CO395 - Introduction to Machine Learning (70050)

Week 2 (Introduction to ML)

Week 3 (Instance-based Learning + Decision Trees)

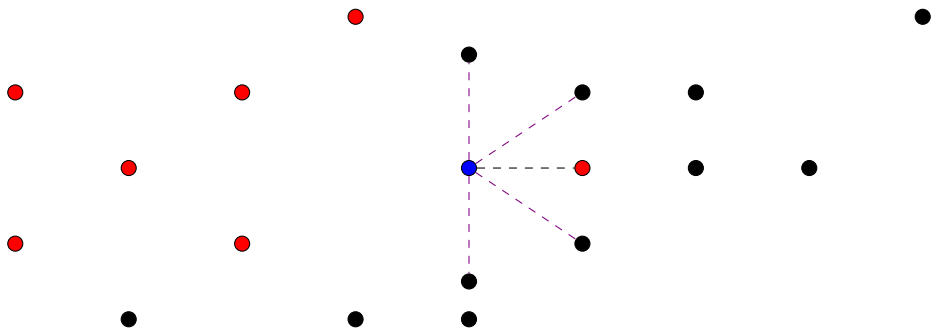
The **k Nearest Neighbours (k-NN)** classifier is classified as a **lazy learner**. A lazy learner stores all the training examples in the data set, and postpone any processing until a request is made (such as a prediction). On the other hand, **decision trees** are classified as a **eager learner**. An eager learner will attempt to construct a general target decision function, which is prepared prior to a query being made.

Classification with Instance-based Learning

The concept behind instance-based learning is that we will use samples in a training data set in order to make inference on a query.

The **Nearest Neighbour** classifier is a specific example, where it classifies a test instance to the label of the nearest training instance, where nearest is subject to some distance metric. This is a **non-parametric model**, which means it naturally emerges from the training set. Note in the example below, an issue with this is that it can be sensitive to noise, as it would classify the blue point to be red, as it is the closest instance in the training set, even though it's more likely to be black - it is very sensitive to noise, and can **overfit** to the training data.

On the other hand, if we consider the **k Nearest Neighbours**, highlighted by the lines in violet, we get the class to be black, as we have 4 against 1. Usually, we need k to be odd, to ensure a winner for the decision task.



Increasing k will give the classifier have a smoother decision boundary (higher bias), and less sensitive to training data (lower variance). Choosing k is dependant on the dataset, normally with a validation dataset.

The distance metric can be defined in many different ways, including the ℓ_1, ℓ_2 and ℓ_∞-norms as seen in **CO233**. Other metrics exist such as the **Mahalanobis distance** for non-isotropic spaces, typically used for Gaussian distributions, or the **Hamming distance** for binary strings.

Another variation is the **Distance Weighted k-NN**. For example, we may not want to trust neighbours which are further away, such as in the example below.

The idea is that we add weights to each neighbour (depending on distance), typically a higher weight for closer neighbours. We then assign the class based on which class has the largest sum. This metric, $w^{(i)}$, is any measure favouring the votes of nearby neighbours, such as;

- inverse of distance

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

- Gaussian distribution

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

The value of k is less important in the weighted case, as distant examples won't greatly affect classification. If $k = N$, where N is the size of the training set, it is a global method, otherwise it is a local method (only considering the samples close by). This method is also more robust to noisy training data, however it can be slow for large datasets.

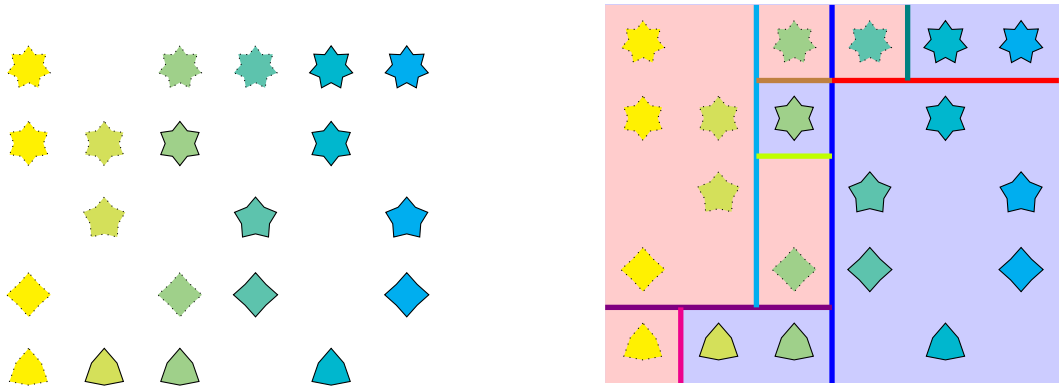
As this method relies on distance metrics, it may not work well if using all features in high dimensional spaces. If these features are irrelevant, instances in the same class may be far from each other. One solution to this is to weight features differently.

k-NN can also be used for regression, either by computing the mean value across k nearest neighbours (which leads to a very rough curve), or by using locally weighted regression, which computes the weighted mean value across k nearest neighbours, leading to a smoother curve.

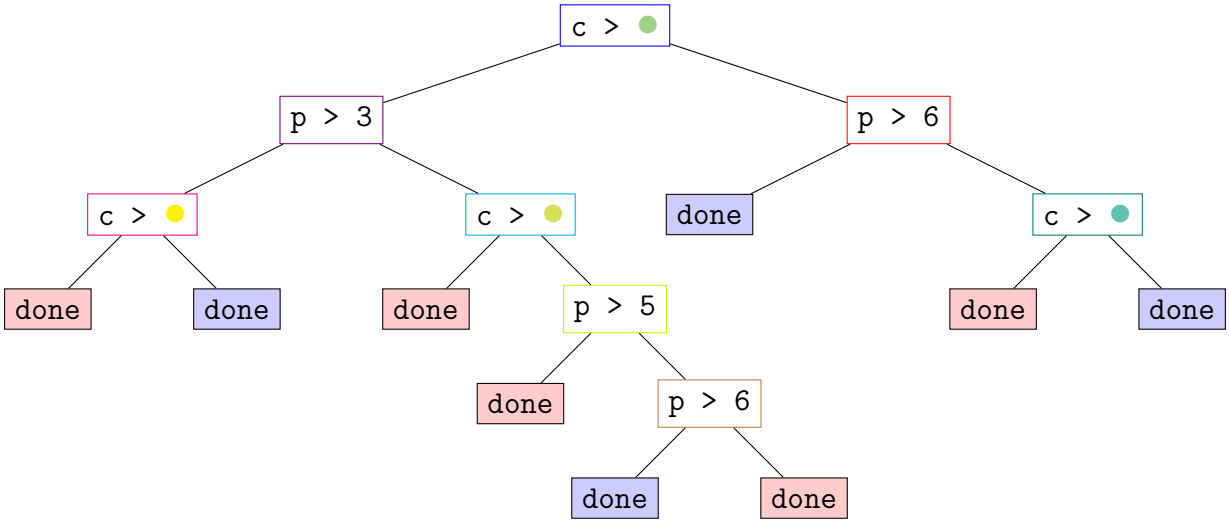
Classification with Decision Trees

Decisions trees are the principal of focusing on a subset or single feature of each sample and then make a decision whether it's true or false (for each feature), and repeat this process to finer decisions until we manage to classify the sample that we want to check.

In decision trees, we learn a succession of linear decision boundaries that we can use to eventually correctly classify samples.



In the example above, we repeatedly choose divisions that result in the fewest number of errors, until we are able to classify everything. This results in the following decision tree, when we are using the attributes of colour and number of points. For brevity, the left branch is the `false` branch, `p` means points, and `c` means colour.



Decision trees are a method of approximating discrete classification functions, by representing them as a tree (a set of if-then rules). The general algorithm (ID3) for constructing a decision tree is as follows;

1. search for the optimal splitting rule on training data
2. split data according to rule
3. repeat 1 and 2 on each subset until each subset is pure (only containing a single class)

How to select the 'optimal' split rule

Intuitively, we want to partition the datasets such that they are more pure than the original set. To do this, we have several metrics;

- **Information gain** ID3, C4.5

  quantifies the reduction of **entropy**

- **Gini impurity** CART

  if we randomly select a point in the feature space and randomly classify it according to the class label distribution, what is our probability of getting it incorrect?

- **Variance reduction** CART

  mostly used for regression trees, with a continuous target variable

To do this, we need to understand information entropy. Entropy is a measure of uncertainty of a random variable. It can also be seen as the average amount of information needed to define a random state / variable. If something has low entropy, it's predictable, and vice versa for high entropy.

Imagine we have two boxes, with something stored in one of the two, with an equal probability in each. To be fully certain, we need a single bit of information, if it's in the left box, the bit is `0`, otherwise (if it's in the right box), it's `1`. Similarly, if we have four boxes, with a uniform distribution, we would need 4 bits to encode the 4 states. In general;

$$
\begin{aligned}
2^B &= K \text{ states} \\
B &= \log_2(K) \\
I(x) &= \log_2(K) && \text{amount of information to determine the state of a random variable} \\
P(x) &= \frac{1}{K} && \Rightarrow \\
K &= \frac{1}{K} && \Rightarrow \\
I(x) &= -\log_2(P(x))
\end{aligned}
$$

As such, we can say;

$$I(x = \text{box}_1) = I(x = \text{box}_2) = I(x = \text{box}_3) = I(x = \text{box}_4) = -\log_2(P(x)) = 2 \text{ bits}$$

However, assume a non-uniform distribution, with the probabilities being 97%, 1%, 1%, and 1% respectively. If we were told it was in box 1, we do not get a lot of new information (low entropy); however if we were told it was in one of the other three, we high entropy (represents very important information).

$$\begin{aligned} I(x = \text{box}_1) &= -\log_2(0.97) \\ &\approx 0.0439 \text{ bits} \\ I(x = \text{box}_2) &= -\log_2(0.1) \\ &\approx 6.6439 \text{ bits} \end{aligned}$$

Entropy is defined as the average amount of information;

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

In our example, we therefore have;

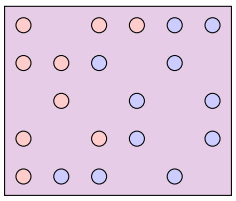
$$H(X) = -(0.97 \cdot \log_2(0.97) + 0.01 \cdot \log_2(0.01) + 0.01 \cdot \log_2(0.01) + 0.01 \cdot \log_2(0.01)) \approx 0.2419 \text{ bits}$$

We therefore need, on average, less information to know where the key is (compared to the uniform distribution).

For continuous entropy, we can use the probability density function $f(x)$ - this is imperfect (it can have negative values), but is still often used in Deep Learning.;

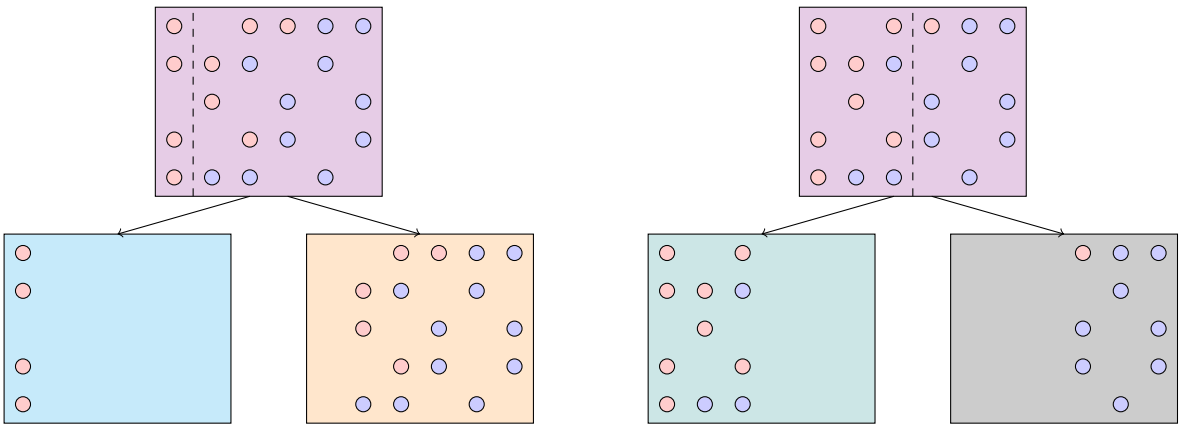
$$H(X) = -\int_{x} f(x) \log_2(f(x)) \, \mathrm{d}x$$

Consider the following example;



$$\begin{aligned} P(\bullet) &= \frac{11}{20} \\ P(\bullet) &= \frac{9}{20} \\ H(\blacksquare) &= -\left(\frac{11}{20} \cdot \log_2\left(\frac{11}{20}\right) + \frac{9}{20} \cdot \log_2\left(\frac{9}{20}\right)\right) \\ &\approx 0.9928 \end{aligned}$$

An entropy value close to 1 would indicate a maximum amount of information needed.



$$
\begin{aligned}
H(\blacksquare) &= 0 \\
H(\blacksquare) &\approx 0.896 \\
H(\{\blacksquare, \blacksquare\}) &\approx \frac{4}{20} \cdot 0 + \frac{16}{20} \cdot 0.896 \\
&\approx 0.7168 \\
H(\blacksquare) - H(\{\blacksquare, \blacksquare\}) &\approx 0.276 \qquad \text{information gain}
\end{aligned}
$$

$$
\begin{aligned}
H(\blacksquare) &\approx 0.8454 \\
H(\blacksquare) &\approx 0.5033 \\
H(\{\blacksquare, \blacksquare\}) &\approx \frac{11}{20} \cdot 0.8454 + \frac{9}{20} \cdot 0.5033 \\
&\approx 0.6915 \\
H(\blacksquare) - H(\{\blacksquare, \blacksquare\}) &\approx 0.3013 \qquad \text{information gain}
\end{aligned}
$$

As the second split has the larger information gain, that is the one we will end up selecting (and generally we want to split to maximise information gain). A formulation of this is as follows;

$$
IG(\text{dataset}, \text{subsets}) = H(\text{dataset}) - \sum_{S \in \text{subsets}} \frac{\mid S \mid}{\mid \text{dataset} \mid} H(S)
$$

$$
\mid \text{dataset} \mid = \sum_{S \in \text{subsets}} \mid S \mid
$$

We can have the following types of input;

- **ordered values**
  - attribute and split point
  - for each attribute, sort the values and consider split points between two examples with different classes
- **categorical / symbolic values**
  - search for the most informative feature and create as many branches as there are values for this feature

Worked example for construction decision tree

Skipped, as this is basically done for the coursework.

Summary and other considerations with decision tree

Note that in general, if we have real-valued attributes, we will end up with a binary tree, with an attribute and threshold at each node. On the other hand, if we have categorical values, we can end up with a **multiway tree**.

Decision trees will **overfit**, like with many machine learning algorithms. This means the algorithm will take into account every sample in the dataset, to the point where it picks up the noise in the dataset. On the other hand, we have an underfitted algorithm, which has low variance and high bias (in contrast).

In decision trees, to deal with overfitting, we can employ the following strategies;

- **early stopping**

  basically stop the algorithm when a condition is met, rather than when the subset is pure (such as maximum depth of tree, or a minimum number of examples in the subset)

- **pruning** will be covered more next week

  1. identify internal nodes connected to only leaf nodes
  2. turn each into a leaf node (with the majority class label)
  3. if the validation accuracy of the pruned tree is greater, we keep it, and then repeat the process until no other pruning can improve the accuracy

To test this, we can reserve part of the dataset for training, and another part for validation. This is called **cross-validation**.

Another approach is to use a random forest. This involves training multiple decision trees, each with a subset of the training dataset, with a random subset of the features, and therefore each focuses on one subset of the features. We then take the majority vote by each of the decision trees as the final outcome.

Decision trees can also be used for regression (**regression trees**). Instead of class labels, each leaf node predicts a real-valued number.

Week 4 (Machine Learning Evaluation)

Evaluation Set-up

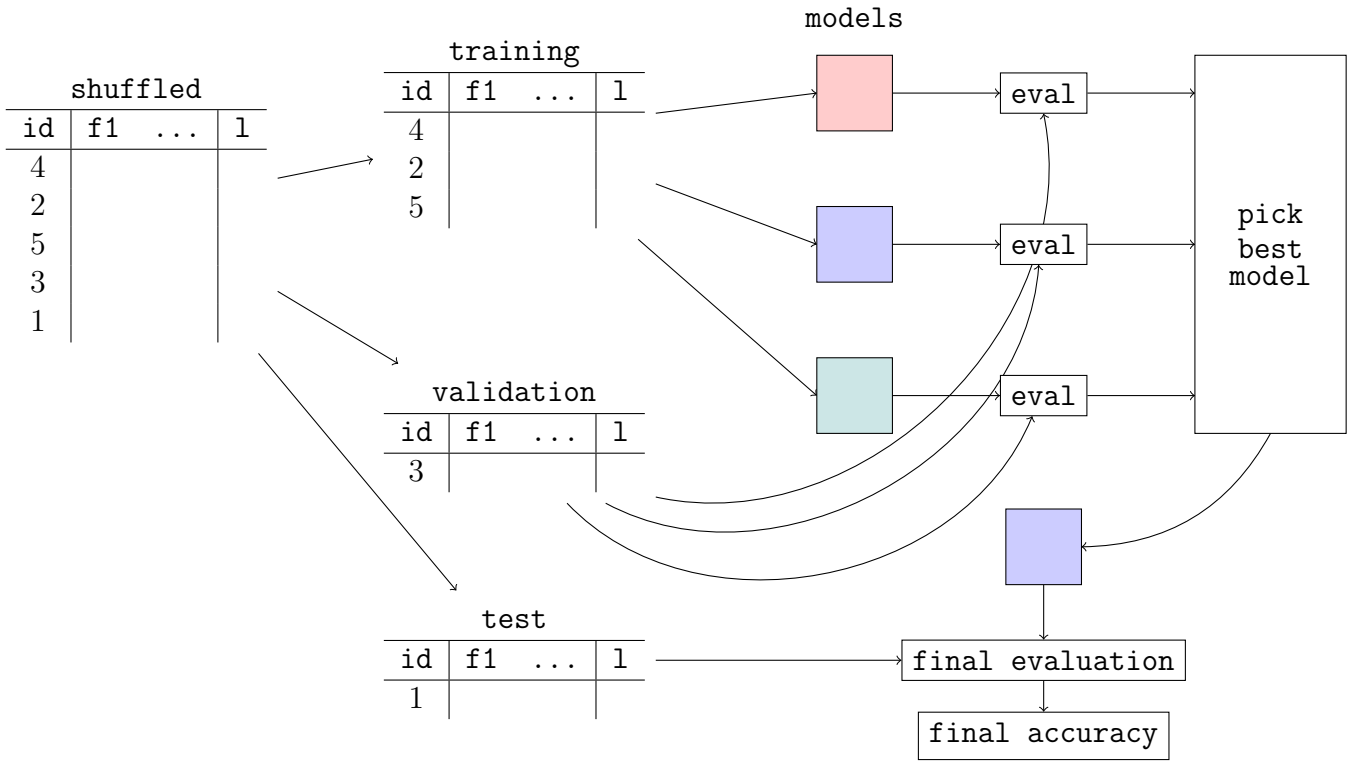
The goal is to create models and algorithms that can generalise to unseen data. We have good accuracy for the training set, since we trained the model for that, however we care more about the accuracy of unknown data.

To ensure meaningful evaluation, we need to split the training dataset from the test dataset (the test dataset should **never** be used to train, as it needs to simulate unknown data). This is done by first shuffling the dataset, and then splitting it into training and test datasets. The training dataset is used to train the model, and the test dataset is then fed into the trained model for final evaluation.

Hyperparameters are model parameters chosen before the training, such as the k value for k-NN algorithm. Our overall objective is to find the values that lead to best performance for unknown data. An incorrect approach for hyperparameter tuning is to try different values on the training dataset, and then select the ones that lead to the best accuracy on the test dataset. This is incorrect because we now use the test dataset as part of the training process, and therefore we cannot say that it is unknown.

As such, the correct approach is to split the dataset into 3; training, validation, and test. The splits for this are between somewhere between $60:20:20$ and $80:10:10$. Different hyperparameter values are attempted on the **training** set, and then the result with the best accuracy on the **validation** set is chosen. The final evaluation is still done on the **test** dataset.

We want to keep the classifier that leads to the maximum performance on the validation test. We can extend this even further by adding the validation dataset to the training dataset and training it on the model with the best parameters to give it more data. Once again, the final evaluation is still done with the test dataset.



Cross-validation

The idea of cross-validation is that the dataset can be divided into k (usually 10) equal splits. $k-1$ of these folds can be used for training and validation, and the remaining split can be used for testing. This is done k times, each time testing on a different portion of the data, in which we test on all of the data (but notice we never train and test on the same data at the same time). The performance on all k held-out test sets can be averaged;

$$\text{global error estimate } = \frac{1}{N}\sum_{i=1}^{N} e_i$$

Note that this is used to evaluate an algorithm, not a particular model.

This method needs to be slightly modified when doing parameter tuning, in which we have the following options;

option 1: At each iteration, we use 1 fold for testing, 1 for validation, and the remaining $k-2$ folds for training. However, this will give us a different set of optimal parameters in each fold.

option 2: Another approach is to do cross-validation within cross-validation. As before, we still separate 1 fold for testing, however we run another internal cross-validation over the remaining $k-1$ folds to obtain the optimal hyperparameters. Once we obtain the best hyperparameters, we can then test it against the fold reserved for testing to obtain the final evaluation. This isn't always practical, as it requires a lot of computation for complex models.

When we go into production (not as common in academia), we may use all the remaining reserved test data for training as well (once we have the optimal parameters). However, this comes with the downside that we are no longer able to estimate the performance of the final trained model.

Performance Metrics

Once we have a model, we want to have a quantifiable way to judge the quality of a model against another. Consider the following results from the test dataset;

id	label	prediction
1	+	+
2	+	+
3	+	-
4	+	+
5	-	-
6	-	+
7	-	-
8	-	+

	class 1 (predicted)	class 2 (predicted)
class 1 (actual)	true positive (3)	false negative (1)
class 2 (actual)	false positive (2)	true negative (2)

This confusion matrix highlights the risk of each prediction - sometimes it can be more important to have fewer false negatives than fewer false positives (such as diagnosing a disease) It also allows for easy identification of confusion between classes (when one class is commonly mislabelled as another). Many other measures can be computed from the confusion matrix. In our example, we have two classes (positive and negative). The common measures are as follows;

- **accuracy** $\frac{TP+TN}{TP+TN+FP+FN}$

  This is simply the number of correctly classified examples divided by the total number of examples. The classification error is $1 -$ accuracy.

- **precision** $\frac{TP}{TP+FP}$

  This is the number of correctly classified positive examples divided by the total number of predicted positive examples. We can also think about it as;

  $$P(\texttt{positive} \mid \texttt{example classified as positive})$$

  A high precision implies that an example labelled as positive is actually positive (few false positives).

- **recall** $\frac{TP}{TP+FN}$

  This can be considered as the inverse of precision. It is the number of correctly classified positive examples divided by the number of actual positive examples. It can be thought of as

  $$P(\texttt{correctly classified as positive} \mid \texttt{actually positive})$$

  A high recall implies that the class is correctly recognised (therefore a small number of false negatives).

- **F-measure / F-score** $F_1 = \frac{2 \cdot \texttt{precision} \cdot \texttt{recall}}{\texttt{precision} + \texttt{recall}}$

  Sometimes it is useful to measure the performance of the classifier with a single number. More generally it can be written as (with more emphasis on precision for higher β);

  $$F_\beta = (1 + \beta^2) \cdot \frac{\texttt{precision} \cdot \texttt{recall}}{(\beta^2 \cdot \texttt{precision}) + \texttt{recall}}$$

For something to be high recall and low precision, most of the positive examples are correctly recognised, but with many false positives. On the other hand if something has low recall and high precision, we miss a lot of positive examples, but the ones that we predict as positive are more likely to be actually positive.

The macro-averaged recall is the mean of the recalls for all the classes. The same can be done for precision and F-measure. In the multi-class case, precision, recall, and F-measure are computed for each class separately (we define one class each time as being the positive class). Note that macro-averaging is done on the class level, and is the average of the metrics for each class. On the other hand, micro-averaging does it on the item level (adding the TP, FP, TN, FN values for each class before calculating the metrics). Note that micro-averaged P, R and F1 will be equal to accuracy.

Another measure is regression tasks, where a lower mean squared error (MSE) is better (where Y_i is a sample from the dataset and $\hat{Y}_i$ is the prediction from the model);

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

However, we don't only care about accuracy, our models should be;

- **accurate** — makes correct predictions
- **fast** — fast to train and query
- **scalable** — works with large datasets
- **simple** — understandable and robust
- **interpretable** — can explain predictions

Imbalanced Datasets

In a balanced dataset, we have an equal number of positive and negative data points. However, this will not always be the case, and we may have an unbalanced dataset where classes are not equally represented. The accuracy goes down, as it tends to follow the majority class. Additionally, the precision may also go down for the minority class. Consider the following case;

	class 1 (predicted)	class 2 (predicted)
class 1 (actual)	700	300
class 2 (actual)	100	0

From this, we obtain the following metrics where the accuracy may be high, but class 2 is completely misclassified;

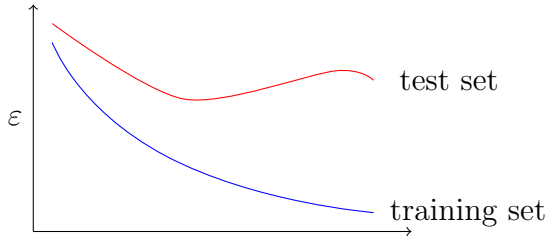
$$\begin{aligned} R(c1) &= 0.7 \\ P(c1) &= 0.875 \\ F_1(c1) &\approx 0.778 \\ R(c2) &= 0 \\ P(c2) &= 0 \\ F_1(c2) &= \texttt{N/A} \\ A &\approx 0.636 \end{aligned}$$

In conclusion, we need to look at different metrics, as well as the confusion matrix as a single metric may be misleading by itself.

We can normalise the confusion matrix by dividing each member of a row by the sum of that row (such that each row adds to one). We can also downsample the majority class, by picking less examples to get the two classes equal, or upsample the minority class by duplicating data. Neither will reflect how the model will generalise.

Overfitting

An overfitted model has good performance on training data, but poor generalisation to other data. On the other hand, underfitting has poor performance on the training data, as well as poor generalisation.



In the example above, it starts off with an underfitted model, and then ends up overfitted. The point where it's correct is just as the error of the test set begins to increase again.

Overfitting can occur under these scenarios (and how we could avoid it);

- model used is too complex (learns too many fine details)

  use the validation set to decide the complexity
- examples in the training set are not representative of all possible situations

  obtain more data
- learning is performed for too long (such as neural networks)

  stopping the training earlier (using the validation set to decide when)

Confidence Intervals

The amount of data used in our test set also affects our confidence of the performance evaluation. A 90% accuracy score on a test set with 10 samples is still less trustworthy than an 84% accuracy score on a test set with 10,000 samples.

We define the true error of the model h as the probability that it will misclassify a randomly drawn example x from distribution D;

$$error_D(h) \equiv P[f(x) \neq h(x)]$$

The **sample error** of the model h based on a data sample S is as follows;

$$\begin{aligned} n &= \text{number of samples} \\ \delta(f(x), h(x)) &= \begin{cases} 1 & f(x) \neq h(x) \\ 0 & f(x) = h(x) \end{cases} \\ error_S(h) &\equiv \frac{1}{n} \sum_{x \in S} \delta(f(x), h(x)) \end{aligned}$$

We can say an $N\%$ confidence interval for some parameter q is an interval with probability N to contain the true value of q. Given a sample S, with more than 30 examples;

$$error_S(h) \pm Z_N \underbrace{\sqrt{\frac{error_S(h) \cdot (1 - error_S(h))}{n}}}_{\text{est. standard deviation of sample error}}$$

Due to the n in the estimation of the standard deviation, if we have a very large n, we can obtain a very tight confidence interval. An example of this applied is as follows;

Emotion recognition results for 3 samples, using 156 training and 50 testing samples.

	attributes	number of classes	classifier	correctly classified
face	67 * 8	C4.5	78%	
body	140	6	BayesNet	90%

We want to classify the 95% confidence interval for this error ($Z_N = 1.96$)

$$\begin{aligned} error_S(h) &= 0.22 \\ n &= 50 \\ Z_N &= 1.96 \\ \text{interval} &= \left[0.22 - 1.96\sqrt{\frac{0.22 \cdot (1 - 0.22)}{50}}, 0.22 + 1.96\sqrt{\frac{0.22 \cdot (1 - 0.22)}{50}}\right] \\ &= [0.11, 0.33] \end{aligned}$$

Significance Testing

Statistical tests can tell us if the means of two sets are significantly different;

- **randomisation test**

  Randomly switch some predictions between two models and measure if the performance difference that we get is greater than or equal to the original difference.

- **two-sample T-test**

  This is used to estimate if two metrics from different populations are actually different. This has lower computational requirements and is easier to calculate.

- **paired T-test**

  Examining significance over multiple matched results, such as classification error over the same folds in cross-validation.

The **null hypothesis** (see **CO245**) is the hypothesis that the two algorithms / models perform the same and the differences are only due to sampling error. These tests return a **p-value**, which is the probability of obtaining the differences we see, assuming the null hypothesis is correct. A small p-value implies that we can be more confident that one system is actually different.

We consider a performance difference to be **statistically significant** of $p < 0.05$. However, $p > 0.05$ does not mean the algorithms are similar, just that we cannot observe a statistical difference.

There's a fairly long bit on **P-hacking**, but not sure why. A way to protect against P-hacking is to use an adaptive p-value;

1. rank p-values from M experiments;

$$p_1 \leq p_2 \leq p_3 \leq \cdots \leq p_M$$

2. calculate the **Benjamini-Hochberg** critical value for each experiment;

$$z_i = 0.05\frac{i}{M}$$

3. significant results are the ones where the p-value is smaller than the critical value

Week 5 (Artificial Neural Networks I)

Week 6 (Artificial Neural Networks II)

Week 7 (Unsupervised Learning)

Week 8 (Genetic Algorithms)