(x, y1, "g")
plt.plot(x, y2, "b")

# We only want to see x and y within a range.
plt.xlim([-10, 10])
plt.ylim([[-40, 40]]

# Labelling and titling.
ply.xlabel("X axis")
plt.ylabel("Y axis")
plt.title("Sample Graph")

# Note that earlier, we used .subplot() if we want more graphs.
# We can now re-use it to create another graph if necessary.
plt.subplot()
...

# Finally, we render the graph on the figure.
plt.show()

# We may also save the graph if necessary.
plt.savefig("test.png")
```

The above example should cover everything that will be used in standard graph plotting with matplotlib, but I've ignored figures because I don't have the energy.

1 The Big Picture

1.1 Introduction

No notes taken, no useful info; generic info.

1.2 AI and machine learning

For the context of this course, we treat AI as techniques that enable computers to mimic human behaviour and intelligence. This could be things like human behaviour, logic, if-then rules. More specifically than this, we consider machine learning to be a subset of AI techniques that uses statistical methods to enable systems to learn and improve with experience. Finally, we consider deep learning as a subset of ML that uses multi-layer artificial neural networks, as well as vast amounts of data.

Assume we have a photo of a cat - we want to write software to tell whether or not a given photo contains a cat or not. Due to the variation of how a cat may look, occlusion, and other factors, this is a vary difficult problem. However, ML trivialises this - we take as input several images, and annotations of whether or not they are cats or not. We then attempt to find hidden or underlying patterns by analysing the given data, in order to use those results to classify unseen data.

1.3 So what *exactly* is machine learning?

Formally, machine learning consists of computer programs that learn from experience for a particular task, and whose performance on that tasks improves with experience. Another nice way to visualise this is by thinking of machine learning as function approximation for some unknown function, which improves as we give it more data.

We can split ML into 3 main paradigms:

a) **Supervised Learning:** Input consists of both variables and their respective output labels. An algorithm should then produce a model that generates an estimated output label.

b) **Unsupervised Learning:** Like supervised learning, but without correct labels. These try and find underlying patterns or hidden structure within the data.

c) **Reinforcement Learning:** We are given variables but no labels, like unsupervised learning. However, an RL algorithm will use estimated output variables to interact with it's environment. This will in turn generate a reward signal which feeds back into the algorithm, improving it in each step. *This is not covered on the course.*

Beyond this, we may have instances of semi-supervised learning or weakly-supervised learning.

1.4 Classification and regression

These are arguably the two most frequent and popular types of ML tasks - classification and regression.
First, classification. We have a classifier, whose job is to take an input and then produce an output label. More formally, a classifier is a function where the input X is some document/image/text, and the output is a label Y - a discrete or categorical label relating to X.

The easiest task is binary classification - our labels are just True and False. Beyond this is multi-class classification, and the classifier has to choose one of many possible labels. The hardest is multi-label classification, where the classifier may have to choose many of many possible correct labels.

Next, regression. We have a regresser function, whose input is the same as a classifier, but the output is a real-valued, continuous floating point number. Consider a regresser which predicts price given house size.

Multiple regression can be used when there are multiple input features, and multivariate regression can be used when we are predicting multiple output labels.

1.5 What have we learnt?

Recall the binary classificaton problem. Let's assume we have an x-y graph and we've plotted two distinct features of our samples on the axes respectively. We can then place our samples on the graph. Now, we can probably attempt to find some sort of line to partition our graph such that everything falling on one side is true, and everything falling on the other side is false.

So we can attempt to find the formula of a straight line, such that we can partition a 2-d space into two different classes. This is formally a **linear classifier**.

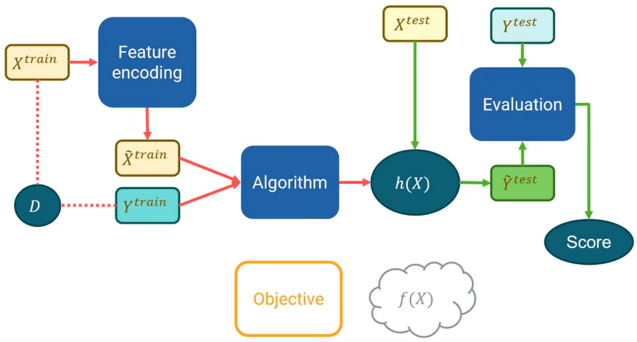
1.6 The supervised learning pipeline

We want to generate a model, that approximates to an unknown function - let's call it h, standing for hypothesis. We say that $h(X) = \hat{Y}$, where the hat over our output label Y indicates that this is an estimate.

An algorithm is used to generate our model. These algorithm will have objective functions, which try to approximate the hypothesis to be as close as possible to the true function. We also have a dataset D, which we assume comes from the unknown true function. The dataset is used as an input to learn. We formally state that D is a sequence of pairs, corresponding to inputs and their output labels: $D = \{(X^{(1)}, Y^{(1)}), (X^{(2)}, Y^{(2)}), ...\}$. We can compact this further by considering $X^{train} = \{X^{(i)}\}^N$ and $Y^{train} = \{Y^{(i)}\}^N$, where N is the training dataset size. At test time, our new examples are just X^{test} and Y^{test}.

We now consider our samples X - it would be much easier to project them to some **feature space**, using feature encoding. This may correspond to changing a category to a numeric value, or normalising some feature. This leaves us with $\hat{X}^{train}$.

We now have a working algorithm which produces a model that should hopefully work - however, we also want to be able to test how well our existing model does on unseen data. We use our test datasets to predict output labels, and then use an evaluation function which compares the "gold-standard" annotations with our predicted ones, resulting in a score of how well our model performed. This is now the complete supervised learning pipeline.



This is one of the most pivotal things to remember, as it can become easy to get stuck with small ML details.

1.7 Pipeline - Feature Encoding

First, we need to understand the data - this may be an image, article, graph, etc. Preliminary examination can be very important, as it can give us clues for designing classifiers.

Data exists in a measurement space, but we want to project it to a feature space. Recall that we have $X^{train} = \{X^{(i)}\}^N$. This consists of N instances of features, and each of these N instances can be represented as $\langle \hat{x}_1^{(i)}, \hat{x}_2^{(i)}, ... \rangle$, some sort of K dimensional vector. Each aspect of this vector can be different data; categorical; integer; real numbers. They may also exist in different ranges, where x_1 exists from 1 to 100, but x_2 from 0 to 1. Therefore, we always try to normalise our data, according to the standardisation formula below:

$$\hat{x}_k^{(i)} = \frac{x_k^{(i)} - \mu_k}{\sigma^k} \tag{1}$$

Now that we have standardised data, we can consider what features to actually use. Intuitively, using more features has a better result, but only up to a certain point; this is the so-called curse of dimensionality. The reason for this problem is that as we grow the number of dimensions, there is increasingly more empty space and our data becomes more sparse. Alongside the increased computational complexity and over-fitting that can occur, we need to be more careful about mapping data to features.

This is where our methods of generating features are important. We can use:

a) **Feature Selection:** Select a subset of the original features, which are more important.

b) **Feature Extraction/Engineering:** Generate a new set of features, perhaps not from the original set.

c) **Automatic:** Use statistical analysis to generate the most important features.

In the era of deep learning, neural networks learn optimal features for the task optimally, which are optimised for particular tasks. This means the entire feature encoding stage as a whole can be skipped.

1.8 Machine learning algorithm

We have two main types of algorithms - lazy and eager learners. A lazy learner will just look at the data and make the decision then and there. There is no training time required, but test time is slower. By contrast, an eager learner spends time training and generalising prior to test time. During test time, the function is already learnt so new samples simply need to be inputted.

We now think about types of model. The first is a non-parametric model - this doesn't assume anything about the inputs. An example of this is the nearest-neighbour (both non-parametric and lazy). There are also linear models, where we assume our data is linearly separable (can be divided by a line, plane or hyperplane). All we are trying to learn is the best line to separate the data. Nonlinear models deal with data which is non-linearly separable - for example, if the function to split the space may be parabolic or sinusoidal. We can combine linear models to produce nonlinear ones.

Finally, we discuss the bias-variance trade-off. This is arguably the most important concept in ML. Assume we try to fit a line to our training data, but it isn't a very good fit. This is an **underfit** which has a **high bias** meaning that it puts in it's own assumptions (for example, assuming linearity when the data is nonlinearly distributed). The opposite is when we have high variance and our model is overfit: this is when our model is too sensitive to the training data. We therefore need to ideally find a sweet spot between variance and bias. This is important to achieve generalisation.

1.9 Pipeline - Evaluation

We go through this at a very high level now, as we go over the content more thoroughly in week 4. We need a metric or measure to determine our success. A common one is accuracy for classification - the number of correct classifications, divided by the number of test instances. For regression, we may use mean squared error. Generally, we can tell whether our model is doing well or not by how much higher it is than a baseline. Often, an absolute lower bound will consist chance or random choice. There may be stronger baselines we can use instead. The main thing about evaluation for now is that it's all comparative.

1.10 Couse roadmap and wrapping up!

No notes taken, no useful info; recap of course structure.

1.11 Notes on Lab 1: Building an Machine Learning Pipeline

This assignment is mainly just testing you across basic numpy concepts, there's nothing especially difficult or weird. The extension isn't really anything.

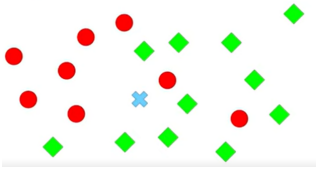
2 K-Nearest Neighbours and Decision Trees

2.1 Introduction to Module 2

No notes taken, no useful info; recap of eager vs lazy seen last week.

2.2 Classification with Instance-based Learning (k-Nearest Neighbours

The main concept is that we will use instances (samples in the training set) to make inferences on a new sample, for example whether it exists in a given class/category or another one. We start with the basic nearest neighbour classifier - it classifies a test instance to a class label of the nearest training instances, according to some distance metric.



We may want to classify in the above example, whether the blue cross belongs to the class of red circles or green squared. It is as simple as determining the nearest shape to be a red circle, so we classify the blue cross as a red circle.

Commonalities between NN and kNN include that they are both non-parametric models - the decision boundary formed by the algorithm is not predefined or imposed; it naturally emerges from the dataset. The main issues with just NN however is that it doesn't account for noisy samples that may be a bit further away from the rest, and it can overfit training data. In the above, there are many more green square in proximity to the cross, but we still choose the red cross.

Instead with kNN, we consider the k nearest neighbours, and choose whichever class appears the most. The only consideration is that k should be odd, so that we never have a tie between two classes. Increasing K will result in a smoother decision boundary, at the risk of potentially incorrectly classifying a few of our cases. It also makes the decision boundaries less sensitive to training data - samples that are slight outliers tend to get ignored.

This lowers the variance of the decision (recall bias-variance-tradeoff from last week), but increases the bias. Choosing a "perfect" value for k depends entirely on the dataset and can only best be found experimentally, through cross-validation.

Now, to determine our neighbours, there are several different distance metrics we can choose from:

a) Manhattan distance (L^1-norm): $d(x^{(i)}, x^{(q)}) = \sum_{k=1}^{K} |x_k^{(i)} - x_k^{(q)}|$

b) Euclidean distance (L^2-norm): $d(x^{(i)}, x^{(q)}) = \sqrt{\sum_{k=1}^{K} (x_k^{(i)} - x_k^{(q)})^2}$

c) Chebyshev distance (L^∞-norm): $d(x^{(i)}, x^{(q)}) = \max_{k=1}^{K} |x_k^{(i)} - x_k^{(q)}|$

We now have a full definition for how to use kNN. However, it makes sense to prioritise neighbours who are closer to our sample, than neighbours who are further away, even if they are in the same neighbourhood. Therefore, we can refine kNN by assigning weights to each niehgbour, based on their distance to the query instance. We then sum the weights per class in the neighbourhood and choose the class with the highest sum. This is **distance weighted kNN**. There are a couple functions we can use to define weight:

Inverse of distance:

$$w^{(i)} = \frac{1}{d(x^{(i)}, x^{(q)})} \tag{2}$$

Gaussian distribution:

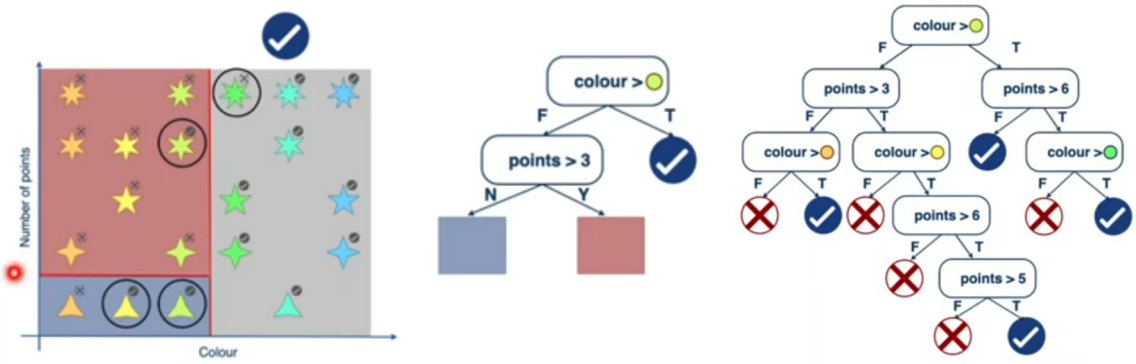
$$w^{(i)} = \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{d(x^{(i)}, x^{(q)})}{2}\right) \tag{3}$$

Good things about distance weighted kNN include that k becomes less important, as far-away examples will have small weights. We can even use $k = N$ as a global method, capturing all the data from every sample. If k is smaller than N, it is called a local method. Lastly, kNN is better with noisy training examples, as it considers more than just one sample.

kNN is very simple, but very powerful. However, it tends to struggle with larger datasets, or with large samples. kNN is especially susceptible to the curse of dimensionality, as it relies on distance metrics to see which points exist in similar neighbourhoods. In higher dimensions, this may not work well as we may have many irrelevant features causing instances to be far apart. A solution is to weight features, or using more strict feature selection. kNN can also be used for regression, by computing the mean value across k nearest neighbours.

2.3 Classification with Decision Trees

This is the principle of trying to focus on a specific subset or feature in one sample, and using them to make finer and finer decisions. We do this across several layers to produce a result about whether or not to include a sample - the easiest way to think about this is a tree of decisions, where the leaves are how we classify a sample. Where NN is a lazy learner, decision trees attempt to learn decision boundaries and are an example of eager learning. See a fully worked example below.



Consider the above diagram. In the first step, we used a vertical line splitting colour into two sets, one of which had mostly ticked samples, and the other of which had mostly crossed samples. The misclassified instances have all been circled. This led to the uppermost node of our decision tree, which tells us to accept if the colour is colder than yellow (this means we fail with one particular sample, but oh well). In the step below stemming from colour being warmer than yellow, we now consider the few samples which are wrongly rejected. This motivates our next decision, which is how many points the samples have. This allows us to instead accept two of the previously rejected triangles. We can go layers deeper and improve the decision tree to be more precise. The final decision tree for this particular example is shown on the right, and an intermediate tree in the middle.

This nicely displays how the decision tree works - it effectively acts as an aggregate of several linear boundaries to partition a sample space, and we assign classes to particular partitions. Therefore, all we need to do for an unclassified sample is determine which partition it lies in, and we can immediately tell its class. At it's core, decision trees act as a set of if-then-else rules, and are great for approximating discrete classification functions.

Decision Tree Algorithm

1. Determine 'optimal' splitting rule on training data.
2. Split dataset according to chosen rule.
3. Repeat 1 & 2 on each new subset until they are small enough to classify.

2.4 How to select the optimal split rule? (information gain/entropy)

Intuitively, we want to partition the dataset into subsets that are as 'pure' as possible, in that most of the samples in one set come from the same class. Several metrics exist for this, but we focus solely on **information gain**, used in the ID3 algorithm.

Before that, we briefly talk about information entropy. It is the measure of uncertainty of a random variable, or the average expected information we would need to fully define a random state. In simple terms, if we have lower entropy, we are more likely to know about an outcome. If there is high entropy, that outcome is less predictable, or more uncertain.

Consider an experiment where we have K boxes, one of which has a key. We are to select which box the key is stored in. We can encode each of our K boxes using binary. Therefore, $I(x) = \log_2 K$, where I(x) is the number of bits we will require in our binary string, and the amount of information we require to fully determine the state of the random variable. In our particular example, we also know that the probability is uniformly

distributed across each box - this means that $P(x) = \frac{1}{K}$. By combining this with the previous equation, we have the following:

$$I(x) = \log_2 \frac{1}{P(x)} = -\log_2 P(x) \tag{4}$$

Imagine then if the probability P(x) of finding the key in a particular box increases drastically - this causes I(x) to decrease substantially, as we have much lower entropy (the result is less uncertain and more predictable). If we then consider the sum of the the entropy for each of our probabilistic cases, we get the following sum defining entropy. Below it, we define a similar definition for continuous cases:

$$H(X) = -\sum_{k}^{K} P(x_k) \log_2(P(x_k)) \tag{5}$$

$$H(X) = -\int_{x} f(x) \log_2(f(x)) \tag{6}$$

Whenever entropy is close to 1, we need a maximum amount of information to determine our random state. When it is 0, the outcome is decided and predictable.

We now consider how to use it to split our dataset. First, we can compute the entropy of our dataset, by considering the probability of randomly selecting one of our classes. Once we use that to make a first decision, we may split into two branches. To calculate the entropy for this new layer of the tree, we can calculate the individual entropies of each subset, and get an average overall entropy by multiplying those values by the fraction of the overall set they consist. Assume we split into D_0 of size 4, where $H(D_0) = 0$. Then D_1 of size 16, where $H(D_1) = 0.8960$. All we then need is $\frac{4}{20} * 0 + \frac{16}{20} * 0.8960$. With each step, our entropy for each layer should be reducing - this is called the information gain, the decrease in entropy between two layers. So then if our entropy reduces from 0.9 to 0.6, we have an information gain of 0.3 in that step. *Note: there's an equation for this, but it's so much easier to just understand it.*

2.5 Worked example for constructing decision tree (categorical attributes)

When selecting which attribute to split on, we calculate the information gain for each, and then choose the one with the highest value. This requires calculating the entropy for the entire dataset (based on the final classification), and then 1 layer down, the weighted average of the entropies of each of our categorical cases.

Once we've identified an attribute to split over, we simply split the dataset into each case for that particular attribute. So if we have 4 cases, we'll end up with four subsets, where each subset has the same value for the chosen attribute. If any of the subsets are perfectly pure (all have the same classification), we can immediately say that if we encounter that particular attribute, we can classify it, giving us a leaf in our tree. Otherwise, we now need to go work out the new best attribute to split on like before (within the subset), and repeat until all branches of our tree end with leaves (pure subsets).

2.6 Summary and further considerations

As with all ML algorithms, decision trees can overfit. To deal with this, we have a couple simple methods to deal with this - either early stopping (where we set a maximum depth, or minimum subset size), or pruning.

We go into pruning in more detail, and describe an algorithm for it:

Pruning

```
PRE: Split data into train and validation set.

1. Identify branches which are connected to 2 leaves.
2. Replace these with leaf nodes, with the majority class leaf as the new class.
3. Evaluate the pruned tree on the validation set.
    - If pruned accuracy is higher, use this as the new tree.
4. Repeat until all such leaf nodes have been tested.
```

The last thing to mention are random forests, where we have several decision trees generated with random samples of the training set, and random subsets of features. We then take the majority of each tree in the algorithm.

2.7 Notes on Lab 2: K-Nearest Neighbours

Nothing in particular, but a couple fun things. Here's a one-line solution for implementing the kNN predict method. It's a bit long, but works with 96%+ accuracy:

1-lined kNN

```
def predict(self, x):
    return np.array([(Counter(self.y[np.argsort(np.linalg.norm(self.x - i, axis=1))
                                [:self.k]]).most_common(1)[0][0]) for i in x])
```

Here was my code for distance weighted kNN:

Distance Weighted kNN

```
ls = []

for i in x:
  cds = {i:0 for i in y}
  for c, t in enumerate(self.x): cds[self.y[c]] += (1/np.linalg.norm(i - t))
  ls.append(max(cds, key=cds.get))

return np.array(ls)
```

2.8 Exercise Recommendations

Personally, I'd do the kNN question (very easy and there's only one), skip the first 2 DT questions, and do the 3rd DT question (also pretty easy, just be careful about what you're splitting on). The middle 2 DT questions are super long (as in, 1+ hours each of just calculations).

3 Machine Learning Evaluation

3.1 Introduction to Module 3

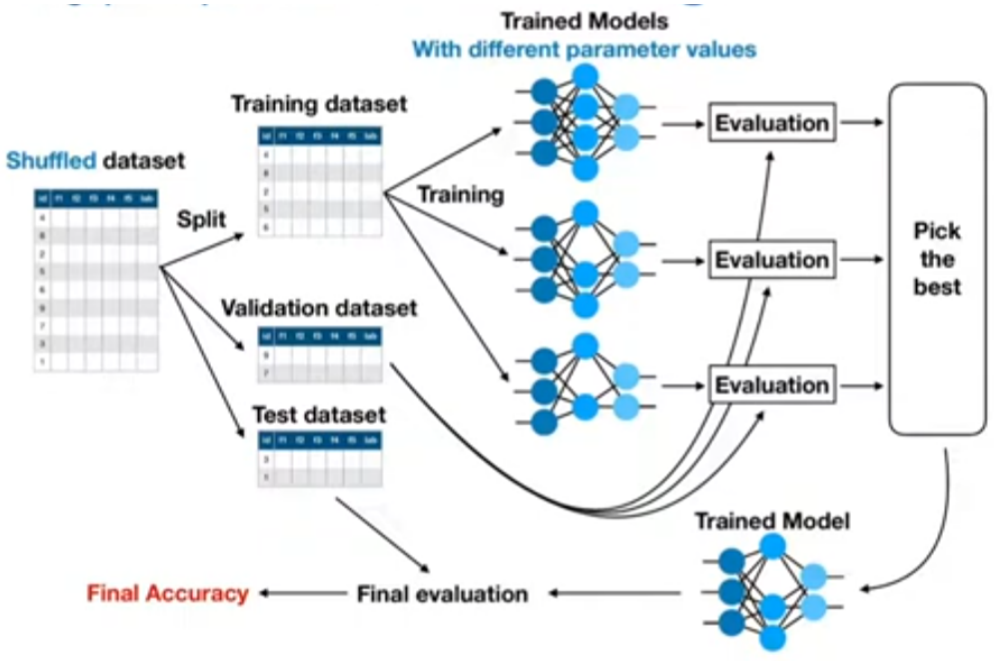
The main focus of this section is how to compare different models, and determine which one is the best to use. *No other useful info.*

3.2 Evaluation set-up

The ultimate goal is to come up with models that generalise to unseen data, as we know - to check this, we use a held-out test dataset. This is never used to train the model, and exclusively to measure model accuracy. As part of this, hyperparameter tuning becomes quite important; a hyperparameter refers to a model parameter chosen before training, like k in kNN. A new objective is to find the hyperparameter which leads to the best performance.

A naive approach is just to try a bunch of different, random values, and select the best one according to its accuracy after evaluation. However, this doesn't generalise well as it can cause overfitting to the training data. We should also not be selecting the best hyperparameters based on performance on the test set, as the test dataset now appears as part of training which is really bad.

Therefore, we instead now split our dataset into 3 sections - training, validation, and testing. Common splits are 60/20/20 and 80/10/10. Here, we try out hyperparameters on the training dataset, and select the best one according to accuracy on the validation dataset. The final evaluation is performed on the test dataset. The entire flow is shown below:



This process of choosing a classifier for maximum performance on a validation set is hyperparameter tuning.

3.3 Cross-validation

When performing MLE, we should ideally be using a train, dev and test set. Sometimes, we may not be able to do this if our datasets are particularly small. Here, instead of diving our dataset once, we divide it several times and create many different splits (or folds). Often, we split into 10 folds, using 8 for training, 1 for validation, and 1 for testing. We iterate through each of our folds, and in each iteration, use different ones as the dev and test sets, and the remaining eight as train sets. Thus, by the time we're done, we'll have tested on all the data,

but never trained on that same data at the same time (which is fine).

An issue with the above approach is that the validation split for tuning hyperparameters is very small. An alternative approach is to do cross-validation, within cross-validation. At each step of folding, assuming k folds, we separate 1 for testing. We then go through k-1 iterations, where we use k-2 of the remaining folds to train, and 1 to tune. This is extremely computationally expensive, but is the best for achieving a good result.

Regardless of the type of cross-validation we do, we can then average their performance metrics using global error estimate: $\frac{1}{N}\sum_{i=1}^{N} e_i$. This will give us a good indication of how an algorithm performs. As a note, cross-validation is not really a method for evaluating a particular model, but more a particular algorithm.

A final couple notes to mention is that after we've done tuning, we may opt to retrain our model on the entire dataset, including the dev/tests sets. This can slightly improve performance since we have another 20% of data to test on, especially if we're sending a model into production. The only problem is that now, we can't re-test the final model.

3.4 Evaluation metrics

This is the way through which we can actually evaluate models and compare them, using a score. We start by looking at confusion matrices. These are used to represent information, in order to easily analyse them. The rows of our table consist of actual class labels, and the columns consist of the predicted labels. Assume for now these are true and false. This means that the intersection of row 1 and column 1 will be classes that were predicted and labelled true - this is good. Similarly, the intersection of row 2 and column 2 will be classes that were predicted and labelled false - this is also good. However, if we take the intersection of row 1 and column 2, these would be things that were predicted false, which were actually true - false negatives. Correspondingly, we have false positives for the intersection of row 2 and column 1. See a diagram below:

	Class 1 Predicted	Class 2 Predicted
Class 1 Actual	**TP**: True Positive	**FN**: False Negative
Class 2 Actual	**FP**: False Positive	**TN**: True Negative

We can use our confusion matrix, to calculate several metrics:

a) **Accuracy:** $\frac{TP+TN}{TP+TN+FP+FN}$. %age of correctly classified examples. Classification error is just 1 - this.

b) **Precision:** $\frac{TP}{TP+FP}$. If a high precision system diagnoses a positive, it is almost definitely positive. However, it may incorrectly predict positives labels for negatives.

c) **Recall:** $\frac{TP}{TP+FN}$. This means it is really good at finding all the positive cases, but it may be predicting the positive label more often than it should.

d) **Macro-averaged Recall:** The average of the recalls for each class.

e) **F-measure:** $\frac{2\times precision\times recall}{precision+recall}$. The harmonic mean/an averaged version that considers both precision and recall.

Each of these generalise fairly easily when we have a multi-class confusion matrix. We just use slightly more interesting labels than just true and false.

For regression tasks, we just have one main metric, mean-squared error. Lower values are better, as it means our regression line is closer to the points on average. See the equation:

$$\frac{1}{N}\sum_{i=1}^{N}(Y_i - \hat{Y}_i)^2 \tag{7}$$

Lastly, although accuracy is a good metric, there are other things to consider. Models should be fast to train and query, scalable in that they should work with large datasets, simple to understand, and interpretable.

3.5 Imbalanced data distribution

In a balanced dataset, there are an equal number of both positive and negative samples (or similar number of instances for each class). If we have an imbalanced dataset, this can cause our metrics to become unreliable. There's no great way of dealing with this, but a reasonable solution is to try and normalise our counts - in other words, dividing the number of correct and incorrect predictions we have by the total number of predictions for that class. We can also try and change our sampling such that we drop samples to get the same number in each class, or duplicate samples to achieve the same goal. In practice though, since we won't apply a system to a balanced dataset IRL, none of these are necessarily the best way of doing things.

3.6 Overfitting

As we've already seen, overfitting is when we have good performance on training data, but poor generalisation on held-out data. Underfitting is when we just have poor performance in general, on both training and testing data.

Typical reasons for overfitting include using models that are too complex, when we have examples in the training set which are outliers/not representative of all situations, and if learning is performed for too long (this is specific to neural networks). In order to combat these, we can get more data, stop training earlier, and opt for simpler models.

3.7 Confidence intervals

These are a way of quantifying our confidence in a particular evaluation result. As seen with imbalanced data, we don't know how much we can trust our performance metrics. We may also have a very small test set, which means our evaluation isn't necessarily reliable either. For example, accuracy of 84% with 10000 samples is far better than accuracy of 90% with 10 samples.

Formally, we define the **sample error** of a model using the following.:

$$error_s(h) \equiv \frac{1}{N}\sum_{x \in S}\delta(f(x), h(x)) \tag{8}$$

For any samples drawn from distribution D, we always want to know the true error. Since we can only ever know the sample error, we provide it with an N% confidence interval. For a given parameter q, this is an interval within which we expect the true value of q to lie, with probability N%. For example, if we have the 95% confidence interval [0.2, 0.4] for q, we can say there is a 95% probability q is between that range. For a sample S with more than 30 cases, we can also say with N% confidence that the true error lies in the interval:

$$error_s(h) \pm Z_N \sqrt{\frac{error_s(h) \times (1 - error_s(h))}{n}} \tag{9}$$

The Z_n is a scaling factor for the desired confidence level. The thing we're multiplying it by is an estimated standard deviation of the sample error. Note that lowercase n is the size of the sample - an interesting thing to note is that as n goes to infinity, the uncertainty becomes negligible and we converge to the true error. In practice obviously, this never happens.

3.8 Testing for statistical significance

Imagine we have two classifiers, both evaluated on a large number of datasets. We can plot out a distribution of their results, over all the experiments. We may need to run a statistical test to see if there is a difference between the two distributions, if they appear to be very similar.

As we saw with stats last year, we have a null hypothesis H_0, that two algorithms or models are the same. Our test will return a p-value, which gives us the probability of obtaining the observed behaviour of the models assuming the null hypothesis. With a smalll p-value, this means that H_0 will be unlikely, and we can instead assume the two models are different. We generally consider it statistically significant if $p < 0.05$.

P-hacking is the misuse of data analysis, where we find correlations or patterns in data that appear significant, but have no underlying effect. To work against this, we use an adaptive P-value suggested by Benjamini & Hochberg. We first rank our p-values from M experiments, and then calculate BH critical values for each. In cases where the p-value is smaller than the critical value, results are statiscally significant.

3.9 Notes on Lab 3: Machine Learning Evaluation

This one was pretty chill. Here's my confusion matrix code:

Confusion Matrix

```
for n, pred in enumerate(y_prediction):
  act = y_gold[n]
  confusion[act][pred] += 1
```

I was going to include other code snippets, but they pretty much just match the sample solution if for a couple shortenings. This lab was really easy as well, not much to see.

3.10 Exercise Recommendations

Q1: we have enough data that can probably just do a train-dev-test split of 80/10/10, with training, tuning and then held-out testing. We can finally retrain on the entire dataset, after both the tuning and testing stages, to regain 20% of the data. Q2: we would now probably want to do cross-validation with 10 folds - this will help us choose the best model, and guide info about metrics and hyperparameters, despite a lack of data. Q4: MSE (regression problem), precision and f-score (both precision and recall are relevant), accuracy (standard classification problem). Q5: we have probably overfit, so we can decrease complexity and increase training data. Q6: we are probably underfitting, so we can increase complexity, train for longer, or use more relevant features. Q7: data may have accidentally been reused in the train and dev sets, so we should be careful and validate this

- it may also be worth cross-validating. Q9: there was a $> 5\%$ chance that they would produce the results that they did, assuming they came from the same distribution.

Q3 and Q8 are just maths, and they're not too hard.