CS3010 Final Year Project (2020-2021)

Daniel Taylor

180125940

Mentor: Diego Faria

BSc Computer Science

Gesture Recognition via Machine Learning, Integrated with the Unity Environment

Introduction

Hand gesture HCI (Human Computer Interaction) is arguably the most natural way for humans to interact with computers, and is an area of research which is constantly being developed. Dynamic hand gestures are defined as a continuous movement of the hand, as opposed to a static hand gesture which remains constant. Therefore, multiple frames of video stream must be recorded in order for dynamic gestures to be recognised by computers, whilst static gestures can simply be recognised using just one frame. Through the use of extracting hand features from these frames, this allows the use of machine learning classifiers to make predictions of gestures using both skeletal data and depth images. Due to the progress of research and development within this area of HCI, interactions with computers are now more natural and no longer require humans to wear invasive equipment.

Classification is an exceptionally large area of machine learning which introduces many techniques ranging from imputing feature values within datasets, to implementing multimodality models. Due to machine learning problems containing different datasets, there is no specific model which is guaranteed to perform accurately on every classification problem. Therefore, it is vital to understand the data domain first, in order to efficiently implement optimal models.

This paper presents a comparative study of the various approaches used to develop an optimal machine learning model. An extensive dataset is collected before fitting classifiers including k-Nearest Neighbour, Random Forest, Naïve Bayes, Multi-layer perceptron, Support Vector Machine, Logistic Regression and a Decision Tree. Other ensembles of both a bagging and boosting nature were also fitted to the dataset, as well as a variety of pre-processing and tuning techniques. To ensure the robustness of the optimal model, an additional dataset is collected which will include unseen data, and therefore the prediction accuracy of the model on this dataset will increase the reliability of results.

The rest of the paper is as follows: The background research section explores related studies. The preparation section consists of relevant research topics, which are required to implement the machine learning models. The deliverable section conveys the implementation and development of machine learning models, and the evaluation and reflection section reflects on the overall study.

Contents

Background Research

Utilising both glove-based approaches parallel to vision-based approaches arguably offers the highest level of accurate, relevant data to be obtained. However, tracking devices attached to subjects can significantly reduce their comfort and lead to the collection of relatively poor data due to issues such as gyroscopic shifts. The technology is also extortionately priced and is therefore infeasible to incorporate within this study. However, vision-based approaches which require only the use of cameras (a minimum of two are required to allow the capture of depth), remain feasible and therefore are the most popular approach to collect hand gesture data.

A Microsoft Kinect sensor which allows subjects to interact with an Xbox 360 without the use of a game controller, consists of an array containing two 3D depth sensors, an RGB camera as well as four microphones. The depth sensors use an Infra-red camera to calculate the depth of objects in front of the sensor, whilst the four microphones have the ability to eliminate background noise. Windows narrator was used alongside the Kinect in [1] to output the corresponding word in a spoken format after a subject demonstrates a sign, although limitations of this study are that the sensor struggled to pick up objects outside of the range of 40cm-4M.

The leap motion controller (LMC) which was first released in 2013, uses stereoscopy and depth-sensing to detect both the joints and bones within a subject's human hand (See the anatomy of a human hand subsection) to a high level of accuracy [2]. The LMC utilises three Infrared Light emitters and two Infrared cameras to determine the exact location and positioning of the hand/s and constructs a model of the hands to replicate the posture/gesture being demonstrated by the subject. It is commonly used for interactive software applications [3] such as those including virtual or augmented reality, with an example of this being the Leap motion application, "Blocks". The optimal range of hands above the controller is approximately 80cm and has a visible area of 150° by 120° wide, however, occluded hands and accessories including watches and bracelets can cause the LMC to work incorrectly [3] [4]. The Asus Xtion Pro Live and Intel RealSense Depth camera are other popular depth and colour sensor alternatives used within studies [5] and [6] respectively.

In [7], surface electromyography (EMG) is used with machine learning to recognise gestures in real-time. A Myo armband is firstly used to acquire 200 observations a second of electrical waves produced by a subject's muscles. As noise is frequently collected within this data acquisition process, it is vital to pre-process the signals to eliminate the noise and therefore both filtering and rectification was included within this study to achieve this. Extracting feature vectors from the EMG signals after the pre-processing stage allowed various classifiers including naïve bayes, logistic regression, artificial neural networks (ANN), support vector machines (SVM), K-nearest neighbour (KNN) as well as decision trees to all be trained. 86% accuracy was obtained using a K-nearest neighbour classifier although as this study only contained five simplistic gestures, the accuracy of this model arguably underperformed. Limitations of this study however, are that old age leads to weaker EMG signals being produced and so do muscle disorders such as distal myopathy. Therefore, this study is not feasible for all users and a fusion of vision-based techniques along with EMG signals may lead to an improved accuracy of recognising gestures as well as improving the accessibility to a wider audience.

Colour Models are important within vision-based techniques as they help to obtain colour constancy which essentially offers a stabilised perception of object colours, allowing objects within images to be recognised with greater ease.

The RGB colour model is an additive model containing three channels (Red, Green and Blue). The combination of all channels with full intensities produces white, and the combination of all colours with no intensity produces black. However, this model generally performs poorly with real world images as the amount of light which reflects off objects in the digital image may vary but the chrominance and brightness are not independent channels and therefore the image cannot be stabilised. Furthermore, this leads to a varied level of RGB intensities, making it exceptionally difficult for computers to distinguish whether pixels contain skin.

The YCbCr model consists of a Y component which represents the level of luminance, and two channels which contain chrominance colour (Cb and Cr). This model is commonly used for image and video compression but has been successfully incorporated for skin detection in previous studies such as [8] [9].

The HSV (Hue, Saturation, Value) model is commonly used in computer vision studies as it is similar to the way humans recognise colours. The hue channel relates to the actual colours whilst the saturation value corresponds to the amount of colour utilised and the value channel relates to the brightness of the hue.

As images are obtained from an LMC and Kinect in an RGB format, these images should firstly be converted to either YCbCr or HSV before applying skin colour segmentation techniques. The conversion equations for these can be seen in Equation 1 and Equation 2.

Conversion of RGB to YCbCr

$$Y = 0.299R + 0.587G + 0.114B$$

$$Cb = -0.169R - 0.331G + 0.500B$$

$$Cr = 0.500R - 0.419G - 0.081B$$

Equation 1: The equations for each channel to convert RGB to YCbCr

Conversion of RGB to HSV

$$R = R/255.0$$

$$G = G/255.0$$

$$B = B/255.0$$

$$Cmax = \max(R, G, B)$$

$$Cmin = \min(R, G, B)$$

$$Range = Cmax - Cmin$$

$$H = \begin{cases} 60 * \left(\frac{G-B}{Range} \% 6\right) & Cmax = R \\ 60 * \left(\frac{B-R}{Range} + 2\right) & Cmax = G \\ 60 * \left(\frac{R-G}{Range} + 4\right) & Cmax = B \end{cases} \quad \text{H=0|Range=0}$$

$$S = \left(\frac{Range}{Cmax}\right) * 100 \qquad Cmax \neq 0 \quad \text{S=0|Cmax=0}$$

$$V = Cmax * 100$$

Equation 2: The equations for each channel to convert RGB to HSV

Results produced using the YCbCr and HSV colour models for skin detection still depend on the threshold value used to determine if skin is present. The YCbCr colour model is the preferred model out of the two in [10] due to a faster transformation from an RGB image whilst also performing well where uneven illumination is present in RGB images.

There are two known methods for detecting skin in images, a pixel-based approach, which is the preferred of the two, or a region-based approach. In the former approach, every individual pixel within an image is examined independently from its neighbours to determine whether the pixel can be classified as skin or not skin. However, a region-based approach which can be seen in [11] uses 'superpixels' which are groups of individual pixels that are of a similar colour. These superpixels can then be classified as either skin or not skin depending on a threshold. This region-based approach produced an accuracy of 91% for detecting skin pixels which was an improvement of Jones and Rehg's pixel-based approach that achieved 90% accuracy [12].

A common application of gesture recognition is to offer hearing impaired subjects the opportunity to interact with computers or other subjects through the use of sign language. [13] carried out a study which allowed hearing-impaired subjects to communicate with other subjects who have no knowledge of the signs being demonstrated through the use of both text as well as speech, allowing the two subjects to communicate with no language barrier and affordable technology.

Other applications include gestures being used to interact with telerobots in life threatening environments. This is similar to virtual reality although instead of them being incorporated within a virtual world, the real world is utilised along with sensors to feedback the surrounding environment.

[14] achieved 95% accuracy when controlling a robot in real-time using ten various gestures. With the use of global thresholding, the video stream frames were converted into binary images, allowing for an XOR function to determine how similar images are to one another. If and only if there was less than 1% variability between an image and its previous image could the current frame be used for classifying a gesture. Extraction of the hand from the frame containing the gesture was then required to ensure the subject's wrist or other features which may influence results is eliminated. Principal Component Analysis (PCA), was the preferred method to match patterns of gestures compared to an ANN which was time consuming and therefore not appropriate for real-time processing.

Variations of Hidden Markov models (HMM) were previously a popular method utilised to recognise gestures where class sequence predictions are made when provided with a sequence of observations. However, limitations of this are that the Viterbi algorithm, which is commonly incorporated to evaluate gestures as it is the most efficient method, is still computationally expensive. Yamato, Ohya and Ishii were the first to publish an academic paper which incorporated HMMs into recognising motions and were successful in doing so with results exceeding 90% accuracy. The study consisted of six different tennis strokes which were demonstrated ten times by three different subjects. Frames within videos of television-broadcasted tennis matches were segmented so that the subject motions could be tracked. However, the results obtained within this study could be perceived as ambiguous and bias in that only a small dataset was collected with very few gestures and subjects involved. The 90% accuracy of classifying gestures related to the evaluation of only one subject, leading to poor results when applied to other subjects carrying out the same motions. Although, when a mixture of multiple subjects was included in the training and testing data, this reduced the bias and produced an accuracy of 70.8% [15]. Following on from this, Petkovic, Jonker and Zivkovic carried out a study which was more robust as it was not limited to specific court types and colours, and that it also considered skeleton features of the final player segmentation to classify gestures [16].

[17] is another study which utilises HMMs, although Alizadeh used a Microsoft Kinect sensor alongside Delicode NI mate [18], to track skeletons of a subject's hands rather than an entire human body. Ten gestures were demonstrated by four subjects of ages ranging between three and eight. Motion direction was evaluated as a more accurate technique (72% accuracy) to recognise gestures than point localisation (68% accuracy) which is logical as the coordinates of hands within a 3D space will not necessarily be the same for all subjects, although the motions which are carried out when performing gestures will be almost identical across participants. Limitations however, include gestures being unintentionally recognised when subjects change between gestures, and that many gestures are not recognised at all due to previous filtering of important data.

Machine learning algorithms which have been frequently incorporated into gesture recognition studies include SVM, KNN, Naïve Bayes and ANN such as in [19]. This study was reliant on the Rapid Miner tool [20] however, and therefore only provides a vague insight into which machine learning algorithms perform better than others for gesture recognition. KNN and naïve bayes are exceptionally fast when applied to the dataset with only five features and produced respectable accuracies too, however, the best set of accuracies was present within the dataset which contained more features. The ANN produced the best accuracy of 97% although it did require 46.5 minutes to train the model which is almost six times as long as the KNN classifier which produced the second-best accuracy of 95%.

[21] incorporates skin colour segmentation with the YCbCr colour space to extract the hand from the background before using Otsu thresholding to binarize the image. PCA is then used as a method to reduce the dimensions within the images while retaining a high percentage of useful information. Template matching was then used to recognise the gestures. Results of 91.25% showed potential with this method, although the dataset was very small as it only contained eighty images (twenty images of four static hand poses), with only four subjects involved in the study. [22] is similar to this previous study in that they both use PCA and template matching, although the former study recognises static hand poses whilst the latter classifies dynamic hand gestures. The results obtained in the latter study illustrate that it has 100% accuracy at recognising gestures although as the test dataset and the training dataset are almost similar, these results are unreliable and therefore cannot be assumed that PCA was necessarily the reasoning behind these impressive results.

In study [23], canny edge detection was used, in order to segment hands from their backgrounds before utilising Oriented Fast and Rotated Brief (ORB) to detect features within the segmented hand images. With the use of K-Means clustering, a bag of words can be generated before training classical machine learning classifiers. The classifiers which were trialled included Multilayer perceptron, Naïve Bayes, SVM, KNN, Random Forest as well as Logistic Regression. The Multi-layer perceptron, random forest and KNN classifiers all performed exceptionally well with both PCA as well as ORB however the use of ORB did lead to a significant improvement of the naïve bayes classifier which performed poorly with PCA.

Deep learning can be seen in [24] where the results obtained using convolutional neural networks (CNN) and stacked denoising autoencoders are extremely promising, and produce less predictive errors which classical machine learning techniques are known to produce. The best models for each of these methods were 91.33% and 92.83% on testing data respectively, and considering there are 600 images within the testing dataset that contains 24 different American Sign Language (ASL) gestures, this proves the validity of the study's results.

In studies [25] [26], both depth and skeleton data was used in order to recognise dynamic gestures. In the former study, a CNN is used to firstly extract features from a depth image where the subject's hand is present, before going on to using a Recurrent Neural Network (RNN) to extract patterns within the sequence of hand skeleton information. The two networks are then fused together with the intent to improve the model's performance. The use of a CNN alone resulted in 76% accuracy although 82% accuracy was obtained using the skeleton data. However, the proposed method of fusing the two networks resulted in 85% accuracy. This accuracy did decrease to 74% when applied to a dataset containing a larger quantity of gestures (28 gestures), although this is expected as the weights within the network were not optimised for the more complexed dataset, but did however still produce a respectable accuracy. In the latter study, Bird uses decision-level fusion of both depth and skeleton data to produce more accurate results for classifying British Sign Language (BSL) signs than what either of the individual components are capable of achieving. A VGG16 model was used to extract features from the depth images whilst an ANN was used to classify gestures using skeletal information. The use of running an evolutionary search three times on the ANN was extremely beneficial as it prevented the likelihood of a local maxima being transferred through the later stages of the study whilst also providing certainty that the neural network model is performing optimally. An accuracy of 77% was obtained by the multimodality approach when applied to unseen data and as the dataset collected within this study contained six

more gestures than what was incorporated in the main study of [25] that produced the accuracy of 85%, the results obtained by Bird are highly respectable.

Reinforcement learning (RL) is commonly incorporated into game applications where agents receive rewards depending on their actions within an environment (the environment must be a Markov decision process where the next state is influenced by the current state). RL can be applied to a variety of aspects within games such as within Non-Playable Characters (NPCs), however, this may be considered inappropriate in some applications such as fighting games where NPCs could potentially be too difficult for users to beat. Another example of RL in game applications includes altering the difficulty of levels although this has the same disadvantage to NPCs such as a level potentially becoming too difficult to complete. The ML-Agents toolkit is an open-source project which allows the incorporation of RL into the Unity game engine and allows agents to be trained using RL or imitation learning [27].

Genetic Algorithms are used within [28] to train the Deep Reinforcement Learning model parameters, and by doing so this removes the need to implement the back propagation algorithm. This proved successful with the Open AI Gym environments, although one of the models still required improvement to prevent the possibility of an agent being trapped in a local optima state.

RL can be split into both model-based RL and model-free RL. Model-based RL offers the ability to plan in advance, helping an agent to determine their policy (the algorithm utilised to determine the action to be taken in a specified state), which also improves the level of sample efficiency. On the other hand, model-free RL is explicitly a trial-and-error approach as the agent does not use a model, and therefore makes decisions depending on the current state it is in when it reaches that state, rather than planning in advance. Agents having access to models tends to not be viable (this restricts it being placed in unknown environments) and as a model-based RL approach is also more complexed to implement, model-free RL is usually the preferred variation of RL [29] [30]. Richard Sutton, however, states in [31] that a more feasible approach is to combine both model-based and model-free RL together. Reinforcement Learning is still however limited to virtual environments rather than real-life situations [32].

Imitation learning is sometimes preferred over reinforcement learning in scenarios where rewards are not frequently awarded. A supervisor carries out demonstrations which eventually allows an agent to learn and imitate the supervisor's actions. Imitation can be seen in studies involving infants and animals and is a considerably faster approach to learn as opposed to methods such as reinforcement learning. Priming and response facilitation can influence imitation such as in [33] where a collection of chimpanzees, bonobos, gorillas and orangutans imitated a supervisors actions including yawning and scratching, although these types of actions are not classified as "true imitation". "True imitation" instead, relates to successful imitation of an unknown task after viewing a demonstration and can be applied to gestures such as in [34]. Once again, a Kinect is utilised to detect subject's motions before transferring the cartesian skeleton data via a USB to a computer. The computer then converts this data so that each of the joints within the NAO humanoid robot correspond to the human subject's joints. Following on from this the relevant commands are then sent to the robot over Wi-Fi, allowing it to mimic the gesture being demonstrated in real-time. In the same paper, an Extreme Learning Machine (ELM) approach is proposed to recognise gestures in real-time before comparing the results to both a Feed forward neural network (FNN) and a KNN classifier. This produced a higher accuracy than the other algorithms (96% on testing data), and was also significantly faster than the other approaches at recognising gestures in real-time.

Anatomy of a Human Hand

The human hand consists of 17 joints which offers 23 degrees of freedom. Each of the fingers are composed of a distal interphalangeal (DIP), proximal interphalangeal (PIP), and a metacarpophalangeal (MCP) joint from the fingertip to the base of the finger respectively. The thumb also consists of three joints which include a thumb interphalangeal (Thumb IP), a thumb metacarpophalangeal (Thumb MP), and a trapeziomatacarpal from the tip of the thumb to the base of the thumb respectively [35]**.**

The phalanges (phalanges is the plural of phalanx) are the bones within the fingers that can be found in between the tip of the fingers and the base of the fingers, which can be split down into the distal, middle and proximal phalanges. The thumb however, only consists of a distal and proximal phalanx. Carpal bones are found within what is known as the wrist and the carpometacarpal (CMC) joints connect the hand to the wrist. A diagram of the bones can be seen below on the left, and the diagram of the hand joints can be viewed on the right:

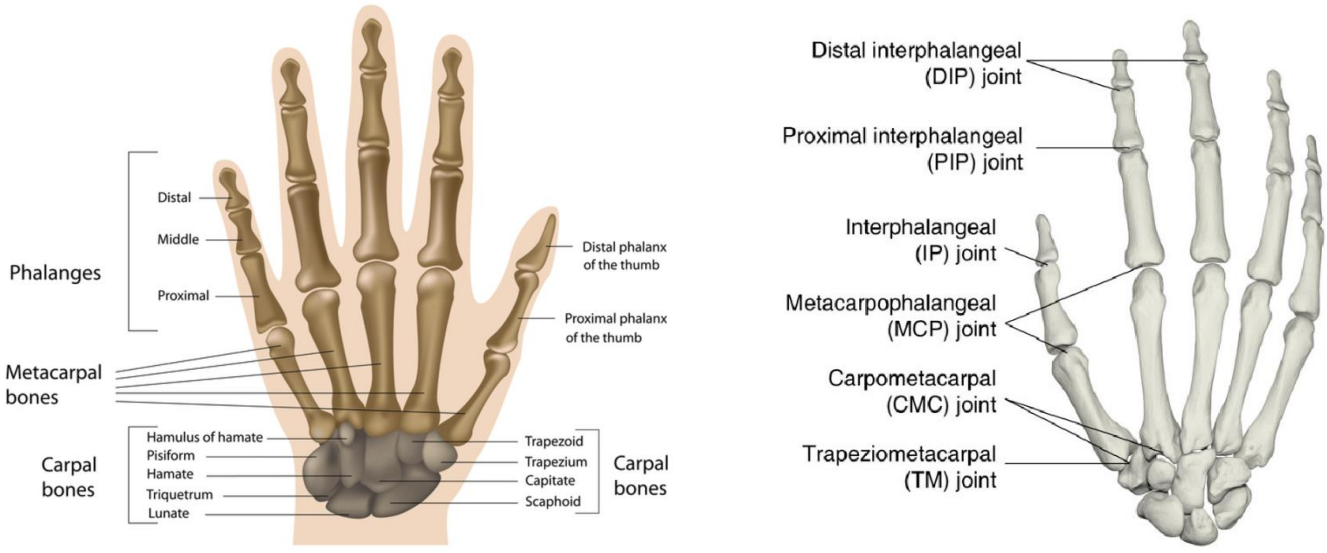


Fig.1: The Bones within a Human hand and wrist (Image on the Left). Image taken from https://www.researchgate.net/figure/Bones-of-a-human-hand-and-wrist-adapted-from-the-Human-Anatomy-library-8_fig1_321823302. The Joints found within a Human Hand (Image on the Right). Image taken from: https://www.researchgate.net/figure/Joints-of-the-right-hand-dorsal-view-Note-that-the-terms-trapeziometacarpal-TM-joint_fig1_257045252.

Preparation

Python

Python is the most popular programming language for Artificial Intelligence and machine learning as it is both simple to learn due to its close relation with the English language and is also extremely effective at processing data with the assistance of its library ecosystem.

Libraries

Numpy

Numpy stands for numerical python and is considered in scientific applications where numerical data is involved. One of its benefits over the built-in python data structures is that lower-level languages such as C can work directly with data stored in NumPy structures.

Matplotlib

This library allows data to be presented using visualisations such as graphs and plots and although there are multiple other libraries which offer visualisations of data such as seaborn or Plotly, this library is considered the most popular library as it is fairly basic and integrates nicely with other libraries within the ecosystem.

Scikit-Learn

This general-purpose machine learning library offers multiple algorithms of both a supervised and unsupervised nature as well as other essential features required for machine learning, including dimensionality reduction and pre-processing.

Pandas

The pandas library allows data to be structured in a high-level data structure known as a ‘dataframe’ which consists of both rows and columns along with labels. Another data structure which is commonly used within this library is a ‘series’ which is a labelled one-dimensional array.

Imbalanced-Learn

This library offers several re-sampling methods which are a necessity for classification problems where an extreme imbalance of classes is present. It is also compatible with the scikit-learn library.

Anaconda

Anaconda is the most popular distribution of machine learning and data science resources for both python and R programming languages, offering simple package management along with deployment. Anaconda offers a graphical user interface (GUI), known as Anaconda Navigator, which allows a simplified approach to “manage conda packages, environments and channels without the need to use command line commands” [36].

Jupyter Notebook

Jupyter notebook is a web-based IDE (integrated development environment) commonly used for machine learning and data science projects (It can be found in the anaconda navigator application). It offers programmers the ability to not only write executable software, but also include text elements such as images, links, and paragraphs. This is useful for projects where explanations can help analyse the code beyond programming comments. Another feature which is advantageous over other IDEs is that the code can be split up into multiple cells with no limitation on the number of cells. This therefore offers efficient debugging of software and does not require the whole program to be rerun but rather just the erroneous cells.

Weka Explorer

Weka explorer is a useful platform to apply machine learning algorithms to datasets using a graphical approach with no need for any programming commands. There is a wide variety of algorithms available to use within the platform, although some of these do have slightly obscure names and it can be considered daunting seeing many of the hyperparameters being utilised within the classifiers and trying to get to grips with the interface. However, once the various aspects are familiar, weka explorer offers both supervised and unsupervised learning techniques along with pre-processing and visualisations of results.

Unsupervised and supervised learning

Supervised learning relates to machine learning problems where both input (X) and output variables (y) are present within a dataset. The process is supervised as the output variables are known and therefore the algorithms can be adapted accordingly to improve their predicting accuracy. An algorithm learns a mapping of the input variables to the output variables so that it can accurately predict the output when provided with a new set of unseen input data. Supervised learning can be split into both classification and regression problems. Classification problems relate to problems which have a categorical output variable such as '*Gesture1*' or '*Labrador*. Regression problems on the other hand, are problems which consist of real values such as '1.38' for an output variable relating to pounds sterling (£), or '11.4' for an output variable relating to weight (st).

Unsupervised learning alternatively relates to machine learning problems where input variables are present but output variables are not. It is considered unsupervised as there are no correct outputs and do not require human intervention to analyse data. The algorithms are simply left to find structure and patterns within the data by their own accord. Clustering is an example of this where groupings are made within the data such as clusters of consumers purchasing habits or perhaps using it for anomaly detection situations.

Types of Data

Training Data

The training dataset is what the models are fitted to in order to make a generalised mapping of inputs to classes. It is important for this dataset to include examples of each of the expected categorical labels as this reduces errors arising when the trained model is applied to unseen data.

This dataset is split into both a training subset and a validation subset using cross-validation. The validation subset is used to evaluate the models' performance on the training subset.

Testing Data

The testing dataset consists of data that is not present in the training dataset to prevent a bias and this ensures that the models' performance on the unseen data is accurate. It is preferred that each of the classes in the training dataset are also included in the testing dataset to ensure that the model is robust to multiple predictions of various classes.

Classification Types

Binary Classification

Binary Classification is where there are only two class labels available to be classified by models (this is usually split into a normal state and an abnormal state). Models which predict a bernoulli distribution are used for binary classification tasks with examples of these including support vector machine and logistic regression models.

Multi-Class Classification

Multi-class classification relates to tasks where there are more than two class labels to be classified by models. Models that predict a multinoulli distribution such as random forest, are commonly used for multi-class classification tasks as they must be capable of predicting the probability of a class from a range of two or more other classes. Examples of multi-class problems include classifying dog breeds or classifying car parts. Most models used for binary classification can also be used for multi-class classification. However, in order for them to support the latter, the binary classification algorithms may require meta-strategies.

Multi-Label Classification

Multi-label classification tasks are those which can have more than one prediction made for each observation. An example of this could be an image which contains multiple objects. This is a multi-label classification problem as a house, car and grass may all be visible in the image and therefore all 3 must be predicted. Models that are applied to multi-label problems, make binary classification predictions for each of the potential classes. Using the previous example, this would ensure that an image will either contain the target being classified or not contain the target, and allow each of the available classes to be checked within the image. Basic classification models must be adapted to support multi-label problems with an example of this being a multi-label random forest or a multi-label support vector machine.

Imbalanced Classification

When the number of instances in each class are not equally distributed within the training dataset, it is said to be an imbalanced classification problem and is biased. An example of this may be to collect numerical attributes of trees, and have 70 observations of one tree species, but only 30 observations of the second tree species. There are two main categories that can cause an imbalanced

classification task, which are data sampling and the domain properties. The data sampling may be biased in some cases as perhaps class labels may not have been changed when collecting data for a new class label, or perhaps the equipment being used to collect the data may have been damaged after already taking previous samples. Domain properties may cause an imbalance as resources, cost or quantity of time required to obtain more samples of a certain class would be infeasible. Common examples of imbalanced classification tasks include fraud detection and outlier detection. A slight Imbalance is where classes are distributed unevenly in the training dataset, but only by a small amount. This is rarely a problem for classification tasks, as opposed to those with severe imbalances which have a large range between class distribution observations in the training dataset and may require special techniques to minimize the impact of training models. The minority class of an imbalanced classification problems is the class which has the smallest quantity of examples, whilst the majority class is the class with the largest quantity of examples. The minority class in a severe imbalanced classification problem takes precedence. Therefore, the ability of a model to correctly predict the minority class is of a higher importance than for the model to predict other classes. The level of difficulty to predict a class label using a minority class is increased however, as there are fewer examples to train models with, making it difficult for models to differentiate a minority class from another class or classes.

Oversampling and Undersampling

When the classes within the training data are severely imbalanced, random oversampling and random undersampling should be considered to resample the training dataset, as without this control, it is a possibility that classifiers could completely ignore a minority class. Oversampling consists of duplicating instances of the minority class to increase the number of observations for the class, whilst undersampling relates to cutting out instances within the majority class to reduce the number of observations for this class. Naïve resampling is preferred for large datasets where the implementation and processing are fast due to no assumptions being made of the data when the resampling takes place. It should be noted that the resampling only takes place on the training dataset though, as once the model has been fitted to a more balanced dataset (after resampling), the performance of the model should then be robust to unseen data. The imbalanced-learn python library offers implementations including '*RandomOverSampler*' *and* '*RandomUnderSampler*' to help resample training datasets.

Unity

Unity offers developers the ability to develop 2D, 3D and Virtual reality applications as well as games. Ranging from real world applications to animations, the use of Unity is varied although it is a very popular platform for projects such as those which incorporate unstable robotics as some situations must be analysed virtually before deploying them in the real world such as autonomous cars or telerobots. With the use of an asset store, this helps developers to implement projects with efficiency too. The primary script API is written using C# although with the use of the Python for Unity package, this allows for python scripts to be integrated within the game engine meaning the final machine learning model does not need to be converted into C#.

Leap Motion Controller

The leap motion controller (LMC) consists of two infrared cameras and three infrared light-emitting diodes (LEDs) which can be seen in Fig.2 along with the visual scope of an LMC in Fig.3.

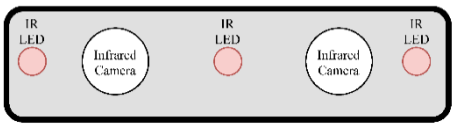


Fig.2: A labelled diagram illustrating the layout of infrared cameras and infrared LEDs in the Leap motion controller. Image taken from https://www.mdpi.com/1424-8220/20/18/5151/htm.

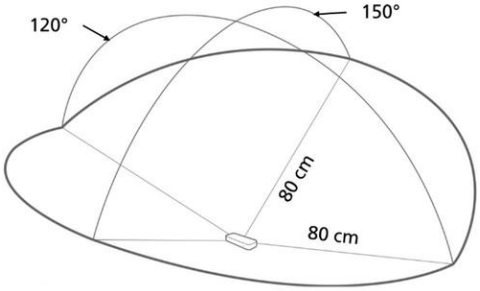


Fig.3: A diagram showing the hand tracking zone of a leap motion controller positioned on a flat surface. Image taken from https://www.researchgate.net/figure/Hardware-left-and-the-field-of-view-right-of-the-Leap-Motion-Sensor-pictures_fig3_339891094.

Following on from the background research section which lightly touched upon the hardware aspects of the leap motion controller, the software is now to be discussed. Frames of the video data are firstly preprocessed to eliminate background objects such as bright lights or human heads using the leap motion service. A 3D visualisation of the subject's hands can then be generated. The hand tacking software, Gemini V5, is used to extract hand features from frames so that they can be tracked and offers a better performance of tracking two hands within the same frame compared to other algorithms such as Orion [37].

Automated Machine Learning (AutoML)

AutoML constructs machine learning models using an iterative approach which increases the efficiency and productivity of data scientists as the time-consuming steps within model implementation are all automated within the iterations.

Microsoft Azure

Microsoft Azure offers machine learning as a service to all users of any experience. Beginners have the ability to create machine learning models using the automated machine learning feature whilst intermediate developers may opt to use the drag-and-drop designer feature. Experienced users however, have the ability to implement models using a wide variety of languages and popular libraries already at hand, without the need of additional instalments. Due to the computational cost of processing certain models on low specification hardware, Azure is considered a cost-effective solution as it does not require developers to own top of the range equipment to process these

models efficiently. The security of data along with the simple deployment of models provides additional beneficial features Azure has to offer.

However, self-management of azure projects, as well as the potential costs of maintaining deployments do provide some disadvantages of using Microsoft azure. Along with this, it can also be viewed as a complicated interface to interact with, and therefore this may lead to refraining away from considering the cloud computing service.

TPOT

Tree-based pipeline optimisation tool (TPOT), is an open-source automated machine learning library in python which uses scikit-learn to manipulate both the data itself and the models which are fitted to the data. Through the use of a genetic algorithm, stochastic optimisation takes place in order to find an optimal solution where randomness is present. The best pipeline which is returned to the developer can then be exported to a python file so it can be loaded later without the need for having to run the whole AutoML process again.

Overfitting

Overfitting takes place when a model is too complexed after the training process, and therefore appears to perform well on the training data but poorly on unseen data (it is clear when a model is overfitting the training data as there is a large variance between the two accuracies). There are many approaches which can prevent overfitting such as using more data for training, by simplifying the model or making use of feature extraction or feature selection.

Underfitting

Models which have both a low training accuracy and low accuracy on unseen data are said to be underfitting models. Approaches to reduce this include increasing the training duration, collecting additional features (although this is usually expensive and not feasible), or by using feature engineering to construct new features using the current features within the training dataset.

Learning Curves

Learning curves are useful to use to determine how many instances of training data are required to produce high accuracy results for both training and validation datasets. This therefore also helps to prevent both overfitting and underfitting.

Using the learning curve in Fig.4, it is clear that the model significantly overfits the data until approximately 30 examples of training data have been seen, and performs particularly well after seeing roughly 55 examples where the standard deviation between the mean training accuracy and mean validation accuracy is minimal. However, it would be preferred if the standard deviation of the validation accuracy was considerably smaller than what is present.

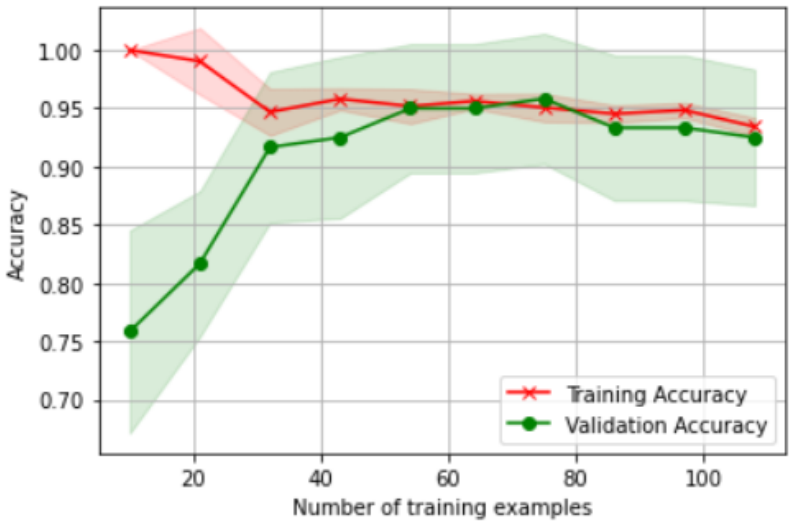


Fig.4: Learning Curve for the Naive Bayes classifier being applied to the load_iris dataset

The learning curve can have an increased number of plotting points by changing the '*lin_space*' variable. However, as the learning curve approach uses stratified k-fold cross validation as the default method to calculate the classifiers accuracy, the number of folds is commonly set to 10 and therefore this is an appropriate number of plotting points to include.

Splitting

Splitting is used for tree induction and aims to split nodes down into sub-nodes in a way that a homogenous class distribution is present where one class dominates all other classes in each of the individual sub-nodes. This approach for splitting is known as the greedy approach.

Measuring Impurity

Both entropy and gini impurity are commonly used to calculate the level of impurity which is used to split features within a decision tree.

Entropy

Entropy is used to calculate the purity of splitting and essentially measures the randomness of the information, where a high value of entropy leads to an increased level of difficulty to make decisions using the information. The equation for entropy can be seen in Equation 3.

$$Entropy(t) = -\sum_{i=1}^{c} p(i|t)\log_2\big(p(i|t)\big)$$

Equation 3: Entropy Equation

The minimum level of entropy is always 0 no matter how many classes are present within the data as it is possible for all observations to belong to one class, although the maximum level of entropy can simply be calculated using Equation 4 which was taken from [38]:

$$Entropy_{\max}(t) = \log_2(classes)$$

Equation 4: Maximum Entropy Equation

Therefore if 8 classes were present in the dataset, the minimum entropy would be 0 whilst the maximum entropy would be 3. An example using a binary classification and a multi-class classification problem can be seen in Fig.5 and Fig.6 respectively. Fig.6 also illustrates how a perfect balance with the observations of classes produces maximum entropy.

Dog	6
Cat	2

P(Dog) = 6/8

P(Cat) = 2/8

$$Entropy = -\left(\frac{6}{8}\right)\log_2\left(\frac{6}{8}\right) - \left(\frac{2}{8}\right)\log_2\left(\frac{2}{8}\right) = 0.81$$

Fig.5: Entropy calculation for the Binary Classification Problem

Dog	3
Cat	3
Sheep	3
Pig	3

P(Dog) = 3/12

P(Cat) = 3/12

P(Sheep) = 3/12

P(Pig) = 3/12

$$Entropy = -\left(\frac{3}{12}\right)\log_2\left(\frac{3}{12}\right) - \left(\frac{3}{12}\right)\log_2\left(\frac{3}{12}\right) - \left(\frac{3}{12}\right)\log_2\left(\frac{3}{12}\right) - \left(\frac{3}{12}\right)\log_2\left(\frac{3}{12}\right) = 2.00$$

Fig.6: An example of maximum entropy being calculated with a multi-class classification problem

Once calculated, entropy can then be used to calculate the information gain (IG) which measures how well a split at a specified node, t, actually is. The IG equation can be seen in Equation 5 which was taken from [38]:

$$IG(t) = Entropy(t) - \sum_{j=1}^{k} \frac{n_j}{n_t} Entropy(j)$$

Equation 5: Information Gain Equation

Where n_j relates to the number of observations at the child node 'j' and n_t relates to the number of observations at the parent node 't'. The value of k corresponds to the number of child nodes coming from the parent node. An example of calculating information gain using the former binary classification problem can be seen in Fig.7, Fig.8 and Fig.9.

Dog	6
Cat	2

$n_t = 8$

Fig.7: Parent Node

	n_1	n_2
Dog	4	2
Cat	1	1

Fig.8: Child Nodes

$$Entropy(Parent\ Node) = -\left(\frac{6}{8}\right)\log_2\left(\frac{6}{8}\right) - \left(\frac{2}{8}\right)\log_2\left(\frac{2}{8}\right) = 0.81$$

$$Entropy(n_1) = -\left(\frac{4}{5}\right)\log_2\left(\frac{4}{5}\right) - \left(\frac{1}{5}\right)\log_2\left(\frac{1}{5}\right) = 0.72$$

$$Entropy(n_2) = -\left(\frac{2}{3}\right)\log_2\left(\frac{2}{3}\right) - \left(\frac{1}{3}\right)\log_2\left(\frac{1}{3}\right) = 0.92$$

$$IG(Parent\ Node) = Entropy(Parent\ Node) - \left[\frac{5}{8}Entropy(n_1) + \frac{3}{8}Entropy(n_2)\right] = 0.02$$

Fig.9: Calculations of the Information gain Example

As the information gain value is significantly low, it would be more appropriate for the splitting position to occur where a higher information gain value is present if possible.

Gini Impurity

When the target class is of a categorical form, gini impurity is the preferred approach to use to split a decision tree. An example may help to consolidate the understanding of this approach and can be seen in Fig.10 and Fig.11.

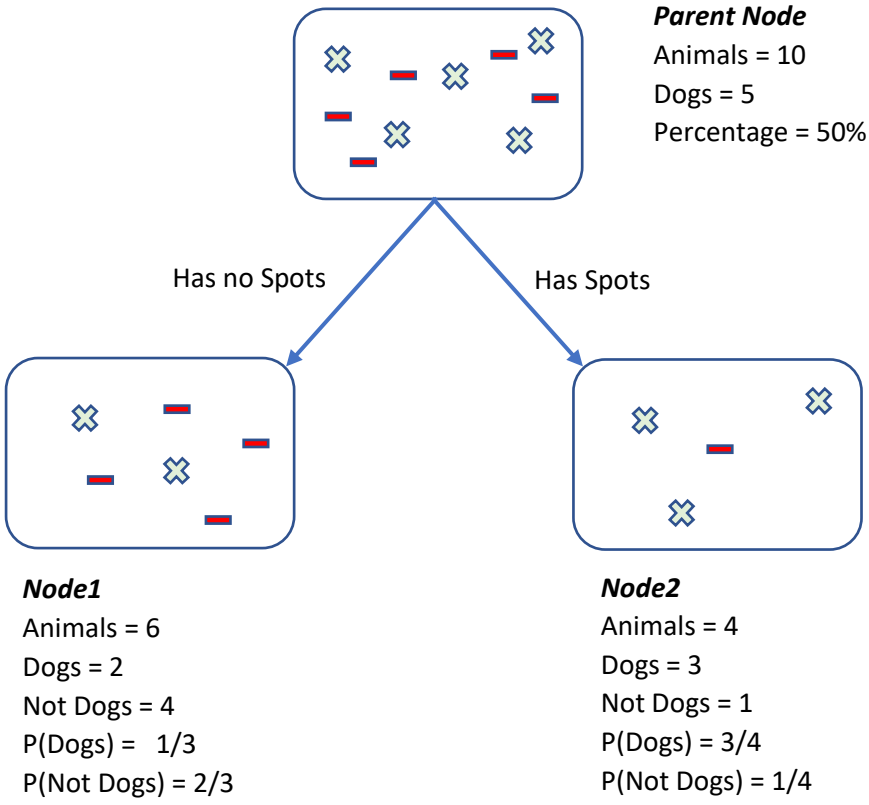


Fig.10: Example of a Decision Tree

$$\text{Gini impurity of Has no Spots} = 1 - \left[\left(\frac{1}{3}\right)^2 + \left(\frac{2}{3}\right)^2\right] = 0.44$$

$$\text{Gini impurity of Has Spots} = 1 - \left[\left(\frac{3}{4}\right)^2 + \left(\frac{1}{4}\right)^2\right] = 0.38$$

$$\text{Weighted Gini impurity} = \left(\frac{6}{10}\right) * 0.44 + \left(\frac{4}{10}\right) * 0.38 = 0.42$$

Fig.11: Calculating the Gini Impurity of the Decision Tree in Fig.9

Once this has been calculated for each of the features, the feature which produces the lowest gini impurity should be used to split the parent node, as the lower the gini impurity, the purer the nodes will be when split and therefore a high homogeneity.

Gain Ratio

A limitation of splitting using entropy is that larger splits take precedence, and therefore this leads to a large quantity of small leaf nodes due to the equal weighting within IG. Furthermore, this makes the process inefficient and potentially ungeneralisable. To eliminate this bias, gain ratio is used. IG is divided by splitting information which essentially accounts for the number of splits and their relative size. The equation which corresponds for this can be seen in Equation 6 and has been taken from [38].

$$Gain\ ratio(t) = \frac{IG(t)}{-\sum_{i=1}^{c}\left[\frac{n_i}{n_t}\log_2\left(\frac{n_i}{n_t}\right)\right]}$$

Equation 6: Gain Ratio Equation

Classification models

Logistic Regression

Logistic Regression is commonly used for both linear and binary classification problems. The name is slightly misleading as it contains the word 'regression' although this classifier is used for classification problems and not regression problems.

The logistic function (sigmoid function) is used to make predictions and does this by taking any real value as an input and mapping it to a value within the range of 0 and 1 with these exact boundaries not being included within the range due to them being asymptotes. The equation which this function incorporates can be seen in Equation 7.

$$y = \frac{1}{1 + e^{-x}}$$

Equation 7: The Sigmoid Function

Where the value of 'x' relates to the input value being mapped, the 'e' symbol relates to the base of natural logarithms, and the value of y is equal to the new input value after the mapping [39].

However, as this study relates to multi-class classification problems, the 'multi_class' hyperparameter within the classifier should be set to the value, 'multinomial'. This essentially transforms the estimator into a multinomial logistic regression classifier which is also referred to as a softmax regression classifier. However, instead of using the sigmoid function, the softmax function is used in replacement with this classifier which can be seen in Equation 8.

$$y = \frac{e^x}{\sum_{i=1}^{n} e^x}$$

Equation 8: The Softmax Function

Where the value of 'x' relates to the input value being mapped and the 'e' symbol relates to the base of natural logarithms. The denominator sums all of the newly mapped inputs in the numerator. 'y' symbolises the values after the softmax function has been applied to the inputted values. The maximum likelihood is used to fit the softmax or sigmoid curve onto the distribution of classes.

Naïve Bayes

Naïve bayes works by using bayes theorem to determine the probability of each observation being classified as each of the classes, with the highest probability being the final predicted class. All of the features are assumed to be independent of one another, meaning they are all equally important and correlations between features does not impact predictions made by the classifier. This classifier is extremely fast although commonly produces poor results when applied to large datasets as it tends to overgeneralise the data. It also performs better when features are of a categorical nature rather than numerical. Equation 9 illustrates the Bayes Theorem Equation.

$$P(y \mid X) = \frac{P(X \mid y) * P(y) * P(X)}{P(X)}$$

Equation 9: Bayes Theorem Equation

Where the 'y' variable relates to the class label output (the 'car or van' variable in Table 1), and the X variable relates to the dataset's input features. An example of the naïve bayes classifier being applied to a simple dataset can be seen in Tables 1-5.

Index	Size	Colour	Car or Van
1	Big	Black	Car
2	Small	Black	Van
3	Small	Red	Van
4	Big	Black	Car
5	Small	Red	Car
6	Big	Red	Car
7	Big	Red	Van

Table 1: A randomly created Dataset for a Binary Classification Problem

Size	Car	Van
Big	3	1
Small	1	2

Table 2: Frequency table of 'Size'

Size	Car	Van
Big	3/4	1/3
Small	1/4	2/3

Table 3: Probability of 'Size''

Colour	Car	Van
Black	2	1
Red	2	2

Table 4: Frequency table of 'Colour'

Colour	Car	Van
Black	1/2	1/3
Red	1/2	2/3

Table 5: Probability of Colour

$$\mathrm{P}(\mathrm{Car} \mid \mathrm{X}) = \mathrm{P}(\mathrm{Red} \mid \mathrm{Car}) * \mathrm{P}(\mathrm{Big} \mid \mathrm{Car}) * \mathrm{P}(\mathrm{Car})$$

$$\mathrm{P}(\mathrm{Car} \mid \mathrm{X}) = (1/2) * (3/4) * 1 = 0.375$$

$$\mathrm{P}(\mathrm{Van} \mid \mathrm{X}) = \mathrm{P}(\mathrm{Red} \mid \mathrm{Van}) * \mathrm{P}(\mathrm{Big} \mid \mathrm{Van}) * \mathrm{P}(\mathrm{Van})$$

$$\mathrm{P}(\mathrm{Van} \mid \mathrm{X}) = (2/3) * (1/3) * 1 = 0.222$$

Therefore, as the higher probability of the two is 0.375, this concludes that the predicted class when inputs 'Big' and 'Red' are passed, is 'Car'.

Support Vector Machines

Support vector machines can be utilised in both linear and non-linear separable situations. In the situation of linear separatable classes, the optimal hyperplane which separates the classes is located where the perpendicular distance between this line and the classes is maximised (this is known as the maximum margin). An analysis of the SVM being applied to data can be seen using Fig.12.

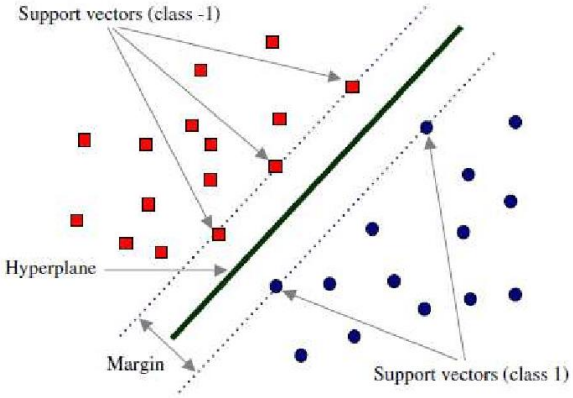


Fig.12: Diagram of an SVM being applied to a binary classification problem. Image taken from https://www.researchgate.net/figure/General-classification-hyperplane-representation-of-SVM-algorithm_fig5_330557084

Hyperplanes are used to segment classes from one another which leads to the SVM predicting classes depending on which side of the hyperplane the new observation is located. However, as there are many potential hyperplanes which can segregate the two classes, SVM must calculate the optimal hyperplane. The optimal hyperplane (just referred to as hyperplane in Fig.12) is located and oriented where the margin between the closest observations of both classes is maximised. The support vectors are the examples which sit on the margin boundaries or are very close to them.

SVM can also be used in situations which are not linearly separable using either kernel tricks (also mentioned in the Feature Extraction section under the Kernel Principal Component Analysis header) or soft margin. The term non-linearly separable relates to situations where instances of both classes are within close proximity to one another to the extent that they cannot be separated by a straight line.

Soft margin can only be used if one of the following situations is present:

- One or a few outlying instances are on the margin line but the wrong side of the decision boundary. An example of this can be seen in the image on the left.
- One or a few outlying instances are on the wrong side of the margin on the wrong side of the decision boundary. The example illustrating this can be seen in the image on the right.

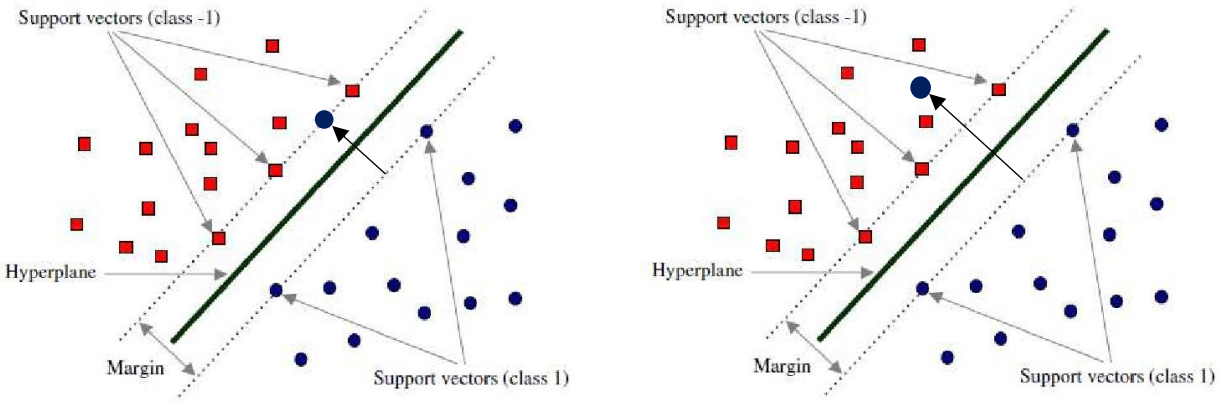


Fig.13: Diagrams of situations where soft margin can be utilised. Image taken from https://www.researchgate.net/figure/General-classification-hyperplane-representation-of-SVM-algorithm_fig5_330557084

Soft margin applied to SVM causes the classifier to find a line which maximizes the margin whilst also reducing the likelihood of misclassifying instances by using a trade-off of the two. The value of the *'C'* hyperparameter within the SVC class in scikit-learn relates to the penalty which SVM obtains when misclassifying instances, leading to a smaller sized margin which also reduces the number of support vectors that can help the classifier to make decisions.

The '*Kernel trick*' on the other hand, relates to using various kernels which offer a non-linear decision boundary. There are five kernels available to choose which include: '*Linear*', '*poly*', '*rbf*', '*sigmoid*', and '*precomputed*', however, only the polynomial and radial basis function are considered here as they are the most common kernels to consider. Note that a linear kernel approach can be seen in the Fig.12 and Fig.13. The polynomial kernel equation can be seen in Equation 10.

$$(a * b + r)^n$$

Equation 10: Polynomial Equation

Where 'a' and 'b' symbolise 2 different instances within the dataset, 'r' determines the coefficient of the polynomial and the value of 'n' is the polynomial degree. Therefore, if the value of 'r' was 1 and the value of 'n' was 2, this would multiply out the brackets to get the polynomial and dot product respectively:

$$2ab + a^2b^2 + 1$$

$$\left(\sqrt{2}a, a^2, 1\right).\left(\sqrt{2}b, b^2, 1\right)$$

This therefore illustrates that the coordinates of the x-axis are increased by a factor of $\sqrt{2}$, whilst the original y coordinates are squared. The z coordinate of 1 is ignored however as it is a constant.

The radial basis function (RBF) kernel works by calculating the distance of all instances to the new observation where the instances closest to it have a stronger weighting to help determine

which class the new observation fits into. The radial basis function uses the equation seen in Equation 11.

$$e^{-\gamma(a-b)^2}$$

Equation 11: Radial Basis Equation

Where 'a' and 'b' are 2 different instances and the subtraction between these being the calculation of the distance between the two which is then squared to ensure there are no negative distances. This distance is then scaled by a factor of gamma (γ) to determine how much influence the instance has on the new observation, with the value of γ being calculated using cross validation. Note that the gamma hyperparameter in the SVC class can be altered to determine the influence that instances have on decisions when the classifier is used with a kernel.

K-nearest neighbour (KNN)

KNN works by calculating the Euclidean distance of the new observation to all of the other instances within the dataset. The Euclidian distance equation can be seen in Equation 12.

$$distance(i,j) = \sqrt{(i1 - j1)^2 + (i2 - j2)^2}$$

Equation 12: Euclidean Distance Equation

The value assigned to 'k' relates to the relevant number of instances closest to the new observation which are to be used to determine the class of the new observation. This value should be an odd number greater than 1 to ensure there is an appropriate definitive class. i.e., if the value of k was equal to 4, there is a possibility that 2 of the 4 closest instances could be of one class and the other 2 closest instances be part of another class, leading to an indecisive class prediction for the new observation. Note that the value of k is important as it could potentially lead to overfitting if the value is too large, or potential underfitting if the value of k is too small.

The value of k could perhaps be determined however, by using a loss function to calculate the variation between the predicted value and the actual value (cross entropy should be incorporated for classification problems).

Decision trees

Decision trees work by firstly selecting which feature should be the root node of the tree (the starting point), using either the gini impurity equation or information gain equation. The feature with the minimum gini impurity value will be used as the tree's root or the feature with the highest information gain value will be set as the root depending on the selected approach to split nodes. To determine which features are then used to split the tree up further, the same splitting process is

repeated until stopping conditions are met such as all of the observations in sub-nodes belonging to one class and one class only, or if all of the feature values are the same. To avoid the possibility of overfitting occurring, pruning should be considered, where the use of pre-pruning prevents the decision tree becoming a fully-grown tree, whilst post-pruning trims down a fully-grown tree. There are a variety of conditions which could be selected to prevent the growth of decision trees including after a specified number of observations or if there is no improvement in the measure of impurity. An example of a decision tree can be seen in Fig.13:

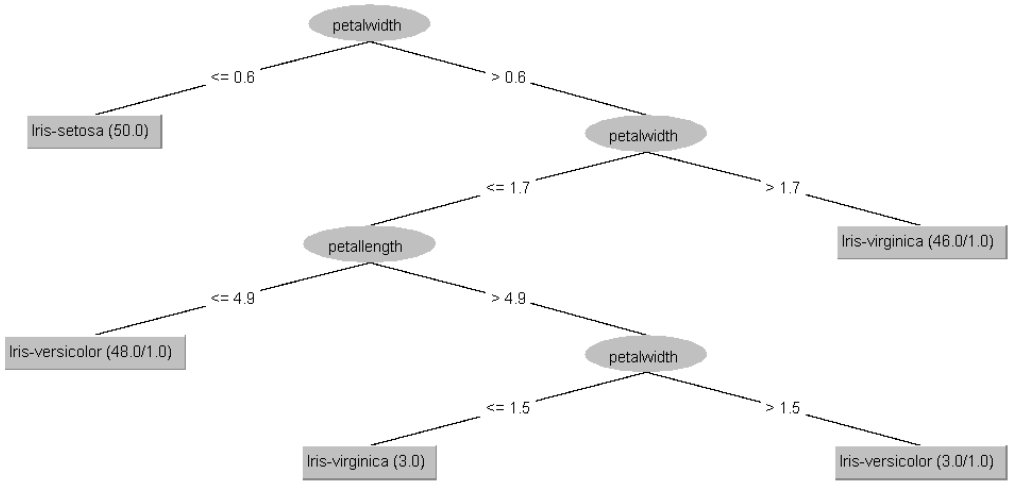


Fig.14: Decision Tree outputted after fitting a J48 tree in weka explorer to the iris dataset

Multi-Layer Perceptron (Neural Network)

A multi-layer perceptron consists of many artificial neurons which essentially use weightings and activation functions to produce outputs. In the image below, a basic diagram illustrating a singular neuron can be viewed:

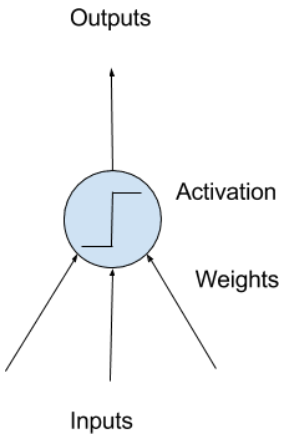


Fig.15: A diagram of an artifical neuron. Image taken from https://machinelearningmastery.com/neural-networks-crash-course/

Every artificial neuron within the network has a bias value of 1.0 and therefore this is considered an input too (in other words, a neuron with 3 inputs actually has 2 inputs and a bias input). The weighting of inputs should ideally be kept small where possible to prevent the complexity of models as activation functions are applied to the sum of all weighted inputs within the perceptron.

Activation functions are used to mimic the activation of neurons in the human brain, for example, if someone likes the taste of a certain food then it will be stimulated as opposed to if the food does not taste as nice. This can be represented as a value between 0 and 1 depending on how much the neuron is activated. It should be noted however, that in binary classification problems, the step activation function is more appropriate as a neuron is either activated or not activated, rather than the possibility of partial activation when multiple classes are present. ReLu (Rectified linear unit) is the most widely used function as it transforms the neuron output value to 0 if the sum of weighted input values is less than 0, or if the sum of the weighted input values is greater than 0, the output value remains unchanged. It is commonly used over other activation functions as it does not encounter the vanishing gradient problem like the sigmoid function does. However, with data which is completely random, it may encounter the dying Relu problem, as the sum of the neurons weighted inputs may be less than 0 and therefore will not learn anything at any point within training. To prevent this, the 'Leaky Relu' and 'ELU' functions are preferred in situations where there are values less than 0 which should not be filtered. It is also to be noted that there is not one specific activation function which works well for all instances and that other functions include tanh, sigmoid and softmax [40].

With the use of the backpropagation algorithm, weights can be adjusted within each epoch, corresponding to the level of error which is calculated between the predicted output and the expected output after the feed forward algorithm has been applied.

The use of hidden layers is also extremely important to include within neural networks where problems are more complexed and are not linearly separable. They are situated between the input layers and output layers and the more hidden layers present, the deeper the network becomes as the data extracted from inputs is outputted to the next layer.

However, in order to use neural networks, all categorical data must firstly be converted into numerical data using encoding. Input data must also be either normalised or standardised to ensure all values have the same standard as one another.

Ensemble methods

Ensemble methods combine multiple classifiers into one meta-classifier which has the ability to generalise data it is fitted to, better than each of the individual classifier components. Majority voting (used for binary classification problems) and Plurality voting (used for multi-class classification problems) are the principles used in the most popular ensemble methods, as they encourage classifiers to predict the class which receives the highest number of votes from the classifier components.

Bagging

Bagging, also referred to as bootstrap aggregating, is commonly used on unstable classifiers as this has been seen to improve their accuracies whilst also decreasing the level of overfitting [41]. It works by applying classifiers onto different subsections of the training data (these subsections are randomly selected with replacement) before combining the predictions from each of these to make a final estimation. Fig.17 illustrates how bootstrap samples work.

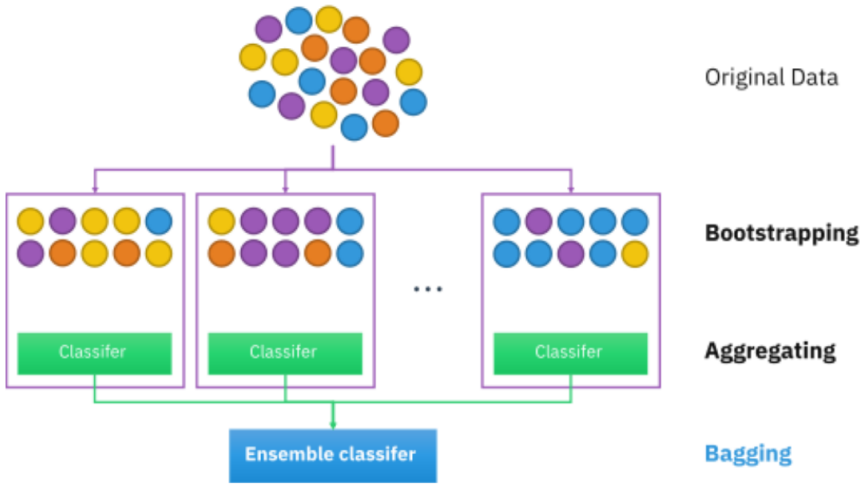


Fig.17: A diagram of an artificial neuron. Image taken from https://machinelearningmastery.com/neural-networks-crash-course/

Due to the replacement after selections, it is clear that subsets can contain none, one or multiple of the same observations. This process is repeated until there is no improvement on the ensemble's performance on a validation dataset. One classifier is then applied to each of these subsets (usually a poor performing basic classifier such as a decision tree which does not include any form of pruning). The predictions of each of these classifiers are then aggregated to make a final prediction which is usually better than what is obtained by one individual classifier which has been well tuned.

The random forest classifier is a unique type of bagging as it also randomly selects feature subsets when each of the individual decision trees are fitted to the training data. As it is extremely resilient to noise, something which is inevitable to be collected when applied in a real time scenario and therefore offers a level of reassurance which other classifiers do not have the ability to cope as well with.

It should also be noted that bagging does not perform particularly well on data where there is an imbalance of classes. However, there are many ways to overcome this problem. OverUnderBagging can be used in an attempt to balance classes out. This term means that oversampling should take place of classes with a minority of instances, whilst undersampling should take place on classes which have the majority of the instances. Using the imbalanced-learn python library, classifiers such as '*BalancedBaggingClassifier*', '*BalancedRandomForestClassifier*' and '*EasyEnsembleClassifier*' can all be used with specific hyperparameters to balance classes.

Stacking Classifier

A stacking classifier works with the use of two levels. Level-1 contains the base classifiers which make their initial predictions of the class, and level-2 is another classifier which is fitted to the level-1 classifier predictions. This second level estimator is the classifier used to make the final set of predictions.

Boosting

Boosting is applied to weak learners in order to create a strong classifier. It works by randomly selecting subsets of training observations from the training dataset without replacement. To explain this further, boosting makes use of the four following steps:

1. Firstly, select a random subset of training observations from the training dataset without replacement and use this to train a weak learner, *Clf1*.
2. Select another random subset in the same way as step 1, but also add 50% of the observations which were misclassified by *Clf1,* to train another weak learner, *Clf2*.
3. The training observations which both *Clf1* and *Clf2* disagreed upon, are then used to train another weak learner, *Clf3*.
4. Each of the weak learners, *Clf1*, *Clf2*, and *Clf3* are then combined and use a majority voting approach to finally predict classes.

AdaBoost (Adaptive Boosting)

AdaBoost firstly fits a decision tree stump (a decision tree which only consists of one level) to the dataset before analysing mistakes made by this classifier. This therefore allows the classifier to attempt to find a solution to all of the instances where errors arose and leave observations alone which were correctly predicted. Another decision tree stump is then fit to the dataset once again with the intention of less errors being present. This is repeated n number of times where n is the value of the '*n_estimators*' hyperparameter (if this is not provided the default is 50). It is different to original boosting in that Adaboost is fitted to the entire training dataset rather than just the training data within the training dataset. Adaboost also gives weightings to each of the training observations in each of the iterations (these weightings can be changed in each iteration), allowing the weak classifiers to learn from misclassified errors, leading to the construction of a strong classifier. Table 6 illustrates how the adaboost method works.

Index	X	y	weightings	Ŷ(2x <=8.0)?	Correct Classification	Reweighted weightings
1	1.0	1	0.1	1	Yes	0.072
2	2.0	1	0.1	1	Yes	0.072
3	3.0	-1	0.1	1	No	0.167
4	4.0	-1	0.1	1	No	0.167
5	5.0	-1	0.1	-1	Yes	0.072
6	6.0	1	0.1	-1	No	0.167
7	7.0	-1	0.1	-1	Yes	0.072
8	8.0	-1	0.1	-1	Yes	0.072
9	9.0	-1	0.1	-1	Yes	0.072
10	10.0	1	0.1	-1	No	0.167

Table 6: Example of Adaboost

Using Table 6, it is clear that the classifier in the 5th column (the index column is the 1st column) is fitted to the data in the 2nd column to determine what the value of Ŷ is where Ŷ relates to the prediction made by the classifier. The 3rd column contains the y values which are the true classification results. Therefore, the results within the 3rd and 5th column are compared with one another to determine if the classes have been correctly predicted which can be seen in the 6th column. The reweighted weightings in column 7 vary depending on whether the classifier has correctly predicted examples or not.

Although this is a binary classification problem where y ∈ {-1,1}, it can still be applied to multi-class classification problems. Note that the number of samples is not restricted to only 10, but if this quantity is changed, the raw weightings in column 4 follow the equation: $\frac{1}{k}$ where k is equal to the number of samples selected to be used.

Gradient Boosting classifier

Gradient boosting is also another popular boosting algorithm which is very similar to adaboost in that the boosting concept is the exact same. The slight difference between the two however, is that gradient boosting does not provide weightings for each of the classifications whether they are correct or incorrect, and also does not calculate weights for each of the models in the ensemble. Instead, the negative log-likelihood loss function (used only for classification problems and not regression) of the weak learners is optimised before adding a new tree, one at a time, meaning a majority voting approach is not required, as the final model is the optimal model within the specified number of '*n_estimators*'. Another difference to both boosting classifiers is that gradient boosting does not require decision tree stumps to be used, but instead allows decision trees themselves. Raschka states in [42] that the three following stages can be used to summarize gradient boosting:

1. *"Construct a base Tree (just the root node)"*
2. *"Build next tree based on errors of the previous tree"*
3. *"Combine tree from step 1 with trees from step 2. Go to step 2."*

Scikit-learn have a '*GradientBoostingClassifier*', although this can be extremely computationally expensive to train models. Therefore, it is preferred to train models with '*HistGradientBoostingClassifier*' as has a training speed faster than the popular XGBoost classifier and debatably produces similar results.

XGBoost and LGBoost

Both XGBoost (Extreme gradient boosting) and LGBoost (Light gradient boosting) are arguably the most popular boosting algorithms currently being used in machine learning as they are not only fast, they also produce highly accurate classifiers.

Cross Validation

Hold-out

The '*train_test_split*' class in scikit-learn is used to split data into training data and testing data using random sampling. The '*test_size*' parameter that is passed determines the size of the test data and is set to 25% of the dataset if it is not passed (common ratios include 70/30 or 80/20). However, a problem with using this approach is that due to the nature of random selection, model accuracies can potentially vary significantly depending on what value is passed for the 'random_state' parameter. An example of where this may become a problem follows:

A multi-class classification problem which contains 100 observations and includes 3 classes. 40 of the observations are of 'Gesture1', another 40 are of 'Gesture2' and the remaining 20 are of 'Gesture3'. If random sampling was used to split the data using a '*test_size*' value of 20%, it is possible for all of the 'Gesture1' and 'Gesture2' observations to be included in the training set, whilst all observations of 'Gesture3' will be included in the test set. This will lead to the model producing an accuracy of 0% on training data. K-fold cross-validation can be used to remove this problem, however, as this still uses random sampling, stratified K-fold cross validation should be considered as it uses stratified sampling instead.

K-fold cross validation

The training dataset is randomly split into 'k' folds without replacement, where k-1 folds are used to train the classifier and the remaining one-fold is used to evaluate the model and this process is repeated 'k' number of times. The average performance of each of these repetitions is then recorded as the model's performance. By incorporating this approach, it ensures that the model is fitted to the entire training dataset, leading to more data being used to train models as opposed to the hold-out method. The value of k is commonly set to 10 due to this being an appropriate trade-off between variance and bias for the majority of real-world machine learning problems [43]. Although, for smaller datasets, it may be worth considering a larger value as this will ensure more training data will be used in each of the iterations but will consequently increase the computational runtime. By decreasing the value of k for larger datasets, this will have an opposite effect.

Stratified K-fold cross validation

The stratified k-fold validation method produces better bias and variance estimates as imbalanced classes do not have any impact upon estimations. By using this cross validation method, it ensures that no matter how many instances of one class are present within the dataset, the splitting of each of these folds into training and testing subsets will be a consistent percentage. Using the same example in the Hold-out section to explain this method further:

The training set for stratified sampling includes 32 observations of 'Gesture1' (80% of 40), 32 observations of 'Gesture2' (80% of 40) and 16 observations of 'Gesture3' (80% of 20). The testing set consists of 8 observations of 'Gesture1' (20% of 40), 8 observations for 'Gesture2' (20% of 40) and 4 observations of 'Gesture3' (20% of 20). To analyse what was done here, the 80% of data used for training was now including 80% of examples from each of the classes whilst the 20% used for testing data was now including 20% of examples from each of the classes.

There is also a repeated stratified k-fold cross validation method which is the exact same as the stratified k-fold cross validation method other than it has a parameter which allows the number of repeats to be specified using '*n_repeats*' and helps to further improve bias and variance estimations in imbalanced data.

Leave One Out Cross Validation (LOOCV)

For very small datasets, the leave one out cross validation method is recommended as the number of folds is equal to the number of training examples. This therefore allows more training data to be used to train the model and only one of the training examples to be used to test the model in each iteration.

Feature Engineering

Feature engineering is the process of creating new features using the original data. These new features offer a clearer understanding of the data domain and this generally leads to an improved model performance on unseen data.

Categorical Encoding

All data in a dataset must be numerical in order for some machine learning classifiers to be trained. Therefore, if features are categorical, they must be encoded to a numerical type as a pre-processing stage. There are several methods which can be used to encode data, although the most commonly used techniques include ordinal encoding and one-hot encoding.

Ordinal Encoding

Ordinal encoding, also referred to as integer encoding, is where unique, categorical, input values are assigned a non-negative integer value as a representation. For example, "Flower" is assigned to 0, "House" is assigned to 1, and "Grass" is assigned to 2. Original categorical values can easily be retrieved after encoding, due to the unique, natural ordering of integers which now represents them. In scikit-learn, categorical values are sorted into alphabetical order before they are encoded. This can be seen in Image 4 where the value 'grass' is encoded before the value 'house'. For

classification prediction problems such as gesture recognition, where the target value is the only attribute containing categorical values, a Label encoder is preferred over an Ordinal Encoder. They are similar as they both ordinally encode categorical values, although the difference between the two is that a label encoder requires an input in the form of a one-dimensional array, as opposed to an ordinal encoder which requires a two-dimensional array as its input.

```
# OrdinalEncoder imported in order to encode labels.
# fit_transform used to fit the categories to the encoder
# and encode them all in one step.
# For loop used to show which encoded value represents
# each categorical Value

from numpy import asarray
from sklearn.preprocessing import OrdinalEncoder

categories = asarray([["Flower", "Vegetable"],["House", "Vegetable"],["Grass", "Fruit"]])
ordEnc = OrdinalEncoder()

transformed = ordEnc.fit_transform(categories)

for x in range(len(transformed)):
    print(str(categories[x]) + " values are respectively assigned " + str(transformed[x]))
```

```
['Flower' 'Vegetable'] values are respectively assigned [0. 1.]
['House' 'Vegetable'] values are respectively assigned [2. 1.]
['Grass' 'Fruit'] values are respectively assigned [1. 0.]
```

```
# LabelEncoder imported in order to encode labels.
# fit_transform used to fit the categories to the encoder
# and encode them all in one step.
# For loop used to show which encoded value represents
# each label

from numpy import asarray
from sklearn.preprocessing import LabelEncoder

categories = asarray(["Flower", "House", "Grass"])
lblEnc = LabelEncoder()

transformed = lblEnc.fit_transform(categories)

for x in range(len(transformed)):
    print(str(categories[x]) + " label is assigned " +str(transformed[x]))
```

```
Flower label is assigned 0
House label is assigned 2
Grass label is assigned 1
```

Fig.46: Example of using the Ordinal Encoder (Left) and Label Encoder (Right).

Using the label encoder example above, it can be seen that there is now a relationship between the three labels. Models will acknowledge the relationship and learn that House > Grass > Flower, which will result in potentially impacting performance in classifiers as there is no ordinal relationship between the labels. To prevent this from occurring, one-hot encoding should be considered.

One Hot Encoding

Instead of an integer representing a unique label, a binary variable is used to represent them. Using the previous Label encoding example in Fig.16, three unique categorical labels can be seen. Therefore, three unique binary variables will be required. Each bit in a binary variable relates to each of the categories, and where a bit value of '1' is located in a binary variable, this relates to the unique class of the observation. With the assistance of Fig.17, it can be seen that [1, 0, 0] is the binary variable for the label 'Flower'. Therefore, the first bit from the left is the bit representing a 'Flower'. The second bit represents a 'Grass' and the third bit represents a 'House'.

```
# OneHotEncoder imported in order to encode labels.
# fit_transform used to fit the encoder to the categories
# and encode them all in one step.
# For loop used to show which encoded binary variable
# represents each label

from numpy import asarray
from sklearn.preprocessing import OneHotEncoder

categories = asarray([["Flower"], ["House"], ["Grass"]])

ohe = OneHotEncoder(sparse=False)
transformed = ohe.fit_transform(categories)

for x in range(len(transformed)):
    print(str(categories[x]) + " label is assigned " + str(transformed[x]))
```

```
['Flower'] label is assigned [1. 0. 0.]
['House'] label is assigned [0. 0. 1.]
['Grass'] label is assigned [0. 1. 0.]
```

Fig.57: Example of one-hot encoding

One-hot encoding does however have a limitation in that it includes representation redundancy. As each binary variable contains a switched-on bit of the value '1', this means that a binary variable which only contains '0's is not utilised. Dummy variable encoding has the ability to remove this multicollinearity.

Dummy Variable Encoding

The number of binary variables required to represent an n number of categories is $n - 1$, and therefore, if the first bit is dropped for each of the binary variables, all representations of labels remain valid and unique. Fig.18 illustrates dummy variable encoding, using the previous example in Fig.17, but the difference with this example is that an extra argument has been passed into the initialisation of the one-hot encoder to drop the first bit.

```
# OneHotEncoder imported in order to encode labels.
# fit_transform used to fit the encoder to the categories
# and encode them all in one step.
# For loop used to show which encoded binary variable
# represents each label

from numpy import asarray
from sklearn.preprocessing import OneHotEncoder

categories = asarray([["Flower"], ["House"], ["Grass"]])

ohe = OneHotEncoder(sparse=False, drop='first')
transformed = ohe.fit_transform(categories)

for x in range(len(transformed)):
    print(str(categories[x]) + " label is assigned " + str(transformed[x]))
```

```
['Flower'] label is assigned [0. 0.]
['House'] label is assigned [0. 1.]
['Grass'] label is assigned [1. 0.]
```

Fig.68: Example of Dummy Variable Encoding

Normalization and Standardization of data

Scaling numerical features to be within a standard range is crucial within machine learning as it usually results in improvements of model performances. Some methods such as PCA require scaling as a pre-processing step and if it was to be avoided, this could potentially lead to important components being discarded.

Normalisation scales each individual feature to fit within a range of 0 and 1 with the use of the formula in Equation 13.

$$y = (x - \min) / (\max - \min)$$

Equation 13: Normalization Equation taken from
https://machinelearningmastery.com/standardscaler-and-minmaxscaler-transforms-in-python/

The '*MinMaxScaler*' class in '*scikit-learn*' can be used to normalise the data. An example of this can be seen in Fig.19.

```
import numpy as np
from sklearn.preprocessing import MinMaxScaler

featureExamples = np.asarray([[-5,   2,  -1],
                              [3,   7,  -9],
                              [5,   5,   1],
                              [18,  12,  3],
                              [2,   8,   4]])

minMax = MinMaxScaler()
normalised = minMax.fit_transform(featureExamples)
print("The normalised varsion of the features is:\n\n",normalised)
```

```
The normalised varsion of the features is:

 [[0.         0.         0.61538462]
 [0.34782609 0.5        0.        ]
 [0.43478261 0.3        0.76923077]
 [1.         1.         0.92307692]
 [0.30434783 0.6        1.        ]]
```

Fig.19: MinMaxScaler fitted to and transforming an array

Standardization on the other hand, scales each individual feature by subtracting their mean and dividing them by their standard deviation in order for the features to obtain a mean of 0 and standard deviation of 1. The equation for this can be seen in Equation 14.

$$y = (x - \text{mean}) / \text{standard\_deviation}$$

Equation 14: Standardization Equation taken from
https://machinelearningmastery.com/standardscaler-and-minmaxscaler-transforms-in-python/

The '*StandardScaler*' class in *scikit-learn* can be used to standardize the data. An example of this is illustrated in Fig.20.

```
from sklearn.preprocessing import StandardScaler

featureExamples = np.asarray([[-5,   2,  -1],
                              [3,   7,  -9],
                              [5,   5,   1],
                              [18,  12,  3],
                              [2,   8,   4]])

standardscale = StandardScaler()
standardized = standardscale.fit_transform(featureExamples)
print("The standardized version of the features is:\n\n",standardized)
```

```
The standardized version of the features is:

 [[-1.28011379 -1.44989302 -0.12958026]
 [-0.2133523   0.06041221 -1.85731711]
 [ 0.05333807 -0.54370988  0.30235395]
 [ 1.7868255   1.57071744  0.73428816]
 [-0.34669749  0.36247326  0.95025527]]
```

Fig.20: StandardScaler fitted to and transforming an array

Dimensionality reduction

Dimensionality reduction is the process where features within the dataset are reduced either by the use of feature extraction or feature selection techniques. This is fundamental for complexed datasets containing several hundred features as it not only speeds up the training process of models, it also provides an increase in the reliability of the accuracy scores obtained after training. This therefore also leads to a decrease in the likelihood of overfitting occurring.

Feature Selection and feature extraction are different in that the algorithms used for feature selection, such as sequential backward selection, ensures original features are kept, whereas feature extraction transforms the features into a new feature subspace.

Feature Selection

- Features which have a high percentage of missing values should be considered to be discarded as it makes the training of models considerably more complexed if data is missing. However, this is not a certainty to be removed from the dataset as it is a possibility that the feature may only represent two possible states and therefore a missing value may actually be important. An example of this could be an 'isRelevent' feature where a 1 represents relevance and no value symbolises no relevance. To prevent this, missing values could be encoded. If the feature accepts more than two states, missing values can be imputed using the scikit-learn class, '*SimpleImputer*' with the '*strategy*' parameter indicating the technique used to determine the imputed value. 'Mean', 'median' and '*most_frequent*' are the possible strategies.
- Features which have low or no variation should be discarded as models will not learn much from these features when being trained. It should be noted that before this feature selection process is applied, the data must be standardised as it is expected that different features will have different scales and therefore this will impact their variances.
- If features are highly correlated with one another, this leads to overfitting. Therefore, pairs of features which have a high correlation with another should be handled by removing one of these features (the feature which has a lower correlation to the label). This ensures that information is not lost but also reduces the number of dimensions.
- Recursive feature selection (RFE) is commonly used for feature selection which recursively discards the least important features until the quantified number of features (the '*n_features_to_select*' hyperparameter) is met. The algorithm is considered a wrapper-style algorithm as it uses a machine learning model to determine which features should be eliminated. Firstly, the machine learning model is fitted to the entire training set. Each of the features are then ranked by their importance (statistical tests or model-based scores) with the least important features being eliminated. This is recurrent until the desired number of features are retained. However, the problems which occur here are that it is common to not know the optimal number of '*n_features_to_select*'. Therefore, recursive feature elimination with cross validation (RFECV) should be considered instead as RFE scores each of the subsets and chooses the collection of features within the subset which obtained the best score.

- The '*SelectFromModel*' approach is similar to RFE in that it uses a machine learning model which has either the '*coef_*' or '*feature_importance'* attribute to remove features.

As it is inefficient and prone to human error to manually search through large datasets for missing values, the '*isnull*' method in the pandas library should be used to determine how many missing values are present within each feature. This can be done using either of the following python lines:

```
In [3]: df.isnull().sum()
Out[3]: left_palm_position_x                0
        left_palm_position_y                0
        left_palm_position_z                0
        left_palm_normal_x                  0
        left_palm_normal_y                  0
                                           ..
        right_pinky_distal_end_z            0
        right_pinky_distal_direction_x      0
        right_pinky_distal_direction_y      0
        right_pinky_distal_direction_z      0
        label                               0
```

```
In [3]: df.isnull().any().any()
Out[3]: False
```

Fig.21: Checking for missing values

Feature Extraction

- Principal component analysis (PCA) is an unsupervised approach which simply compresses the number of features in the dataset onto a new feature subspace. However, it is vital that features are standardised before applying the PCA to them, as it is inevitable that features will have ranges that vary considerably. The outcome is n components where n is equal to the number of features in the dataset. The eigenvalues (the level of variance the component explains) should preferably be sorted in decreasing order to ensure that component 1 (components can also be referred to as eigenvectors) accounts for the most variance explained within the dataset and the 'n'th component having the least variance explained within the dataset. This therefore allows many of the components which have very minimal data variance to be discarded in a simplified manner.

- Linear Dimensionality Analysis (LDA) is similar to PCA with the only difference being that it is a supervised approach to reducing dimensions whilst maximising the separation between classes. It is intended to be used for classification problems (either binary or multi-class) where categorical labels are present. However, like PCA, it is vital to standardize the features before applying LDA to ensure that each of the features have the same variance standard, and to also remove any outliers as this will prevent the mean and standard deviation of features being affected. LDA works by firstly estimating both the mean and variance of each of the classes within the dataset, and then uses bayes theorem to estimate the probability of new data belonging to each of the classes.

- Kernel principal component analysis (KPCA) is used to transform features into a lower dimensional subspace which are not linearly separable (PCA and LDA are to be used if it is linearly separable), which then allows linear classifiers to be applied appropriately to the data. The data is transformed into a higher dimensionality subspace using KPCA, allowing either PCA or LDA to then be used to transform the data into a lower dimensionality subspace. However, as this is extremely computationally expensive, the 'kernel trick' is utilised which calculates the level of similarity amongst two high-dimensional features within the raw feature space using the dot product. The '*KernelPCA'* class within '*sklearn.decomposition*' allows a very simple transformation of the features, similar to the PCA approach with a slight difference being that a kernel parameter can be specified. The most popular kernels include:

    - Radial Basis Function (RBF) also known as the gaussian kernel.
    - Polynomial kernel
    - Sigmoid Kernel

  An example of KPCA being applied to features within the '*make_moons*' dataset can be seen in Fig.22.

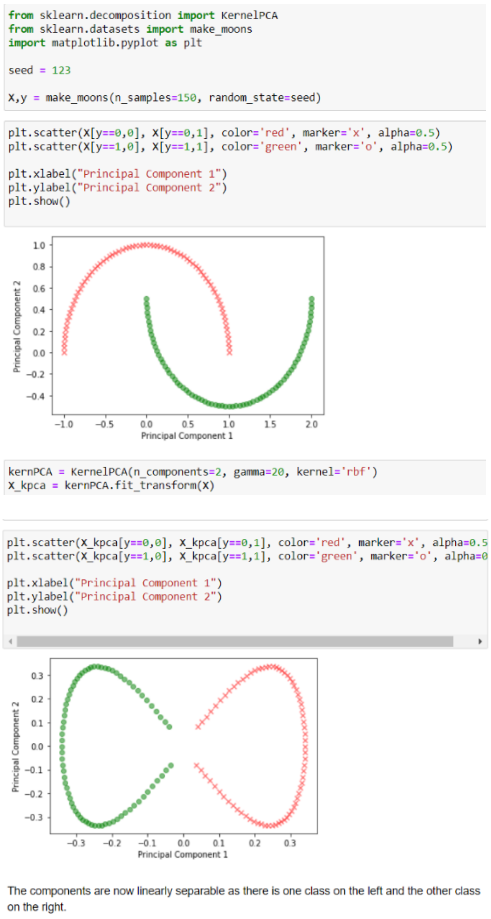


Fig.22: Kernel PCA applied to the make_moons dataset

Pipelining

Scikit-learn has a '*make_pipeline*' class, which allows the combination of multiple transformers such as scalers or dimensionality reducing methods, rather than them being written independently over multiple lines of code. The data is passed through each of the fitting and transformation steps within the pipeline until an estimator is reached, therefore, the estimator should be the final feature within a pipeline.

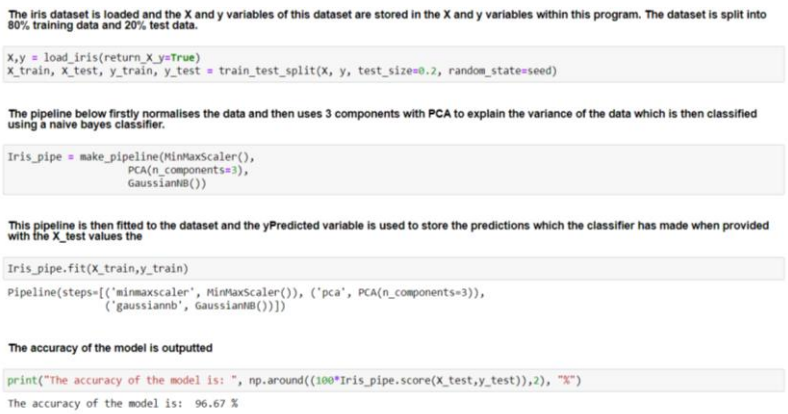


Fig.23: An example of implementing a pipeline

Using Fig.23, it can be seen that the pipeline consists of a '*MinMaxScaler*' transformer, PCA transformer and a Naïve Bayes classifier with the data being processed in this exact order. The file for this example can be found within the 'Pipelining' Notebook.

Discrete Probability Distributions

Bernoulli Distribution

Bernoulli probability distribution is used to predict the probability that a single event will have a binary outcome of either a 0 or a 1. An example of this could be flipping a coin, where the two classes can be determined as "heads", which would be the normal state or "not heads", which would be the abnormal state (note that it has been assumed that the coin is fair).

Binomial Distribution

Binomial distribution is where multiple independent bernoulli events are repeated. The probability of a successful outcome multiplied by the number of trials, returns the expected number of successful trials. Fig.24 illustrates this with the previous example of flipping a coin.

```
from numpy.random import binomial

# probability is set to 0.5 as there is a
# 50% chance of either heads or not heads.
# Value of n can vary.
# An increased value of n will result in a
# headSuccess value closer to 50% of n.

probability = 0.5
n = 1000
headSuccess = binomial(n, probability)
print('Successful Head Flip Total: %r' % headSuccess)
```

```
Successful Head Flip Total: 499
```

Fig.24: Binomial distribution of flipping a coin

Multinoulli Distribution

Events which have an n number of potential outcomes, where n is greater than 2, are said to be of a multinoulli distribution. An example of this is rolling a die as there are 6 possible outcomes (assuming that a fair die is used). Probabilities are assigned to each of the possible outcomes and the sum of all probabilities must be equal to 1, therefore, the likelihood of each number being rolled after one independent trial, is a sixth. Gesture recognition typically has a multinoulli distribution as there are usually more than 2 gestures to distinguish between.

Multinomial Distribution

Multinomial distribution is where multiple independent multinoulli events are repeated. It is expected that each of the numbers on the die should be rolled approximately a sixth of the value of n times using the previous example. This scenario can be seen in Fig.25.

```
# n = number of repetitions
# p = probabilities of the outcomes

from numpy.random import multinomial
p = [1/6, 1/6, 1/6, 1/6, 1/6, 1/6]
n = 1000

outcomes = multinomial(n, p)

for x in range(len(outcomes)):
    print('Number %r: %r' % (x+1, outcomes[x]))
```

```
Number 1: 172
Number 2: 160
Number 3: 169
Number 4: 171
Number 5: 164
Number 6: 164
```

Fig.25: Multinomial Distribution of rolling a die

Meta-Strategies

Datasets which contain multiple classes must be split down into smaller binary classification problems in order for binary classifiers to be fitted to the data. There are two ways in which these strategies can be applied to models, the first is to pass a parameter, such as '*decision_function_shape*' or '*multi_class*', into the binary classifier when it is initialised. The second method is to import a class in scikit-learn which allows the specific strategy to be used with any classifier.

One-vs-One (Ovo)

The number of binary classifiers to be generated from the original dataset can be calculated using Equation 15.

$$Number\ of\ Classifers\ Required = \frac{N\ (N-1)}{2}$$

Equation 15: Equation for the number of classifiers to be generated

Where N is the number of classes in the multi-class classification problem. Therefore, if the example of 'Flower', 'House' and 'Grass' is considered again, the value of N is 3 and the number of classifiers required is 3 which can be seen in Fig.26.

Binary Classifier 1 *– Flower vs. House*

Binary Classifier 2 *– Flower vs. Grass*

Binary Classifier 3 *– House vs. Grass*

Fig.26: The binary classifiers generated for the 'Ovo' met-strategy

Each of the three binary classifiers are then fitted to the dataset and the majority vote of these classifiers is the final prediction.

One-vs-Rest (OvR)/ One-vs-All (OvA)

A binary classifier should be implemented for each of the classes and therefore the number of binary classifiers should be equal to the number of classes. If the previous example is considered which contains three unique classes, there should also be three binary classifiers generated which can be seen in Fig.27.

Binary Classifier 1 *– [Flower] vs. [House, Grass]*

Binary Classifier 2 *– [House] vs. [Flower, Grass]*

Binary Classifier 3 *– [Grass] vs. [Flower, House]*

Figure 27: The binary classifiers generated for the 'OvR' meta-strategy

Training datasets are then to be created for each of the classes, where a value of one in the class column illustrates that the observation belongs to that class and that class only (all other classes in the column should instead be assigned a value of negative one). Using this approach provides certainty of predicting the correct classes as the only class with a positive value of one will be the correct label. An example of this can be seen in Table 7.

Features			House
x1	x2	x3	-1
x4	x5	x6	-1
x7	x8	x9	+1
x10	x9	x12	-1
x13	x14	x15	-1
x16	x17	x18	+1

Table 7: Training Dataset for the House Class

Using table 7, features x7, x8 and x9 belong to the House class which the positive value of one represents. The same applies to features x16, x17 and x18. The binary classifiers are then fitted to their corresponding dataset for training before being fitted to testing data. The final prediction of the class is made by the model which has the highest percentage of certainty (the prediction probabilities are passed into the argmax function in order to obtain the highest probability).

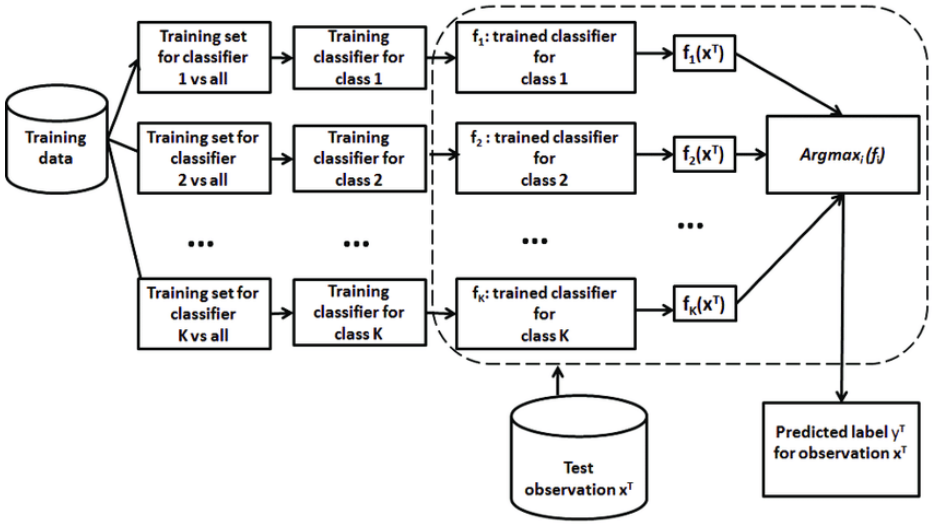


Fig.26: 'OvR' approach to multi-class classification problems. Image taken from https://www.researchgate.net/figure/The-considered-one-vs-all-multiclass-classification-approach_fig2_257018675

Evaluation Metrics

There are four significant terms when discussing model evaluation metrics which include the following:

- A ***True Positive (TP)*** where the prediction made is positive and the actual outcome is supposed to be positive
- A ***False Negative (FN)*** where the prediction made is negative but the actual outcome is supposed to be positive.
- A ***True Negative (TN)*** where the prediction made is negative and the actual outcome is supposed to be negative.
- A ***False Positive (FP)*** where the prediction made is positive and the actual outcome is supposed to be negative.

Confusion Matrix

Confusion matrices are commonly used to summarize results of classification predictions and are useful to determine errors that are being made by classifiers. Predictions are made for each observation in the testing dataset and these are compared with the expected outcomes. Table 8 illustrates an example of this.

Expected	Predicted
House	House
Grass	Flower
Flower	Flower
Flower	Grass
House	House
Grass	Flower
Flower	Flower
House	House
Grass	Flower
Grass	Grass

Table 8: Table of the expected values compared to the predicted values

6 out of the 10 predictions were correct and therefore the classifier has an accuracy of 60%. However, in order to determine what errors were made, a confusion matrix should be constructed which can be done by firstly evaluating each class before evaluating the incorrect class predictions. This can be seen in Fig.27 and Fig.28 respectively.

'Flower' classified as **'Flower'**: *2*

'House' classified as **'House'**: *3*

'Grass' classified as **'Grass'**: *1*

Figure 27: Evaluation of each of the individual class predictions which were TP

'Flower' classified as **'House'**: *0*

'Flower' classified as **'Grass'**: *1*

'House' classified as **'Flower'**: *0*

'House' classified as **'Grass'**: *0*

'Grass' classified as **'House'**: *0*

'Grass' classified as **'Flower'**: *3*

Figure 28: Evaluation of each of the individual class predictions which were Incorrect

Using the values in both Table 8, Fig.27 and Fig.28, a confusion matrix can be generated, where the predicted classes are along the bottom of the matrix, whilst the expected classes are along the left side of the matrix which is labelled in Fig.30. With the assistance of the confusion matrix displayed in Fig.29 and Fig.30, by summing the number of instances within the rows, it can be determined how many observations of each class are present within the testing dataset. Therefore, there should be 3 observations of the 'Flower' class, 3 observations of the 'House' class and 4 observations of the 'Grass' class which is supported using Table 8. However, using the number of instances within the columns illustrates what the classifier is actually predicting. Using Fig. 29 and Fig.30 again, it can be seen that 5 observations of the 'Flower' class, 3 observations of the 'House' class, and 2 observations of the 'Grass' class are being predicted. Therefore, a conclusion can be drawn from this confusion matrix that errors are being made by predicting 'Grass' as 'Flower', and predicting 'Flower' as 'Grass'. 'House', however, is predicted with 100% accuracy as there are no incorrect predictions.

Output a confusion matrix

- 0 = 'Flower' Class
- 1 = 'House' Class
- 2 = 'Grass' Class

```
In [9]: import matplotlib.pyplot as plt
        import pandas as pd
        import seaborn as sn
        from sklearn.metrics import confusion_matrix

        expect = [2,3,1,1,2,3,1,2,3,3]
        predict = [2,1,1,3,2,1,1,2,1,3]
        confusionMat = confusion_matrix(expect, predict)
        print(confusionMat)

        df_cm = pd.DataFrame(confusionMat)
        plt.figure(figsize=(10,7))
        sn.heatmap(df_cm, annot=True, cmap="YlGnBu", annot_kws={"size": 16})
        plt.xlabel("Predicted")
        plt.ylabel("Expected")

[[2 0 1]
 [0 3 0]
 [3 0 1]]
```

Fig.29: Confusion Matrix Implementation in python

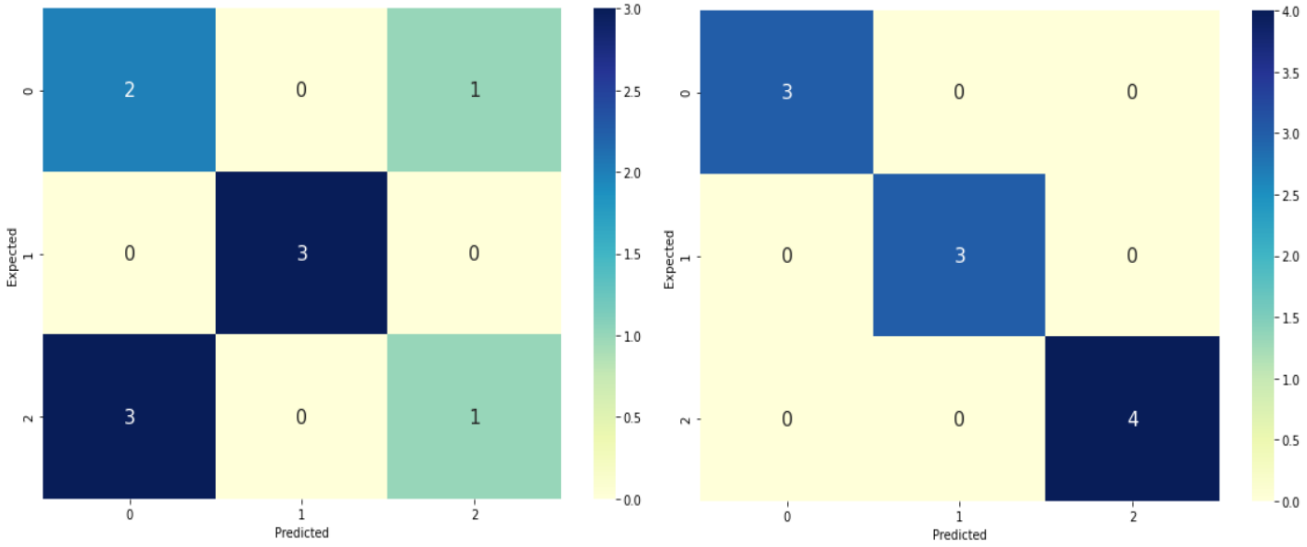


Fig.3010: The confusion matrix relating to the example can be seen on the Left. The image on the right is the optimal solution.

Accuracy

The accuracy of models is frequently used as an evaluation metric when balanced data is present. The calculation to achieve the model accuracy is simply the sum of both the true positive and true negative predictions divided by the sum of all the predictions. The equation can be seen in Equation 16.

$$\frac{TP + TN}{TP + FP + TN + FN}$$

Equation 16: Accuracy Equation. Image taken from https://towardsdatascience.com/various-ways-to-evaluate-a-machine-learning-models-performance-230449055f15

Using the previous example, it can be seen that the model has a 60% accuracy (Inputting the values into the equation (2+3+1)/10). Note that the accuracy can also be calculated by subtracting the prediction error from 1 where the prediction error is the sum of all incorrectly predicted observations (false negative and false positive) divided by the sum of all of the predictions. The equation for this can be seen in Equation 17.

$$1 - \frac{FN + FP}{TP + FP + TN + FN}$$

Equation 17: Another Equation to calculate accuracy.

Precision

The Precision of a model states the percentage of correct positive predictions out of the actual number of positive cases predicted. The equation for this can be seen in Equation 18.

$$\frac{TP}{TP + FP}$$

Equation 18: Equation for Precision. Image taken from https://towardsdatascience.com/various-ways-to-evaluate-a-machine-learning-models-performance-230449055f15.

Recall

Recall is also referred to as sensitivity or the True Positive Rate and is calculated as the percentage of correct positive predictions out of the correct number of positives within the data. The equation for this can be seen in Equation 19.

$$\frac{TP}{TP + FN}$$

Equation 19: Equation for Recall. Image taken from https://towardsdatascience.com/various-ways-to-evaluate-a-machine-learning-models-performance-230449055f15.

It should be noted that recall and precision impact one another. In other words, if the recall value was to increase, the precision value would decrease. Therefore, the F1 score would be the preferred metric.

Specificity

The specificity is the percentage of correctly predicted negatives out of the total number of negatives within the data which is essentially the same as recall but focuses on the negatives rather than the positives. Equation 20 illustrates the equation to calculate this.

$$\frac{TN}{TN + FP}$$

Equation 20: Equation for Specificity. Image taken from https://towardsdatascience.com/various-ways-to-evaluate-a-machine-learning-models-performance-230449055f15.

F1 score

The F1 score is used to offer a balance between precision and recall scores by providing a mean of the two. Equation 21 illustrates how to obtain the f1 score.

$$\frac{2 * precision * recall}{precision + recall}$$

Equation 21: Equation for F1 score. Image taken from https://towardsdatascience.com/various-ways-to-evaluate-a-machine-learning-models-performance-230449055f15.

However, as there is a balanced weighting for both precision and recall values, some situations may require one of these to have more importance over the other and therefore a weighted F1 score may be more appropriate to consider.

ROC Curve (Receiver operating characteristic curve)

ROC graphs can be useful tools as they illustrate how models perform in respect to True Positive Rates (TPR) and False Positive Rates (FPR). The diagonal of an ROC curve is considered 'random guessing' and any classifier which falls below this line is measured as being worse than randomly guessed and should be discarded. Ideally, optimal classifiers will sit at the top left corner of the graph as this states that it has a minimal FPR of 0 whilst maximising the TPR at 1. The area under the curve can then be computed to determine the actual performance of the model. An example of an ROC curve can be found in Fig.31.

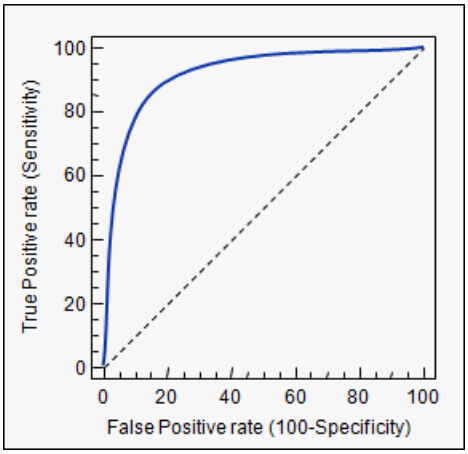


Fig.31: Example of an ROC curve. Image taken from https://www.medcalc.org/manual/roc-curves.php

Hyperparameter Tuning

Grid Search

All of the possible hyperparameter combinations within a search space are tried on the specified model which is being tuned (essentially using brute force). Therefore, this is extremely expensive in terms of both time and space complexity for models which have a large search space, especially if a large dataset is being used to train the model. Another factor which will significantly increase the time complexity is the number of folds the chosen cross validation process has as this essentially multiplies the number of models by the number of folds. An example which explains this can be seen using Fig.32 and Table 9.

Hyperparameters for a Support Vector Machine:

svmParam_grid = { 'decision_function_shape' : ['Ovr', 'Ovo'],

'max_iter' : [100, 200, 300]}

Fig.32: Example of a search space to tune SVM hyperparameters

	decision_function_shape	max_iter
Model 1	Ovr	100
Model 2	Ovr	200
Model 3	Ovr	300
Model 4	Ovo	100
Model 5	Ovo	200
Model 6	Ovo	300

Table 9: Combinations of Hyperparameters within the Grid Search approach

The various combinations can be seen in Table 9. If a K-fold cross validation method with 5 splits is used for this scenario, there will be 30 models fitted to the data in total. It must be noted that there are many more hyperparameters that can be tuned for a support vector machine and that the values for these hyperparameters can also be extensive. However, those used in the example were selected for the purpose of a basic explanation. The best parameters which are returned, are the hyperparameter values in the best performing model fitted to training data.

Random Search

A random search is used in preference of a grid search as it considerably less expensive to perform on classifiers which have a large search space, and still produces relatively accurate results in a fraction of the time that a grid search takes. The number of iterations that are passed to the random Search function also offers greater control over the computational cost of searching hyperparameters as the smaller the value, the less computationally expensive. However, a very small number of iterations will almost certainly lead to a poor classification accuracy, especially if the classifier is fitted to a large dataset.

	decision_function_shape	max_iter
Model 1	Ovr	300
Model 2	Ovo	200
Model 3	Ovr	200

Table 10: Combinations of Hyperparameters within the random search approach

With the aid of Table 10, it is clear that the number of iterations is three as there are three models present. Therefore, this tuning approach will be processed faster but does not necessarily mean the computational duration will be halved, as the time complexity of each of the models is not equal. Therefore, it is possible that the three models selected within Table 10 may be the three most computationally expensive models within the search space. Limitations of this approach however, are that not all of the values for hyperparameters are tested, meaning the global optimum may be missed. In the scenario above, the value 100 was not randomly selected for the '*max_iter'* hyperparameter but may have perhaps produced the best training accuracy with either of the '*decision_function_shape'* values. However, this is something which the programmer would not be aware of unless the random search process was repeated or if the number of iterations were increased.

Bayesian Search

A Bayesian search is a more efficient approach to tune hyperparameters than the Grid Search method as not all hyperparameter values are trialled, leading to a significant reduction in the time duration for tuning models. The number of iterations can be passed to the Bayesian search function, just like the random search approach previously discussed. However, with the use of cross validation to determine which hyperparameter values should be selected from the search space, the Bayesian

search provides a more optimised solution for tuning hyperparameters compared to the random search and grid search approaches.

Nested Cross Validation

Nested Cross validation is a useful approach to determine which classifiers are appropriate to be selected for the given dataset. It works by using an inner and outer loop. The outer loop uses k-fold cross validation to split the data into training and test folds, whilst the inner loop uses k-fold cross validation to select the model by splitting the training data into training and validation folds used to tune the hyperparameters. The accuracy which is averaged and returned, offers an appropriate estimation of how well a model will perform on unseen data if it's hyperparameters were tuned. However, the more hyperparameters provided to tune, especially when a large dataset is present, results in a computationally expensive process and therefore may not necessarily be worthwhile. An example of nested cross validation can be seen in the 'Nested Cross Validation' notebook.

Validation Curves

The '*validation_curve*' function also uses stratified k-fold cross validation to make an estimation of the classifiers performance, just like the '*learning_curve*' function did.

Note that the value for the '*param_name*' parameter must be the classifier written in lower case characters, followed by two underscore characters and the estimator parameter which is being tuned. An example of this being *'svc__C'* where the validation curve is to be drawn for the inverse regularisation parameter within the SVC estimator.

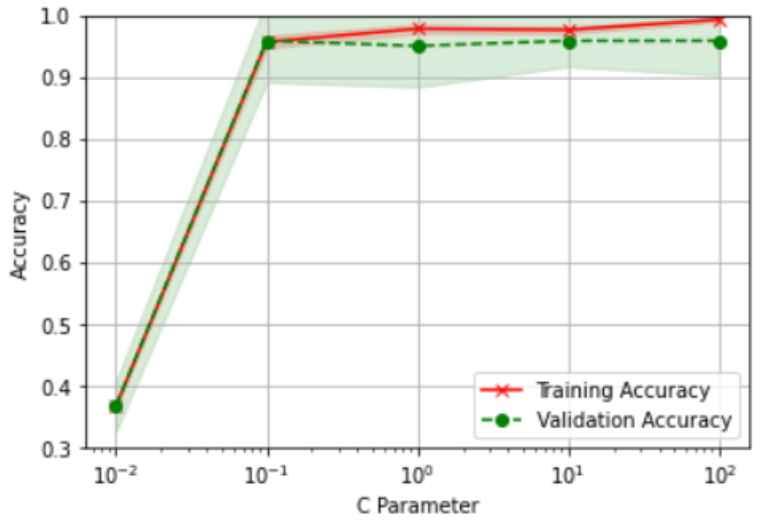


Fig.33: Validation Curve for the C parameter within the SVM classifier being applied to the load_iris dataset

Fig.33 illustrates the classifiers performance with values of the *'C'* parameter ranging from 0.01 to 100.0. By using this graph, it is clear that when the *'C'* parameter is set to 0.01, the model severely

underfits the data. However, when a value of 0.1 is assigned to the parameter, both the training and validation accuracies appears to be of a high percentage with minimal variation between the two. Therefore, it would be appropriate in this scenario to consider using this value for the *'C'* parameter. This example can be found in the 'Learning Curves & Validation Curves' notebook.

Deliverable

Application in Weka

Weka Explorer was firstly used to gain an understanding of how machine learning techniques and models differ using the ionosphere dataset. However, as this dataset is a binary classification problem, whereas gesture recognition is a multi-class classification problem, the results were irrelevant to an extent. Therefore, the iris, glass and soybean datasets, which are all multi-class classification problems, were also considered. The model accuracies outputted after being fitted to the additional datasets were similar to those displayed with the ionosphere dataset, showing that the type of classification problem does not necessarily impact the accuracy of models. Tables 11, 12, 13 and 14 present the metrics obtained from fitting various classifiers to these datasets.

(Note that the random seed for each of these models was kept at the default value of 1 and all other hyperparameters of classifiers were unchanged from their defaults)

Classifier	Accuracy	Recall	Precision	F-Measure	ROC area
Naïve Bayes	82.62%	0.83	0.84	0.83	0.94
Decision Tree	91.45%	0.92	0.92	0.91	0.89
K-nearest Neighbour	86.32%	0.86	0.87	0.86	0.83
Support Vector machine	88.60%	0.89	0.89	0.88	0.85
Multi-Layer Perceptron	91.17%	0.91	0.92	0.91	0.92
Random Forest	92.88%	0.93	0.93	0.93	0.98
Voting Classifier *(Decision Tree, Random Forest, Multi-Layer Perceptron, Naïve Bayes, Support Vector Machine, K-nearest Neighbour)*	92.59%	0.93	0.93	0.93	0.97
Adaboost *(Random Forest)*	93.45%	0.93	0.94	0.93	0.98
Logistic Regression	88.89%	0.89	0.89	0.89	0.87

Table 11: Results of models applied to the Ionoshpere Dataset

Classifier	Accuracy	Recall	Precision	F-Measure	ROC area
Naïve Bayes	96.00%	0.96	0.96	0.96	0.99
Decision Tree	96.00%	0.96	0.96	0.96	0.97
K-nearest Neighbour	95.33%	0.95	0.95	0.95	0.97
Support Vector machine	96.00%	0.96	0.96	0.96	0.98
Multi-Layer Perceptron	97.33%	0.97	0.97	0.97	1.00
Random Forest	95.33%	0.95	0.95	0.95	0.99
Voting Classifier (Decision Tree, Random Forest, Multi-Layer Perceptron, Naïve Bayes, Support Vector Machine, K-nearest Neighbour)	95.33%	0.95	0.95	0.95	1.00
Adaboost (Random Forest)	95.33%	0.95	0.95	0.95	0.99
Logistic Regression	96.00%	0.96	0.96	0.96	0.98

Table 12: Results of models applied to the Iris Dataset

Classifier	Accuracy	Recall	Precision	F-Measure	ROC area
Naïve Bayes	48.60%	0.49	0.50	0.45	0.76
Decision Tree	66.82%	0.67	0.67	0.67	0.81
K-nearest Neighbour	70.56%	0.71	0.71	0.70	0.79
Support Vector machine	71.96%	0.72	0.72	0.72	0.85
Multi-Layer Perceptron	67.76%	0.68	0.67	0.87	0.93
Random Forest	79.91%	0.79	0.80	0.79	0.94
Voting Classifier *(Decision Tree, Random Forest, Multi-Layer Perceptron, Naïve Bayes, Support Vector Machine, K-nearest Neighbour)*	75.23%	0.752	0.754	0.748	0.904
Adaboost *(Random Forest)*	80.37%	0.80	0.81	0.80	0.94
Logistic Regression	64.49%	0.65	0.63	0.63	0.83

Table 13: Results of models applied to the Glass Dataset

Classifier	Accuracy	Recall	Precision	F-Measure	ROC area
Naïve Bayes	92.97%	0.93	0.94	0.93	0.99
Decision Tree	91.51%	0.92	0.92	0.91	0.98
K-nearest Neighbour	91.2%	0.91	0.92	0.91	0.98
Support Vector machine	93.56%	0.94	0.94	0.94	0.99
Multi-Layer Perceptron	93.41%	0.93	0.94	0.93	0.99
Random Forest	92.97%	0.93	0.93	0.93	1.00
Voting Classifier *(Decision Tree, Random Forest, Multi-Layer Perceptron, Naïve Bayes, Support Vector Machine, K-nearest Neighbour)*	93.70%	0.937	0.941	0.936	1.00
Adaboost *(Random Forest)*	92.68%	0.93	0.93	0.93	0.98
Logistic Regression	93.85%	0.94	0.94	0.94	0.99

Table 14: Results of models applied to the Soybean Dataset

The Multi-Layer Perceptron classifier and voting classifier took a fairly long time to return results on the soybean dataset which only consists of 683 observations and 36 attributes. Therefore, as these models were computationally expensive with only a small dataset, and produced similar accuracies to other classifiers which processed the data faster, it would be inappropriate to consider these classifiers as primary contenders. However, this does not rule them out completely, as models such as naïve bayes, which is extremely fast to train and is capable of generalizing data well, also tends to commonly underperform due to overgeneralizing data. An instance of this can be seen with the 48.60% on the glass dataset.

It is clear from the tables above that there are also no specific models which have the best accuracy for every dataset. However, it is evident that the random forest, voting classifier and adaboost models performed consistently well on each of the datasets, which stood out in particular with the glass data where other classifier performances diminished.

Data Collection

The third-party python script written to collect the leap motion controller data and store it in a comma separated variable file, was written by Jordan Bird and can be found in the 'Jordan Birds Third Party Script' folder. However, the original script incorporated the OpenCV library which is utilised to capture images using a webcam. This feature was removed from the third-party script to ensure all data collected of subjects is non identifiable and non-invasive and as images of gestures

are not required for this study would be unnecessary to collect. For each session of recording data, the label and subject variables within the script were adapted according to the gesture and the subject who was demonstrating the gesture (names were not included to eliminate the possibility of subjects being identified and instead were replaced with non-identifiable markers such as 'subject1' or 'subject2').

A basic dataset consisting of 6 different hand gestures was originally collected to test the various classifiers using the python programming language. The gestures which were demonstrated within this dataset consisted of numbers 1-3 in the form of Chinese finger counting. Fig.34 illustrates what these gestures look like

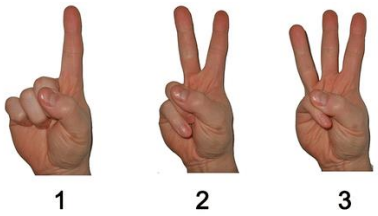


Fig.11: Chinese finger counting. Image taken from https://www.researchgate.net/figure/Chinese-finger-counting-system_fig1_301695915

The gestures in Fig.34 were recorded for both the right and left hand, hence the 6 overall gestures. The gestures each lasted approximately 10 seconds long which resulted in 377 observations being recorded (this training dataset can be found in the '_trial.csv' file whilst the testing data can be found in the '_testTrial.csv' file)**.** However, problems were quickly discovered by using this approach. Due to the dataset being too small and the train test split method being used on this data, the majority of the classifiers struggled to accurately predict classes. This was simply due to the lack of data in both the training and testing datasets which resulted in extreme overfitting. To remove this problem, k-fold cross-validation was considered with 10 folds. To ensure that the same results could be replicated throughout this study, a random state value of 123 was provided where possible.

As this dataset was not shuffled either, this meant that the classifiers could have potentially been recognizing the class of the previous adjacent observation and predict the current observation to be of the same class. This would then be repeated until the predicted class is incorrectly guessed at which point the classifier will then begin to predict every observation as the new class. To remove the possibility of this, each of the observations within the dataset were shuffled the moment they were stored in a dataframe.

Naively thinking that collecting more training data would suddenly improve the accuracy of some of the models, resulted in slightly improved model accuracies although they were still significantly lower than what I had anticipated. This new extensive dataset was the dataset which was utilized throughout the rest of the study and contained 14 classes, similar to the dynamic hand gesture 14/28 dataset used in [25]. Seven gestures were recorded for both the right and left hands for a minute long each and were repeated three times. To extend the dataset further, four other subjects were also invited to take part within the study, leading to the collection of approximately

39500 observations. As this dataset is to be used solely for training purposes, it was essential to collect some additional gesture data to be used in a testing dataset, which the trained model can make predictions with. By doing this, it illustrates how robust the model actually is when provided with new unseen data. The training dataset can be found in the 'TrainingData.csv' file, whilst the testing dataset can be found in the 'TestingData.csv' file.

The generic gestures included in the extensive training dataset can be seen in Fig.35.

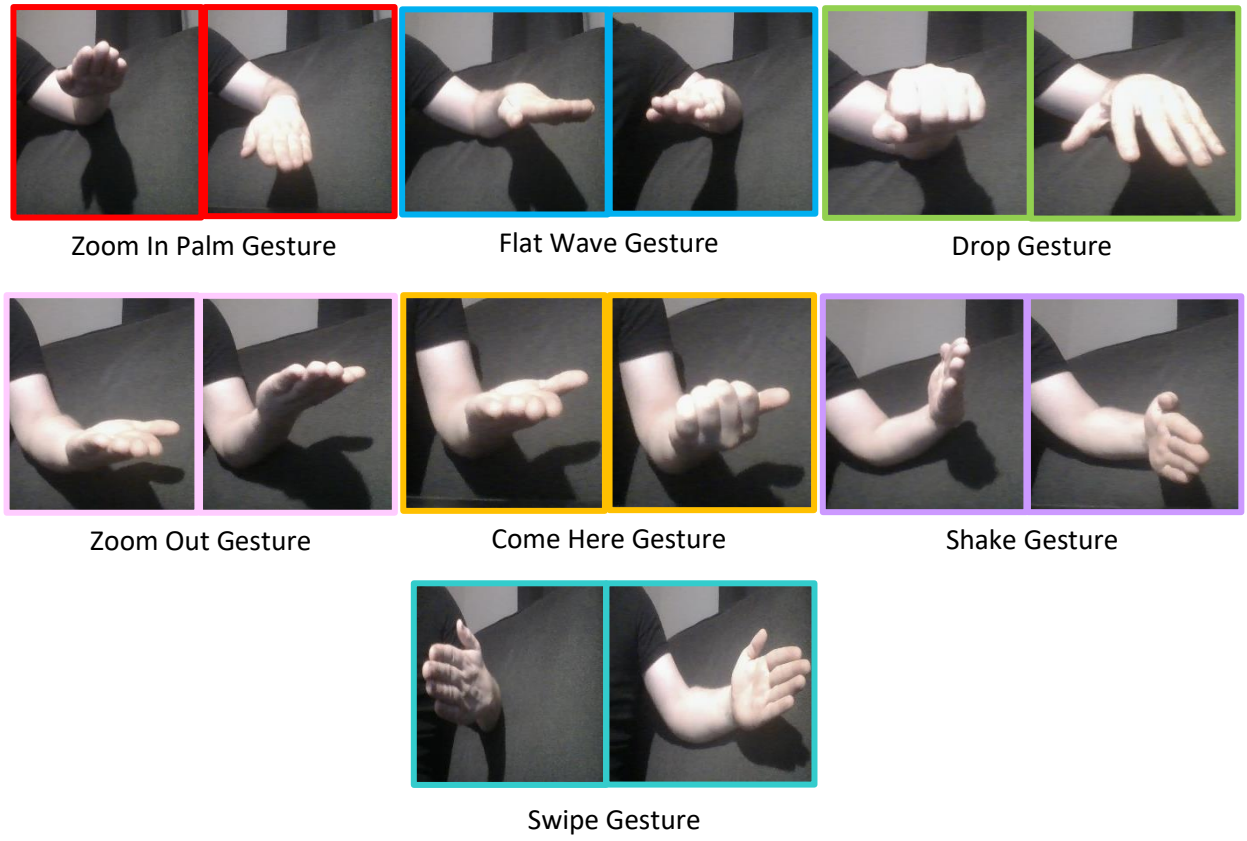


Fig.35: A generic version of the gestures which are included within the extensive training dataset.

The reason behind selecting these specific gestures to be included within the dataset was due to the similarities between them all with an example of this being the Zoom In Palm gesture and the flat wave gesture (the coordinates and normals of the hand will overlap across observations).

Implementation with python

Classifiers were firstly trained with no hyperparameters using the raw training dataset and were tested using the raw testing dataset. This was done to obtain benchmark accuracies for each of them and give some insight into the possibility of discarding poor classifiers early on. The classifiers which were trained and tested in the initial stages of development were:

- Decision Tree
- Multi-layer Perceptron (MLP)
- Random Forest
- K-nearest Neighbour (KNN)
- Support Vector Machine (SVM)
- Naïve Bayes
- Logistic Regression

The accuracies of each of the benchmark classifiers are presented in Table 15 and can be seen in the 'Pre-processing Approaches' notebook.

Classifier	Training Mean Accuracy	Training Standard Deviation	Unseen Data Accuracy
Decision Tree	98%	0.002	40%
Multi-Layer Perceptron	99%	0.003	39%
Random Forest	100%	0.001	68%
KNN	99%	0.002	62%
SVM	96%	0.004	51%
Naïve Bayes	70%	0.011	53%
Logistic Regression	90%	0.009	48%

Table 15: Benchmark Accuracies for each of the Classifiers.

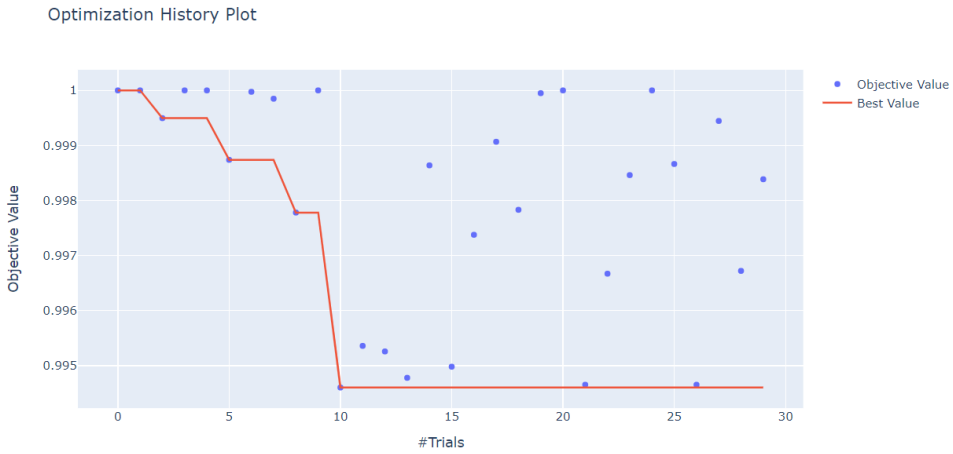
With the use of Table 15, it is evident that overfitting is occurring with all of the classifiers as their accuracies on the training data are considerably higher than their accuracies on the testing data. This essentially means that they are simply not generalizing the training data enough to accurately classify unseen observations.

All classifiers other than the KNN and Random Forest, were discarded due to their poor performances on unseen data. By doing this, it also allowed a greater focus on improving a smaller number of classifiers rather than attempting to improve them all.

Attempts were then made to tune the hyperparameters of these two models using both gridSearchCV and RandomizedSearchCV, with the intention of improving their accuracies on unseen data. However, applying these tuning methods to the random forest, was computationally expensive (approximately 16 hours for the gridSearchCV approach to tune the random forest model) to tune. The model accuracies on training data did fractionally increase, although unfortunately their performances on unseen data diminished, supporting the concept of overfitting. As the tuning of hyperparameters intends to improve the accuracy of classifiers on training datasets, and not unseen testing datasets, this is the cause for the increased level of overfitting.

It was at this stage that other tuning methods were considered and researched, leading to the consideration of Optuna, as it was significantly more efficient in the way it tuned hyperparameters. The use of Optuna's '*best_trial.params*' values for model hyperparameters were used, as well as the graphs which displayed the trialed hyperparameter values. This allowed for a manual analysis of values to determine the performance of other parameter values used in the various trials. The optimization process can be found in the 'Pre-processing Approaches' notebook.

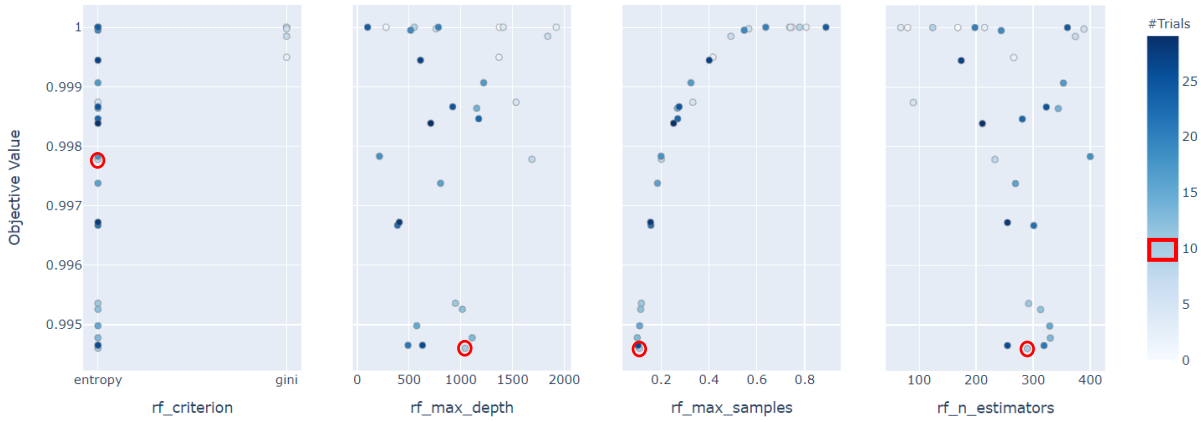
Graph 1 and Fig.36 illustrate the number of trials processed to find the optimal set of hyperparameters, as well as the actual values of these hyperparameters used within the optimal trial respectively. Graph 2, on the other hand, presents the hyperparameter values which were selected for each of the trials when tuning the random forest.



Graph 1: Optimization History Plot

```
[I 2021-06-21 21:18:18,200] Trial 14 finished with value: 0.9986382892878757 and parameters: {'rf_max_depth': 1154, 'rf_criteri
on': 'entropy', 'rf_n_estimators': 344, 'rf_max_samples': 0.2677695253558929}. Best is trial 10 with value: 0.9946035908815816.
[I 2021-06-21 21:18:54,592] Trial 15 finished with value: 0.9949818438571717 and parameters: {'rf_max_depth': 575, 'rf_criterio
n': 'entropy', 'rf_n_estimators': 329, 'rf_max_samples': 0.110123108131033}. Best is trial 10 with value: 0.9946035908815816.
[I 2021-06-21 21:19:44,590] Trial 16 finished with value: 0.9973774460359088 and parameters: {'rf_max_depth': 805, 'rf_criterio
n': 'entropy', 'rf_n_estimators': 269, 'rf_max_samples': 0.18484537469743667}. Best is trial 10 with value: 0.9946035908815816.
[I 2021-06-21 21:21:39,697] Trial 17 finished with value: 0.9990669759935444 and parameters: {'rf_max_depth': 1221, 'rf_criteri
on': 'entropy', 'rf_n_estimators': 353, 'rf_max_samples': 0.32447684511184527}. Best is trial 10 with value: 0.994603590881581
6.
[I 2021-06-21 21:22:58,265] Trial 18 finished with value: 0.9978313496066169 and parameters: {'rf_max_depth': 215, 'rf_criterio
n': 'entropy', 'rf_n_estimators': 400, 'rf_max_samples': 0.19964824680948592}. Best is trial 10 with value: 0.9946035908815816.
[I 2021-06-21 21:25:05,434] Trial 19 finished with value: 0.9999495662699214 and parameters: {'rf_max_depth': 516, 'rf_criterio
n': 'entropy', 'rf_n_estimators': 244, 'rf_max_samples': 0.546959080055815}. Best is trial 10 with value: 0.9946035908815816.
[I 2021-06-21 21:26:50,727] Trial 20 finished with value: 1.0 and parameters: {'rf_max_depth': 784, 'rf_criterion': 'entropy',
'rf_n_estimators': 198, 'rf_max_samples': 0.6370566896852949}. Best is trial 10 with value: 0.9946035908815816.
[I 2021-06-21 21:27:22,411] Trial 21 finished with value: 0.9946540246116603 and parameters: {'rf_max_depth': 491, 'rf_criterio
n': 'entropy', 'rf_n_estimators': 319, 'rf_max_samples': 0.10118942406131032}. Best is trial 10 with value: 0.9946035908815816.
[I 2021-06-21 21:28:07,422] Trial 22 finished with value: 0.9966713738148073 and parameters: {'rf_max_depth': 389, 'rf_criterio
n': 'entropy', 'rf_n_estimators': 301, 'rf_max_samples': 0.15719895934619707}. Best is trial 10 with value: 0.9946035908815816.
[I 2021-06-21 21:29:18,995] Trial 23 finished with value: 0.9984617712326004 and parameters: {'rf_max_depth': 1173, 'rf_criteri
on': 'entropy', 'rf_n_estimators': 281, 'rf_max_samples': 0.2691758429777864}. Best is trial 10 with value: 0.9946035908815816.
[I 2021-06-21 21:33:21,772] Trial 24 finished with value: 1.0 and parameters: {'rf_max_depth': 101, 'rf_criterion': 'entropy',
'rf_n_estimators': 360, 'rf_max_samples': 0.888518925605582 8}. Best is trial 10 with value: 0.9946035908815816.
[I 2021-06-21 21:34:46,421] Trial 25 finished with value: 0.9986635061529151 and parameters: {'rf_max_depth': 922, 'rf_criterio
n': 'entropy', 'rf_n_estimators': 323, 'rf_max_samples': 0.2752385855558396}. Best is trial 10 with value: 0.9946035908815816.
[I 2021-06-21 21:35:13,648] Trial 26 finished with value: 0.9946540246116603 and parameters: {'rf_max_depth': 630, 'rf_criterio
n': 'entropy', 'rf_n_estimators': 255, 'rf_max_samples': 0.10378151352560802}. Best is trial 10 with value: 0.9946035908815816.
[I 2021-06-21 21:36:19,272] Trial 27 finished with value: 0.9994452289691346 and parameters: {'rf_max_depth': 611, 'rf_criterio
n': 'entropy', 'rf_n_estimators': 174, 'rf_max_samples': 0.40101169705957634}. Best is trial 10 with value: 0.9946035908815816.
[I 2021-06-21 21:36:58,980] Trial 28 finished with value: 0.9967218075448886 and parameters: {'rf_max_depth': 407, 'rf_criterio
n': 'entropy', 'rf_n_estimators': 255, 'rf_max_samples': 0.15561543650815 4}. Best is trial 10 with value: 0.9946035908815816.
[I 2021-06-21 21:37:55,411] Trial 29 finished with value: 0.9983861206374823 and parameters: {'rf_max_depth': 710, 'rf_criterio
n': 'entropy', 'rf_n_estimators': 211, 'rf_max_samples': 0.25216110757355535}. Best is trial 10 with value: 0.9946035908815816.
```

Fig.36: Best Parameters outputted by Optuna



Graph 2: Trialled Parameter values

The '*best_trial.params*' values were evidently found in trial 10, and the hyperparameter values which were used for this trial were therefore included in the random forest classifier to determine whether this had any improvement of the model on unseen data. Fig.37 illustrates this.

Updating the random forest classifier with the tuned hyperparameters from Optuna

In [322]:
```
#'rf_max_depth': 1042,
#'rf_criterion': 'entropy',
#'rf_n_estimators': 290,
#'rf_max_samples': 0.10933268394129136

rf = RandomForestClassifier(n_estimators=290, max_samples=0.10933268394129136, max_depth=1042, criterion='entropy', random_state=
rfScore = []
```

Outputs the Results of the tuned Classifier on the Training Data:

In [324]:
```
print("Random Forest:")
print(rfScore)
standar = np.std(rfScore)
print("\nStandard Deviation = %0.4f" % (standar))
meanVal = np.mean(rfScore)
print("Mean Value is = %0.4f" % (meanVal))
```

```
Random Forest:
[0.9909228441754917, 0.9926878466969239, 0.9914271306101866, 0.9919314170448815, 0.9881492687846697, 0.992183560262229, 0.99092
05548549811, 0.9881462799495586, 0.9939470365699874, 0.9894073139974779]

Standard Deviation = 0.0018
Mean Value is = 0.9910
```

Outputs the Accuracy of the tuned Classifier being applied to Testing Data:

In [60]:
```
df_unseen = pd.read_csv("../Study's Main Raw Datasets/TestingData.csv", names=col)
df_unseen = pd.DataFrame(df_unseen.sample(frac=1, random_state=seed).reset_index(drop=True))

X_unseen = df_unseen.iloc[:, :-1]
X_unseen = min_max_scaler.fit_transform(X_unseen)

Y_unseen = df_unseen.iloc[:, -1:]
Y_unseen = Y_unseen.values.ravel()

score_unseen = rf.score(X_unseen, Y_unseen)
print("The score of the random forest on unseen data:", np.around((100*score_unseen),decimals=2),"%")
```

```
The score of the random forest on unseen data: 64.16 %
```

Fig.37: The accuracy of the model after tuning with Optunas best parameters

Using Fig.37, it can be evaluated that the accuracy on unseen data after tuning the random forest classifier with the '*best_trial.params',* performed slightly worse than the benchmark classifier. However, the KNN classifier on the other hand did improve slightly to 65%. The results for both of these classifiers can be seen in Table 16.

Classifier	Hyperparameters after Tuning	Training Mean Accuracy	Training Standard Deviation	Unseen Data Accuracy
Random Forest	(n_estimators = 290) (max_samples = 0.10933268394129136) (max_depth = 1042) (criterion = 'entropy') (random_state = 123)	99%	0.002	64%
KNN	(n_neighbors = 11)	99%	0.002	65%

Table 16: Hyperparameters after Tuning using Optuna's best parameters

A conclusion was drawn here that classifiers which have a small number of very basic hyperparameters to tune, such as the KNN classifier, should continue to use the '*best_trial.params*' values for optimization. However, classifiers which are slightly more complexed such as the random forest, should perhaps use a different approach to tune hyperparameters.

At this point within the development, a new trial-and-error approach using optunas graphs was considered to help reduce the problem of overfitting. Using graph 2, the best parameter values which were returned by optuna can be seen in the red circles. Each of these points are of the same blue shade which the 10th trial in the red rectangle represents. With the use of this information, this allows the possibility to combine various hyperparameter values which were not used together within the optuna optimization trials.

Hyperparameter values which were used within trials 10 to 15, were then mixed with one another using a trial-and-error approach. This led to an improvement of model accuracies on unseen data. The reason behind selecting this specific range of trials was due to the mean objective value within this range being the lowest available mean across 5 consecutive trials (the lower the objective value, the better the classifier generalises the data, leading to a higher accuracy on unseen data). As the trials were all consecutive, this also made it easier to determine whether hyperparameters were used within any of these five trials as one specific shade of blue was no longer required but now the choice between 5 could be selected. The results of this approach were promising and can be seen in Fig.38. Note that an increase in the number of trials, '*n_trials*', would lead to more hyperparameter values being considered within optunas optimisation process. However, this would lead to the scale of blue being fitted to a larger quantity of trials and therefore be difficult to distinguish between them. It would also increase the computational cost, defeating the purpose of using Optuna. It should also be noted that a trial-and-error approach certainly does not offer the most optimal

solution to the problem, although it is a cost-efficient solution which has proved valid to produce higher accuracies on unseen data.

```
In [328]: rf = RandomForestClassifier(n_estimators=194, max_samples=0.160885, max_depth=638, criterion='gini', random_state=seed, n_jobs=-1
          rfScore = []
```

Outputs the Results of the tuned Classifier on the Training Data:

```
In [330]: print("Random Forest:")
          print(rfScore)
          standar = np.std(rfScore)
          print("\nStandard Deviation = %0.4f" % (standar))
          meanVal = np.mean(rfScore)
          print("Mean Value is = %0.4f" % (meanVal))

Random Forest:
[0.9931921331316188, 0.9936964195663137, 0.9919314170448815, 0.9926878466969239, 0.9899142713061019, 0.9936964195663137, 0.9919
293820933165, 0.9891551071878941, 0.9931904161412358, 0.9911727616645649]

Standard Deviation = 0.0015
Mean Value is = 0.9921
```

Outputs the Accuracy of the tuned Classifier being applied to Testing Data:

```
In [64]: df_unseen = pd.read_csv("../Study's Main Raw Datasets/TestingData.csv", names=col)
         df_unseen = pd.DataFrame(df_unseen.sample(frac=1, random_state=seed).reset_index(drop=True))

         X_unseen = df_unseen.iloc[:, :-1]
         X_unseen = min_max_scaler.fit_transform(X_unseen)

         Y_unseen = df_unseen.iloc[:, -1:]
         Y_unseen = Y_unseen.values.ravel()

         score_unseen = rf.score(X_unseen, Y_unseen)
         print("The score of the random forest on unseen data:", np.around((100*score_unseen),decimals=2),"%")

The score of the random forest on unseen data: 68.89 %
```

Fig.38: The new trial-and-error approach used with a random forest model on unseen data

An increase of 68% to 69% accuracy on unseen data with the random forest classifier is only a fractional increase but does illustrate that the trial-and-error approach is successful. However, the accuracy of the model after training was still considerably higher than what was being obtained when applying it to unseen data. Table 17 shows the results from tuning the random forest and KNN classifiers using the trial-and-error approach with optuna's graphs.

Classifier	Hyperparameters after Tuning	Training Mean Accuracy	Training Standard Deviation	Unseen Data Accuracy
Random Forest	(n_estimators = 194) (max_samples = 0.160885) (max_depth = 638) (criterion = 'gini') (random_state = 123)	99%	0.002	69%
KNN	(n_neighbors = 11)	99%	0.002	65%

Table 17: The hyperparameters used from the trial-and-error graph approach

The sampling method which was used for cross validation was then altered to determine whether this would have any impact upon overfitting. This was trialed as it is possible that with the original random approach used with k-fold cross validation, that the testing subset may consist of a large quantity of observations belonging to a certain class, and therefore only a minimal number of observations of the same class being present within the training subset, resulting in a poor accuracy on unseen data. Therefore stratified k-fold sampling was considered instead and through using this approach, it ensured that the same percentage of each of the classes were used in the training subsets and in the testing subsets. The number of splits was unchanged with the value of 10, and both of the classifiers which were fitted to the data contained the same hyperparameter values which were obtained using the previous trial and error approach with Optuna. Table 18 illustrates the two classifiers being fitted to the dataset using stratified k-fold cross validation, whilst the implementation can be found within the 'Random Forest Tested Approaches' notebook.

Classifier	Training Mean Accuracy	Training Standard Deviation	Unseen Data Accuracy
Random Forest	99%	0.002	68%
KNN	99%	0.002	65%

Table 18: Accuracies of both KNN and Random Forest classifiers after using stratified k-fold cross validation over k-fold cross validation

It can be concluded that the stratified k-fold approach did not have any positive impact upon the accuracies on unseen data using the the trial-and-error hyperparameters. Therefore, it was concluded that either of these methods are satisfactory to include for cross validation.

As there are 14 labels to distinguish between (7 gestures for the left hand and the same gestures for the right hand), the dataset was manipulated slightly by changing the labels to make them more generic such as changing 'LeftSwipe' and 'RightSwipe' to just 'Swipe'. The reason behind this was perhaps it could have made it easier for the models to distinguish labels whilst maintaining enough classes to control the main aspects of a robotic arm. Unfortunately, the accuracy of the

models on unseen data did not improve after applying this strategy, and instead produced accuracies below the original benchmarks. Therefore, this approach was not considered any further. The results for this can be seen in Table 19 with the random forest and KNN classifiers which had been trained using the 'TrainingMergedGestures.csv' file and tested using the 'TestingMergedGestures.csv' file. The implementation which produced these accuracies can be found within the 'Random Forest Tested Approaches' notebook.

Classifier	Training Mean Accuracy	Training Standard Deviation	Unseen Data Accuracy
Random Forest	100%	0.001	66%
KNN	99%	0.001	60%

Table 19: Accuracies after fitting the KNN and Random Forest classifiers to the merged dataset

However, as overfitting was still occurring, it was evident that feature selection and feature extraction methods should be considered. Principal component analysis was the next approach implemented, with the purpose of improving the generalization of models which will reduce overfitting, and will also speed up the training and tuning of models considerably. Both 95% (12 eigenvectors) and 99% (23 eigenvectors) variance explained by components were trialed with the models, to discover which of the two was most appropriate for models to generalize the data. The random forest which was fitted to the new subspace contained no hyperparamaters other than the '*random_state*', as this would give insight into if a better benchmark classifier could be obtained. The KNN classifier, on the other hand, had the same hyperparamaters which were previously discovered during the tuning stage (n_neighbors = 11). The results for both of the classifiers being fitted to the new subspace can be seen in Table 20 and Table 21.

Classifier	Training Mean Accuracy	Training Standard Deviation	Unseen Data Accuracy
Random Forest	96%	0.003	34%
KNN	95%	0.004	67%

Table 20: Accuracies of both KNN and Random Forest classifiers after using PCA with 95% variance applied to the dataset

Classifier	Training Mean Accuracy	Training Standard Deviation	Unseen Data Accuracy
Random Forest	99%	0.002	38%
KNN	98%	0.002	61%

Table 21: Accuracies of both KNN and Random Forest classifiers after using PCA with 99% variance applied to the dataset

Using Tables 20 and 21, it can be seen that PCA with 95% explained variance had a slight improvement on unseen data for the KNN classifier, although the Random Forest performed considerably poorly on unseen data. Therefore feature extraction approaches was discontinued. The implementation for PCA can be found within the 'Random Forest Tested Approaches' notebook.

However, feature selection processes were then trialled to determine if all 428 features were appropriate to retain. Firstly, all features were checked whether they contained any null values within them to determine whether they would need values imputing or whether the features could simply just be eliminated. There were no null values within the dataset and therefore imputation was not required. Therefore, the next feature selection method consisted of calculating the correlation between pairs of features. The use of this feature selection approach was to remove all of the highly correlated features by using the '*corr()*' method with a dataframe in relation to the target column. This list was then sorted into a descending order so that the top of the list relates to the features which are highly correlated to other features, whilst the features at the bottom of the list have little relation to other features and should be retained. Through the use of trial and improvement, it was discovered that by eliminating all features with a correlation value above 0.2 led to 300 features being removed from the dataset with 128 still remaining. As this was a significant number of features being removed from the training dataset, some of the classifiers originally trialled were fitted to the new dataset along with some boosting classifiers including XGBoost and LGBoost. The results can be seen in Table 22 and the implementation of feature selection processes can be seen in the 'Pre-processing Approaches' notebook.

Classifier	Training Mean Accuracy	Training Standard Deviation	Unseen Data Accuracy
Random Forest	98%	0.002	83%
KNN	96%	0.002	82%
MLP	95%	0.004	51%
Naïve Bayes	85%	0.007	24%
Logistic Regression	80%	0.006	53%

Table 22: The accuracies of the five main classifiers fitted to the dimensionality reduced dataset

This was significant progress within the study, as the previous top two models (Random Forest and KNN) both increased by a minimum of 15%. However, the other classifiers did continue to underperform and were therefore discontinued for a second time. The confusion matrix for both the Random forest and the KNN classifier can be seen in Fig.39 which illustrates the predictions of the classifiers. This therefore allows an analysis of how many instances have been missclassified and what they were incorrectly classified as. Table 23 should also be used here to determine what each of the encoded gestures in the matrix correspond to.

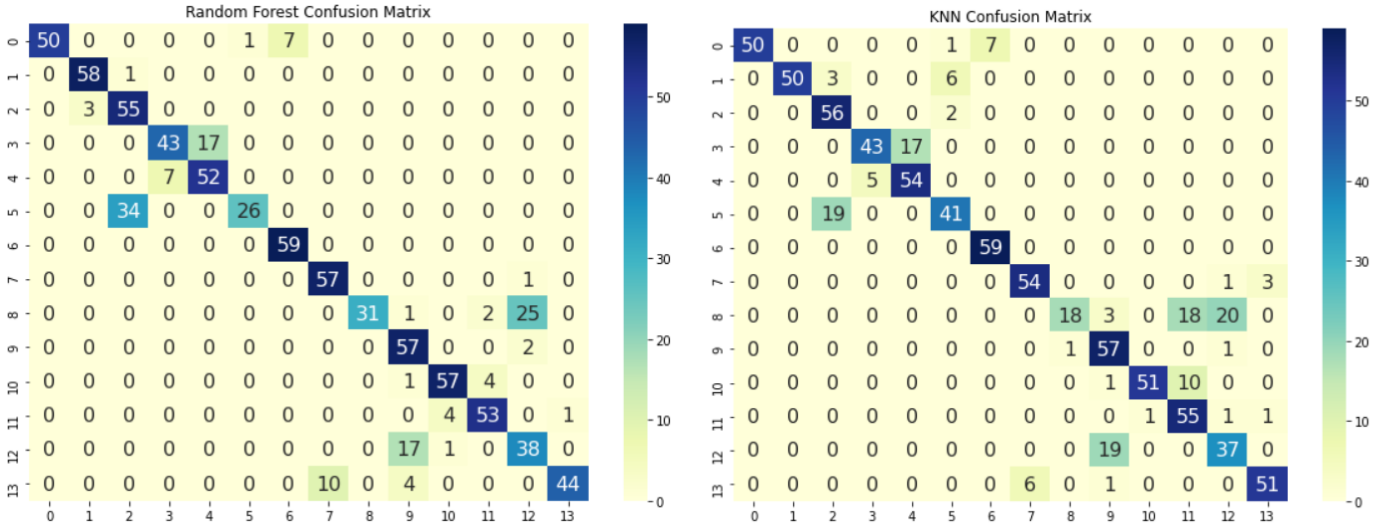


Fig.39: Confusion matrices for the top two classifiers (Random Forest is on the left and KNN is on the right)

Value in the confusion matrix	Gesture which is represented
0	LeftComeHere
1	LeftDrop
2	LeftFlatWave
3	LeftShake
4	LeftSwipe
5	LeftZoomInPalm
6	LeftZoomOut
7	RightComeHere
8	RightDrop
9	RightFlatWave
10	RightShake
11	RightSwipe
12	RightZoomInPalm
13	RightZoomOut

Table 23: The inverse transformations of what each of the numbers in the confusion matrix axis represent

Therefore, using these confusion matrices, it can be concluded that both classifiers struggled to distuniguish the difference between the 'ZoomInPalm' and 'FlatWave' gestures for both the left and the right hands. Other misclassifications which occurred across both classifiers included predictions of the 'LeftSwipe' gesture when the 'LeftShake' gesture was expected and predicting the 'RightZoomInPalm' gesture when the 'RightDrop' gesture was expected. I anticipate that the reasoning behind these missclassifications is simply due to the gestures being very similar to one another and as both features such as palm coordinates and hand direction are still present in the dataset, it is more than likely that the values for these features will be similar at some stage when performing the gestures. Therefore, although the accuracies of both of these classifiers are considerably higher than what they previously were, a hidden markov model approach could be worthy to consider as this would help determine gestures based on previous gestures. However, this would result in not being able to shuffle the data before training and therefore create an unnecessary bias where classifiers predict labels based on what the previous class was.

As this previous feature selection method was a success, another feature selection method, recursive feature elimination with cross validation (RFECV), was considered in hope that it may lead to a more accurate model without the need of trial and improvement, essentially offering more relibility. However this approach was computationally expensive due to the high number of recursions and unfortunatly this had to be interrupted early. The code for this to be run can be seen in Fig.40 as well as in the 'Pre-processing Approaches' notebook.

Recursive feature elimination (RFE)

In [ ]:
```
from sklearn.model_selection import cross_val_score
from sklearn.feature_selection import RFECV
from sklearn.pipeline import Pipeline

rfe = RFECV(estimator=rf)
pipeline = Pipeline(steps=[('rfe',rfe),('rfModel',rf)])

n_scores = cross_val_score(pipeline, X, y, scoring='accuracy', cv=kf, n_jobs=-1, error_score='raise')

print('Accuracy: %.3f (%.3f)' % (np.mean(n_scores), np.std(n_scores)))
```

This approach was computationally expensive and therefore it was interrupted before completing the process. It caused extreme overheating and took a considerable amount of time.

Fig.40: Recursive Feature Elimination implementation

Another approach which was considered was using auto machine learning. Both the autoML in microsoft azure, and a python library known as TPOT were considered. TPOT finds an optimal model which provides the best accuracy and only requires the dataset in order to do so. The code to obtain the optimal model can be seen in Fig.41 and can also be found in the 'Random Forest Tested Approaches' notebook.

TPOT AutoML

```
In [5]: from sklearn.model_selection import RepeatedStratifiedKFold
        from tpot import TPOTClassifier

        y = LabelEncoder().fit_transform(y)

        rskf = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=seed)
        tpModel = TPOTClassifier(generations=5, population_size=50, cv=rskf, scoring='accuracy', random_state=seed, n_jobs=-1)

        tpModel.fit(X, y)

        Generation 1 - Current best internal CV score: 0.8870620344385874

        Generation 2 - Current best internal CV score: 0.8870620344385874

        Generation 3 - Current best internal CV score: 0.8870620344385874

        Generation 4 - Current best internal CV score: 0.8870620344385874

        Generation 5 - Current best internal CV score: 0.8870620344385874

        Best pipeline: GaussianNB(ZeroCount(input_matrix))

Out[5]: TPOTClassifier(cv=RepeatedStratifiedKFold(n_repeats=3, n_splits=10, random_state=123),
                       generations=5, n_jobs=-1, population_size=50, random_state=123,
                       scoring='accuracy', verbosity=2)

In [6]: tpModel.export('TPOT_best_model_Gestures.py')
```

Fig.41: TPOT automated machine learning code and best model output

As this model was exported to a python script within the last cell of Fig.41, the python script 'TPOT_best_model_Gestures.py' was run and this produced the accuracies which can be seen in Table 24.

Classifier	Training Mean Accuracy	Training Standard Deviation	Unseen Data Accuracy
make_pipeline(ZeroCount(), GaussianNB())	89%	0.006	21%

Table 24: The accuracies obtained from the TPOT best model

Although it was fairly computationally expensive (approximately 2 hours) due to the sheer size of potential combinations in order to reach an optimal model, the time that it would have taken to manually train and test each classifier, tune their hyperparameters and appropriately pre-process the training dataset would have been considerably more expensive. However, the results which were produced in the Table 24 illustrate that the 'optimal' model actually performs extremely poorly on unseen data and therefore this model was inappropriate to consider.

Microsoft Azure works in a similar way as it combines both machine learning algorithms and feature selection techniques with one another and tunes the model's hyperparameters in order to improve their accuracies where appropriate. However, this also produced severely poor results which can be seen in Fig.42 where the blue rectangle contains the various models trialled, and the red rectangle illustrates the accuracies of the corresponding models. The green rectangle includes the time duration for each of the models to be processed too.

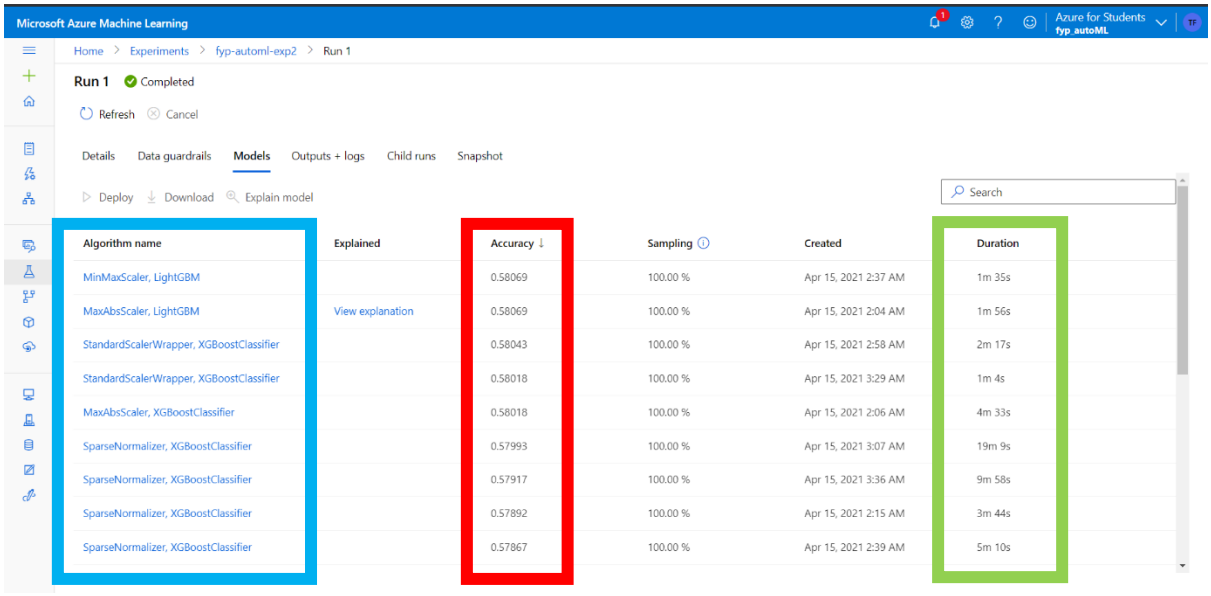


Fig.42: The outputted models, their duration periods to be trained and their accuracies in Microsoft Azure AutoML

It is clear that the 6th top scoring model, 'SparseNormalizer, XGBoostClassifier', was computationally expensive compared to the other models with it taking over 19 minutes to be processed. It can also be concluded from these results that the 'MinMaxScaler' scaling technique had a positive impact upon producing the best results, and therefore illustrates that this should be maintained within the final python model. 7 out of the top 9 performing models contained 'XGBoostClassifier', showing that perhaps this is a strong boosting method, whilst the top two models incorporated 'LightGBM' which is another boosting algorithm. As 'LightGBM' is approximately 7 times faster than the 'XGBoostClassifier' [44], and has been known to produce higher accuracies on large datasets (it is prone to overfitting with small datasets), this boosting classifier had to be investigated further. However, as the highest scoring model within Fig.42 was still below the benchmark value of 82% and 83%, all of these models were discarded.

Another experiment was then setup to run within azure again, but for a slightly longer period of time as the previous experiment was restricted in case the process was computationally expensive. The parameters which were passed to the new experiment can be seen in Fig.43.

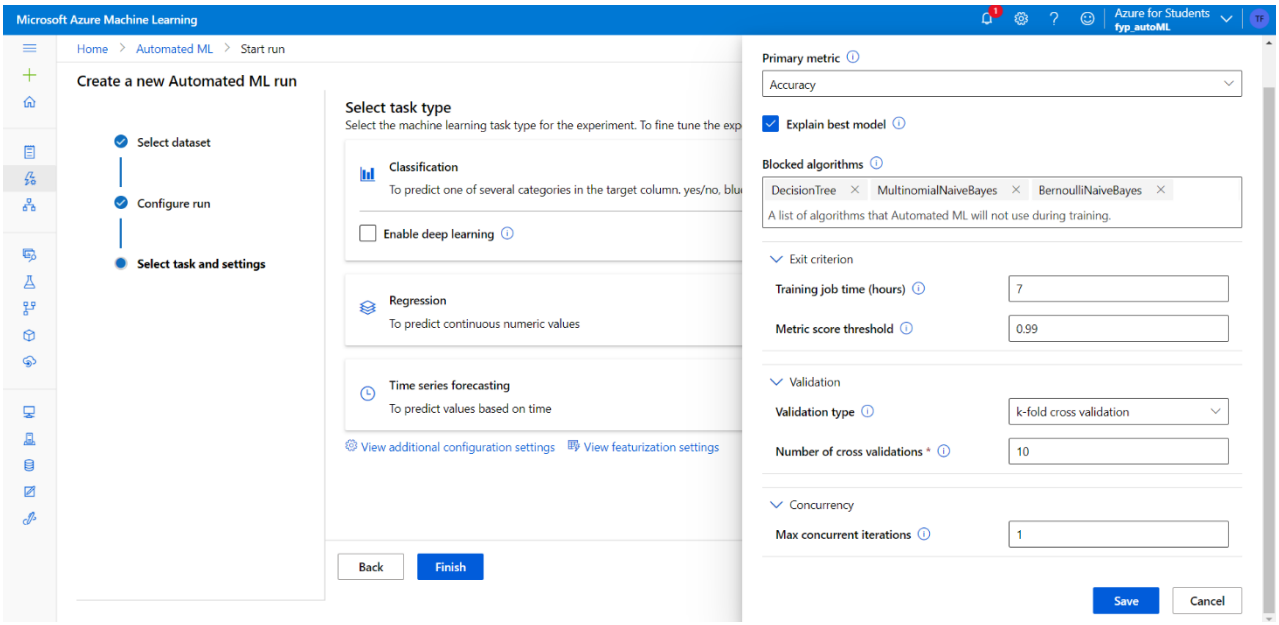


Fig.43: The initialisation process of Microsoft Azure AutoML

Exit criteria was declared to ensure that if 99% accuracy was not met within the 7-hour timeframe, that the experiment would end. This specified period of time is at least double the previous experiments timeframe although the experiment was not expected to run for that long. By removing some of the poor performing classifiers such as Decision Tree and Naïve Bayes, this allowed more time to be invested into the stronger models. However, the experiment was over fairly fast and produced a similar set of results to the first experiment, leading to the discontinue of AutoML within the study. Although, as automatic featurization was enabled, this did fortunately provide some insight into the dataset regarding whether any preprocessing stages were required, which luckily was not the case. This information can be seen in the Fig.44.

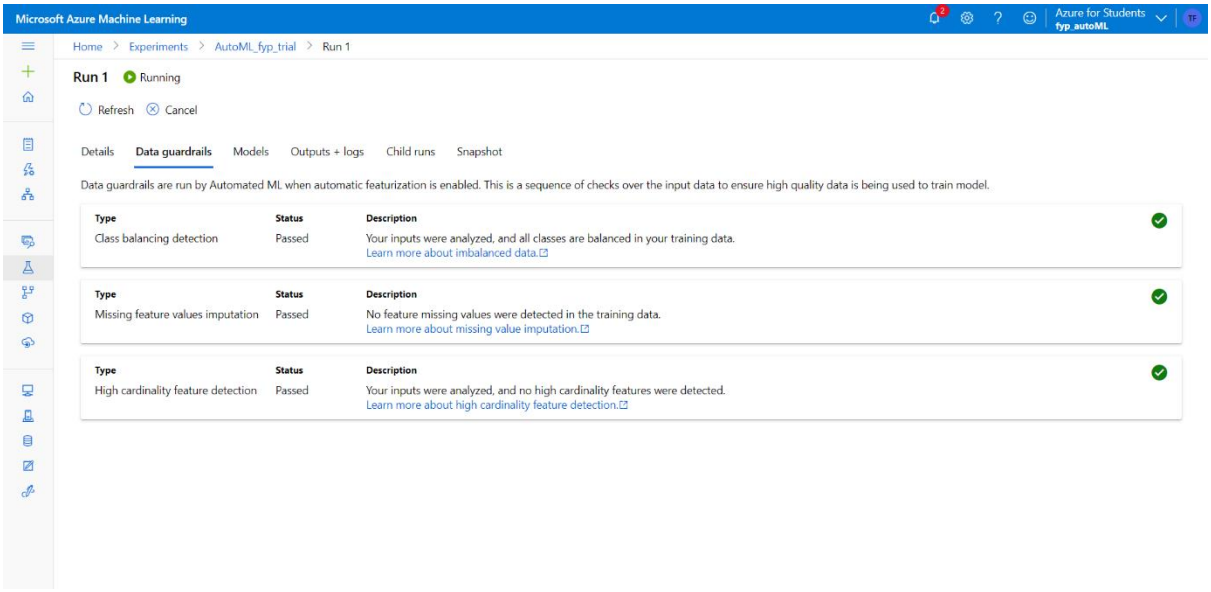


Fig.44: Microsoft Azure analyses whether classes are balanced, if there is high cardinality and if data is missing

Due to the frequency of both 'XGBoost' and 'LGBoost' appearing in the top models outputted by Azure's autoML, both of these classifiers were implemented within python. The results of these can be seen in Table 25 or within the 'Pre-processing Approaches' notebook.

Classifier	Training Mean Accuracy	Training Standard Deviation	Unseen Data Accuracy
XGBoost	99%	0.001	62%
XGBRF	92%	0.005	64%
LGBoost	99%	0.001	65%

Table 25: The accuracies of popular boosting classifiers

The XGBoost classifier was too computationally expensive without the '*tree_method*' hyperparameter set to '*gpu_hist*' and still took approximately 15 minutes to train the model with this hyperparameter present. Although this hyperparameter value is utilized to speed up the process considerably, the classifier is still slower than the original random forest model and the LGBoost model. Using Table 25, it is clear that all three of these boosting classifiers did not come close to improving the updated benchmark models of 82% or 83% and therefore were discarded as possible models.

Another ensemble method classifier which was trialed, included the voting Classifier as it originally performed well within weka. As it is not possible for the classifier to perform worse than the individual classifier components, but does offer the potential to improve an overall accuracy, this model had to be implemented. The results for a voting classifier with both hard voting and soft voting can be seen in Table 26. The 'Pre-processing Approaches' notebook contains the code for this.

Classifier	Training Mean Accuracy	Training Standard Deviation	Unseen Data Accuracy
Voting Classifier - Hard Voting	97%	0.002	84%
Voting Classifier - Soft Voting	98%	0.003	83%

Table 26: The accuracies of voting classifiers with both hard and soft voting strategies

As expected, there was a fractional improvement in accuracy by 1% with hard voting, whilst an unchanged accuracy obtained using soft voting occurred, proving the theory that the accuracy of voting classifiers cannot be less than the classifier components. However, as the time taken to train this model was considerably longer than that of the random forest, pursuing the voting classifier should be considered with caution.

Once again, the hyperparameters for the top 2 classifiers (KNN and the Random Forest) were tuned with the intention to improve the accuracies of these further. Due to the feature selection process cutting out 300 of the original features, 128 features still remained and was therefore still computationally expensive to tune hyperparameters with either the 'GridSearchCV' or

'RandomisedSearchCV'. Optuna was therefore used once more due to its efficiency, and the values which were returned as the 'best parameters' were compared with both the KNN and Random Forest classifier. The results for these can be seen in Table 27 and the 'Pre-processing Approaches' notebook.

Classifier	Training Mean Accuracy	Training Standard Deviation	Unseen Data Accuracy
Random Forest	97%	0.003	86%
KNN	96%	0.002	82%

Table 27: The accuracies of the top two classifiers after hyperparameter tuning again

This time, it appeared that the hyperparameter tuning process did in fact improve the model accuracies using the '*best_trial.params*' values. This essentially illustrates how the model now generalizes data better than it did previously. However, the trial-and-error approach using optuna's graphs was still considered to ensure that a better accuracy could not be obtained. Fig.45 illustrates the variability between the values returned by optuna, and the values used from the trial-and-error approach, however they both produced the same accuracy of 86% on unseen data. This therefore suggests that the trial-and-error approach is successful and that the model may not improve much more without incorporating the individual tuned classifier into ensembles.

```
# The 'best parameters' provided by the optuna method
# 'rf_max_depth': 1246
# 'rf_criterion': 'entropy'
# 'rf_n_estimators': 193
# 'rf_max_samples': 0.10242244670584226

# OR

# The values obtained through my theory including trial and error produced the same accuracy of 86%
# 'rf_max_depth': 1058,
# 'rf_criterion': 'entropy',
# 'rf_n_estimators': 377,
# 'rf_max_samples': 0.1533258
```

Fig.45: The difference of hyperparameters selected by the framework itself and the trial-and-error graph approach

Fig.44 suggests that all classes were balanced, although as this had not been considered before this stage, this was analyzed further. In an attempt to prevent collection of imbalanced data when the raw data was collected, a timer was set up on a mobile device to ensure that 60 seconds was correctly counted. However, as the human hand data is collected every 0.2 seconds, the likelihood of a few extra observations being recorded for the gestures is inevitable. I naively assumed this using a stopwatch would prevent the imbalance of data and not have a noticeable impact on the model predictions as the imbalance would not be severe. However, using Fig.46, it was evident that an imbalance of data was present with approximately 800 extra samples within the majority class ('RightZoomInPalm') than there were within the minority class ('LeftShake').

Checking the Imbalance of the original data

```
labarr = []
labels = ['RightSwipe', 'LeftZoomOut', 'RightZoomOut', 'RightZoomInPalm',
          'RightShake','LeftComeHere', 'RightFlatWave', 'LeftZoomInPalm',
          'LeftShake', 'RightDrop', 'RightComeHere', 'LeftDrop', 'LeftSwipe',
          'LeftFlatWave']

for l in labels:
    labarr.append(Counter(y)[l])

zipLbl = zip(labarr, labels)
zipLbl = dict(zipLbl)
zipLbl = sorted(zipLbl.items())
print(zipLbl[0], " compared to: ", zipLbl[-1])
```

```
(2566, 'LeftShake')  compared to:  (3316, 'RightZoomInPalm')
```

There is clearly an imbalance with the minority class and majority class within the training dataset and therefore requires resampling

Fig.46: The imbalance of classes

This therefore required resampling to take place, where oversampling was applied to the dataset as a pre-processing step with the hope that this would perhaps lead to an improved accuracy on unseen data, as classifiers will not be learning more of one set than another. A few oversampling methods were considered including random oversampling, ADASYN, and SMOTE. Oversampling was preferred over undersampling as it does not decrease the number of observations but instead increases them. However, the results which were obtained suggest why the AutoML tool in Azure perhaps stated that the classes may be considered 'balanced', as the accuracy of both the KNN and random forest classifiers were only fractionally impacted (KNN increased to 84% whilst Random Forest decreased to 85%). This can be seen in the 'Pre-processing Approaches' notebook.

Ideally if there was enough time, undersampling should also have been considered as this would prevent repetition of any observations and decrease the computational cost but may also decrease the accuracy on unseen data as there would be less training data available to train classifiers. Combining both oversampling and undersampling would also have been interesting to consider although I believe both of these may not have a severe improvement of models if any at all.

Penultimately, in an attempt to improve the random forest classifier further, an adaboost classifier which used the tuned random forest classifier as its estimator was implemented. This was extremely computationally expensive and produced poorer results than what the individual random forest classifier had previously outputted. The results which were obtained along with the corresponding confusion matrix can be seen within Table 28 and Fig.47. This can also be viewed in the 'Pre-processing Approaches' notebook.

Classifier	Training Mean Accuracy	Training Standard Deviation	Unseen Data Accuracy
Random Forest	97%	0.003	86%
Adaboost with base_estimator = Random Forest	96%	0.003	82%

Table 28: The accuracies of applying Adaboost with the base estimator of a Random Forest

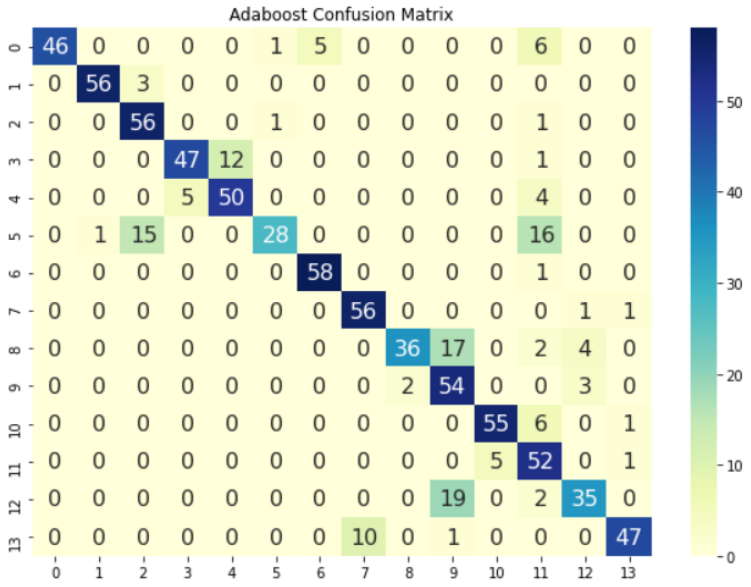


Fig.47: A confusion matrix of the Adaboost results

Finally, due to the success of the voting classifier which was previously used within the study, this was considered once again using the top four models up to this stage. The four classifiers included the KNN classifier (an individual accuracy of 82%), the two variations of random forest (one using the trial-and-error theory values and the other using the best parameter values returned, both of which had 86% accuracies) as well as the previous hard voting classifier which obtained 84%. This final ensemble was successful in that it improved the accuracy of the model to 87% on unseen data which was the best model obtained throughout this study. The accuracies and confusion matrix for this model can be seen in Table 29 and Fig.48 respectively.

Classifier	Training Mean Accuracy	Training Standard Deviation	Unseen Data Accuracy
Random Forest	97%	0.002	87%

Table 29: The accuracy of the voting classifier which uses classifiers including two random forests, a knn and another voting classifier

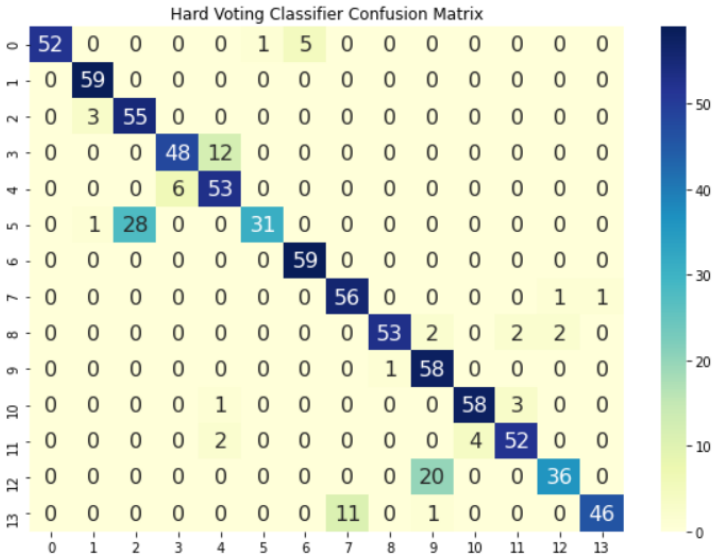


Fig.48: The confusion matrix of the best performing voting classifier

This finalized model can be found in the 'Final Model' notebook and can be loaded from the 'Final_Model.sav' file by using pickle which allows for the serialization and de-serialization of python objects.

Evaluation and Reflection

Evaluation of the study

It can be concluded that the best model within this study was the voting classifier consisting of two random forest classifiers which both had an accuracy of 86%, a KNN classifier with an accuracy of 82%, and another voting classifier (consisting of both a KNN and a random forest classifier) which had 84% accuracy. This model was fitted to the training dataset that had been manipulated using feature selection to remove highly correlated features, before they were scaled. The accuracy of this model (87%) on unseen data was extremely impressive considering each of the gestures were very similar to one another and outperformed similar studies from the literature review including [25] which was a very similar study in that they both consisted of 14 dynamic hand gestures which incorporated skeletal data. However, considering fusion of both skeletal data and a CNN was used within Lai and Yanushkevich's study, 82% accuracy was achieved using skeletal data alone, whilst 85% accuracy was achieved after fusion, which is still approximately 2% less than the best model within this paper. Therefore, this proves the significance of results obtained within this study.

However, I believe the results in this paper could have been improved further by using an approach which will be explained using Table 30.

Instance	Feature 1	…	Feature 128	Label
0	140.53	…	54.03	LeftSwipe
1	130.35	…	59.48	LeftSwipe
2	137.24	…	61.53	LeftSwipe
3	139.98	…	63.91	LeftSwipe
4	142.46	…	60.13	LeftSwipe
5	118.45	…	40.52	LeftShake
6	128.64	…	48.93	LeftShake
7	139.53	…	52.10	LeftShake
8	158.19	…	47.86	LeftShake
9	163.44	…	42.61	LeftShake

Table 30: A Sample training dataset

It can be seen that instance 0 and instance 7 are very similar in their values for the features, but as they both belong to different classes, it can be very difficult for classifiers to make a correct prediction. Therefore, making use of classifiers in a hidden Markov model maybe worth implementing as this would take into consideration a previous observation to help determine the current observation when necessary. A multimodality approach including the best model within this study and an RNN which includes long short-term memory layers would also have a similar impact upon model accuracy. However, all three types of fusion (data-level, feature-level and decision-level)

should be considered when implementing this and with an evolutionary searching algorithm similar to what was incorporated within [26] as this would ensure local optima does not impede the model's performance.

However, these fusion approaches would certainly slow down the training time duration, and perhaps even the real-time predictions of observations, as they rely on previously seen data to make predictions. In particular, if too many previous samples were to be remembered in the RNN approach, this would be inappropriate for a real-time scenario especially if only one sample is recorded every 0.2 seconds. However, by increasing the frequency of collecting more data samples, this would reduce the real-time lag caused by the RNN model. Needless to say that the data within the training dataset can no longer be shuffled if this approach is considered and this therefore may introduce a bias.

Due to the KNN classifier having very few hyperparameters, it is slightly difficult to improve its accuracy on unseen data by tuning these as the number of possible combinations are extremely limited. The '*n_neighbors*', '*p*' and '*metric*' hyperparameters are those which commonly lead to improvements in prediction accuracies, although in this study only the '*n_neighbors*' hyperparameter was tuned to simplify the tuning process. This was also the case with the random forest classifier as the majority of the hyperparameters available were not included. However, the '*n_estimators*', '*criterion'*, '*max_depth*' and '*max_samples*' were all tuned as they are considered some of the most important hyperparameters for this classifier. Ideally, more trials would have been specified with the optuna tuning approach as this would have provided more insight into possible hyperparameter values. Unfortunately, computational limitations severely prevented this, leading to a compromise between the quality of the returned values and the computational cost (an adequate number of 30 trials were therefore used).

Something which was picked up during this study however, was that by tuning the hyperparameters of classifiers, this improved their accuracies on training data, but degraded the accuracy on the unseen testing data which led to an increase of model overfitting. Knowledge of this helped to explain why the trial-and-error graph-based approach, which was implemented to use the values of hyperparameters belonging to trials that obtained the lowest objection values, was successful. The theory behind this being that the lower the objection value, the poorer the accuracy of the classifier on training data but the better the generalisation of unseen data. This was proved successful with evidence of this where the random forest classifier improved from 68% to 69% on unseen testing data. However, once the overfitting is removed or reduced, the hyperparameter tuning process did begin to output parameter values which improved the accuracies of both testing and unseen data.

The elimination of highly correlated features from the datasets led to the significant progress made within this study as it reduced the level of overfitting dramatically. Therefore, the 'best' hyperparameter values which were returned after tuning the models, did successfully have an improvement of the random forest model from 83% to 86% on unseen data. The trial-and-error approach using the optuna graphs also proved successful here, as the values obtained using this method outputted the same accuracies on training and unseen data which were produced using the returned parameter values, even though they were completely different values.

It can also be concluded that auto machine learning methods provided minimal benefits to this study too. This was seen with both of the Microsoft Azure models outputted where none of the

models scored above 60% accuracy, whilst the best TPOT model only predicted a poor 21% of the unseen data correctly.

It was evident that the raw feature subspace was highly linearly compressible as 95% variance of the raw data could be explained using 12 components and 23 components could be used to explain 99% variance rather than the original 428 features. However, with the random forest classifier struggling to make predictions using the new feature subspace (a poor 34% and 38% accuracy on unseen data using 95% and 99% variance respectively), it was clear that PCA only had a positive influence upon the KNN classifier (an increase from 65% to 67% accuracy predicting classes using unseen data).

Although the classes were considered as 'balanced' within Microsoft Azure, I believe that this could be argued against due to the range between the majority and minority classes being rather large. By balancing the classes, this improved the KNN classifiers accuracy on unseen data to 84%. However, the random forest classifier obtained 85%, which was only a 1% decrease, but may be the reason behind Azure making the assumption that classes were 'balanced', as the impact on accuracies from balancing classes was minimal. However, as the number of observations within the balanced dataset was significantly increased (6768 extra observations) due to oversampling, this led to a smaller number of correlated features being removed from the dataset using feature selection. This therefore caused an increase in computational cost to train the model. Due to both this disadvantage as well as producing a poorer accuracy on the best model, this meant that it was not appropriate to use oversampling for the balancing of the classes. However, it would be interesting to use the optuna hyperparameter tuning approach on the balanced dataset which has had feature selection applied to it, as the optimal hyperparameters of the classifier which are returned could potentially lead to an accuracy greater than 86% on unseen data. Unfortunately, time restrictions prevented this, as well as the possibility to experiment with both undersampling, and a combination of the two sampling methods.

None of the boosting classifiers, which included XGBoost, LGBoost and Adaboost had any improvement of accuracy on unseen data, although adaboost did score a respectable 82%. However, as the tuned random forest was provided as the estimator within the boosting classifier, it was clear that adaboost was a redundant model, as the estimator itself had a higher accuracy on unseen data and was less computationally expensive.

More evaluation metrics should have been included within this study to determine how well the models performed rather than just using their accuracies and confusion matrices. However, as the other metrics can be determined manually using a confusion matrix, I felt that these would have been slightly unnecessary to include, and most likely would not have been used other than for presentation.

Limitations within the Study

There are some limitations within the study which is expected when using sensors as they cannot offer 100% reliability. The range which the sensors can detect hands is a limitation, as an increase in the distance between the hands and the leap motion controller leads to a diminishing hand detecting accuracy. This also occurs if the hands are presented too close to the sensor. Therefore, hand data should have ideally been collected from varying distances to reduce the impact of this limitation.

It is also common for the leap motion controller to sometimes glitch when tracking hands, and therefore, the hand model representations within the leap motion visualiser may be seen in an unnatural pose, or the incorrect hand may sometimes be detected. This is a limitation as instances within the dataset will no doubt be impacted upon this. As the frequency of these glitches are minimal, they are relatively easy to remove from the dataset. Due to the labelling of the classes being specific, it was easy to determine whether the right hand or left hand were intended to be present. Therefore, if numerical values are recorded for the opposite hand, this instance within the training dataset was removed. This could have been done using the drop method in pandas, although due to the ordering of features within the dataset, this made it simple to just manually remove the few inaccurate recordings. The glitches which occur with the correct hand however, will most likely be ignored by the better performing classifiers such as the random forest, and therefore do not require any manipulation, assuming that significantly more instances were accurately collected (otherwise the classifier will expect glitches in order to perform correctly).

Unfortunately, due to time limitations, I was unable to include the final stage of the project, where the python script was to be run in the unity game engine, allowing the control of a robotic arm in a virtual world. However, as I chose to invest more time into the machine learning side of the project, in order to obtain a more accurate model, was unfortunately not capable of completing this stage.

Computational limitations have also prevented the possibility of completing tuning processes such as Grid Search CV to tune hyperparameters, leading to theses methods being interrupted early. As the study consisted of five subjects within the same household who were all white and over the age of 18, this also results in limitations. Subjects would have ideally had a wider variety of hand sizes, as well as a larger variety of ethnical backgrounds taking part within the study. Although, due to covid-19 restrictions, this has impeded the possibility of doing so as well as this taking a considerable amount of time to collect. However, it should be noted that a larger dataset containing dynamic hand gestures from these additional subjects, would considerably improve the reliability of the model. The unseen testing dataset would also have ideally been collected from a subject who was not involved within the study to ensure reliability and robustness of the model.

Environmental conditions should also not be ruled out as limitations. Although poor lighting does not affect how well hands are detected due to the IR LEDs within the controller, the occlusion of hands is still a limitation. A common example of this is smudging on the sensor window. The third-party script that was used to collect the hand data only recorded data whilst hands were detected within the controller's view. Therefore, it was a necessity to ensure that the visualiser could be viewed whilst the timing of the data was taking place. This allowed the stopwatch to be paused and continued once the hands were visible again. This strategy worked to an extent but an improved approach would have been to adapt the third-party script so that a specific number of frames were collected for each recording (300 instances should be recorded for every minute of data collected). This would ensure that all classes are perfectly balanced without the need for resampling at a later stage. The orientation of the leap motion controller is also a limitation as it could result in inverse values being recorded (positive values where negatives values are expected), leading to poor performance of models. To prevent this, the light on the leap motion controller was facing away from the subjects when they demonstrated the gestures, as well as enabling the Auto-orient Tracking setting.

However, the largest limitation of this study was that as I had no understanding of machine learning beforehand. Therefore, it was hard to determine when I had invested enough time into the

preparation section of the study to start on the implementation section. This therefore led to some content being researched parallel to implementing models, which was beneficial in that it gave me the opportunity to improve my understanding of these topics by applying them as they were researched, although this undoubtedly slowed down the implementation stage, as some of these newly researched techniques did not improve models. Some of the methods such as those within the feature selection and feature extraction topics were vital to consider before the application of classifiers to data, although unfortunately, I only discovered these methods were available after various models had already been implemented. A lack of knowledge about many of the less common hyperparameters and how they influence classifiers was certainly another limitation, and something which I would like to research beyond this project.

Reflection of the Study

This project has been exceptionally challenging at times as support has been limited. Before I embarked on this challenge I had very basic knowledge of the python programming language let alone the machine learning and data science aspects of the study. However, with the use of many websites, books, blogs, academic papers and the occasional assistance from lecturers as well as PhD students, have managed to grasp a high level of understanding within a remarkably wide variety of topics within the machine learning field, with some of the PhD students even admitting they had not read up on some of these themselves. The knowledge I have gained has not been unnoticed by others in the machine learning sector either, as I have been offered a graduate role at CGI as a machine learning engineer and java developer, helping me to take the next step of implementing other areas within machine learning such as reinforcement learning, deep learning and supervised learning.

The background research involving academic papers was rather overwhelming at time as certain topics such as hidden Markov models and the various segmentation techniques with various colour spaces was slightly difficult to wrap my head around with little support to consolidate my understanding even after reading several papers relating to the topics.

If I were to complete the study again, I believe it would have a considerable improvement of not only the efficiency, but also the results of models obtained. Something which this project lacked was a pre-understanding of the various techniques which were available to utilise. An example of this is the imbalanced-learn library which can be used to balance classes through the use of resampling. Unfortunately, I only came across this topic towards the end of the project and therefore was restricted with what I could implement to try and improve model accuracies. However, after reading up on the majority of these libraries, classes and methods, I am now extremely confident that I can implement models which achieve a high degree of accuracy on both training and unseen data using classical machine learning techniques.

The initialisation of the software was fairly difficult to install correctly as specific versions were required, to ensure there was flawless integration which I was not originally aware of. An example of this was python 3 not being compatible with the leap motion controller and therefore required python 2 to be installed so that data could be collected. However, with the use of some extremely vague instructions over emails and many attempts of downloading software on both windows and macOS platforms (and even a raspberry Pi), I managed to install the correct versions and collect the hand gesture data. The anaconda navigator and all of the relevant libraries were also

problematic as originally, I was unaware of how to install python packages which were not available in the anaconda directory, and did not understand the difference between anaconda environments.

At times I was considerably naïve in that I believed that once all of the classifiers had been applied to the training dataset, at least one of them would have a high accuracy on unseen data. The reasoning for thinking this was that the accuracy of the models within weka were all 80%+ for the majority of the datasets. However, these accuracies are arguably not robust as the classifiers may perform poorly on an unseen dataset which is independent to the one used to train the models.

Ideas for Future Work

Future work which could extend from this study includes the usage of the best model in a real-time scenario such as a conversation or perhaps controlling robotics. Trialling combinations of dimensionality reducing techniques before applying the classical machine learning classifiers, or using Deep Neural Networks (DNNs) to acquire models would also be worthwhile to consider implementing. If the third-party data collection script maintained the original feature of collecting images of frames whilst the numerical hand data was recorded, this would also allow Convolutional Neural Networks (CNNs) to be implemented to classify gestures. It would also be interesting to research into incorporating reinforcement learning to autonomise robotics, however this would eliminate the requirement of involving hand gestures and could potentially be fatal if not demonstrated in a controlled environment such as the Unity game engine beforehand. Applying machine learning to datasets which consist of bimanual gestures rather than just unimanual gestures would also be an interesting study to research as this is frequently used within sign language.

References

[1] S. Rajaganapathy, B. Aravind, B. Keerthana and M. Sivagami, "Conversation of Sign Language to Speech with Human Gestures," 2015.

[2] F. Weichert, D. Bachmann, B. Rudak and D. Fisseler, "Analysis of the Accuracy and Robustness of the Leap Motion Controller," 2013.

[3] A. Bracegirdle, "Investigating the Usability of the Leap Motion Controller: Gesture-Based Interaction with a 3D Virtual Environment," 2014.

[4] P. Wozniak, O. Vauderwange, A. Mandal, N. Javahiraly and D. Curticapean, "Possible applications of the LEAP motion controller for more interactive simulated experiments in augmented or virtual reality," 2016.

[5] J. M. Palacios, C. Sagüés, E. Montijano and S. Llorente, "Human-Computer Interaction Based on Hand Gestures Using RGB-D Sensors," 2013.

[6] Q. D. Smedt, H. Wannous and J.-P. Vandeborre, "Skeleton-based Dynamic hand gesture recognition," Lille, France, 2016.

[7] A. G. Jaramillo and M. E. Benalcázar, "Real-time hand gesture recognition with EMG using machine learning," Salinas, Ecuador, 2017.

[8] Y. Xu and G. Pok, "Identification of Hand Region Based on YCgCr Color Representation," *International Journal of Applied Engineering Research,* vol. 12, no. 6, pp. 1031-1034, 2017.

[9] S. Kolkur, D. Kalbande, P. Shimpi, C. Bapat and J. Jatakia, "Human Skin Detection Using RGB, HSV and YCbCr Color Models," 2017.

[10] K. B. Shaik, G. Packyanathan, V. Kalist, S. B. S and J. M. M. Jenitha, "Comparative Study of Skin Color Detection and Segmentation in HSV and YCbCr Color Space," *Procedia Computer Science,* vol. 57, pp. 41-48, 2015.

[11] R. P. K. Poudel, H. Nait-Charif, J. J. Zhang and D. Liu, "Region-Based skin color detection," 2012.

[12] M. J. Jones and J. M. Rehg, "Statistical Color Models with Application to Skin Detection," 2002.

[13] K. Manikandan, "Hand Gesture Detection and Conversion to Speech and Text," 2018.

[14] J. L. Raheja, G. A. Rajsekhar and A. Chaudhary, "Controlling a remotely located Robot using Hand Gestures in Real-time: A DSP implementation," 2016.

[15] J. Yamato, J. Ohya and K. Ishii, "Recognizing human action in time-sequential images using hidden Markov model," 1992.

[16] M. Petkovic, W. Jonker and Z. Zivkovic, "Recognizing Strokes in Tennis Videos Using Hidden Markov Models," 2001.

[17] A. Alizadeh, "Gesture Recognition based on Hidden Markov Models fromJoints' Coordinates of a Depth Camera for Kids age of 3–8," Helsinki, 2014.

[18] "NI mate," 2012. [Online]. Available: https://ni-mate.com/.

[19] P. Trigueiros, F. Ribeiro and L. P. Reis, "A comparison of machine learning algorithms applied to hand gesture recognition," 2012.

[20] "Rapid Miner," 2006. [Online]. Available: https://rapidminer.com/.

[21] M. K. Ahuja and A. Singh, "Static vision based Hand Gesture recognition using principal component analysis," 2015.

[22] S. Kumar, T. Srivastava and R. S. Singh, "Hand Gesture Recognition Using Principal Component Analysis," *Asian Journal of Engineering and Applied Technology,* vol. 6, no. 2, pp. 32-25, 2017.

[23] A. Sharma, A. Mittal, S. Singh and V. Awatramani, "Hand Gesture Recognition using Image Processing and Feature Extraction Techniques," *Procedia Computer Science,* vol. 173, pp. 181-190, 2020.

[24] O. K. Oyedotun and A. Khashman, "Deep learning in vision-based static hand gesture recognition," *Neural Computing and Applications,* vol. 28, pp. 3941-3951, 2017.

[25] K. Lai and S. N. Yanushkevich, "CNN+RNN Depth and Skeleton based Dynamic Hand Gesture Recognition," 2020.

[26] J. Bird, A. Ekart and D. R. Faria, "British Sign Language Recognition via Late Fusion of Computer Vision and Leap Motion with Transfer Learning to American Sign Language," 2020.

[27] Unity, "Unity ML-Agents Toolkit," [Online]. Available: https://github.com/Unity-Technologies/ml-agents.

[28] M. S. Nadeem, "Using Deep Neuroevolution to train Deep Reinforcement Learning Agents".

[29] OpenAI, "Part 2: Kinds of RL Algorithms," 2018. [Online]. Available: https://spinningup.openai.com/en/latest/spinningup/rl_intro2.html.

[30] integrate.ai, "What is Model-Based Reinforcement Learning?," 2018. [Online]. Available: https://medium.com/the-official-integrate-ai-blog/understanding-reinforcement-learning-93d4e34e5698.

[31] R. S. Sutton and A. G. Barto, Reinforcement Learning: An Introduction, 2 ed., 2017.

[32] H. Nguyen and H. La, "Review of Deep Reinforcement Learning for Robot Manipulation," 2019.

[33] F. Amici, A. Filippo and J. Call, "Response facilitation in the four great apes: is there a role for empathy?," *Primates,* vol. 55, pp. 113-118, 2013.

[34] E. Yavşan and A. Uçar, "Gesture imitation and recognition using Kinect sensor and extreme learning machines," *Measurement,,* vol. 94, pp. 852-861, 2016.

[35] J. LaViola, "A survey of hand posture and gesture recognition techniques and technology," 1999.

[36] Anaconda, "Anaconda-Navigator," [Online]. Available: https://anaconda.org/anaconda/anaconda-navigator.

[37] Ultraleap, "How Hand Tracking Works," Ultra Leap, 22 July 2013. [Online]. Available: https://www.ultraleap.com/company/news/blog/how-hand-tracking-works/.

[38] F. Campelo, "UNIT 3: CLASSIFICATION I," 2021.

[39] J. Brownlee, "Logistic Regression for Machine Learning," 1 April 2016. [Online]. Available: https://machinelearningmastery.com/logistic-regression-for-machine-learning/.

[40] X. Liu, J. Zhou and H. Qian, "Comparison and Evaluation of Activation Functions in Term of Gradient Instability in Deep Neural Networks," 2019.

[41] L. Breiman, "Bagging Predictors," Kluwer Academic Publishers, 1996.

[42] S. Raschka, "STAT 479: Machine Learning Lecture Notes," 2019. [Online]. Available: https://sebastianraschka.com/pdf/lecture-notes/stat479fs19/07-ensembles__notes.pdf.

[43] R. Kohavi, "A study of cross-validation and bootstrap for accuracy estimation and model selection," 1995.

[44] J. J. LaViola Jr., "A survey of Hand Posture and Gesture Recognition Techniques and Technology," June 1999.

[45] H. Chung, Y. Chung and W. Tsai, "An Efficient Hand Gesture Recognition System Based on Deep CNN," 2019.

[46] R. Chauhan, K. K. Ghanshala and R. C. Joshi, "Convolutional Neural Network (CNN) for Image Detection and Recognition," 2018.

[47] M. R. Ahsan, M. I. Ibrahimy and O. O. Khalifa, "Electromygraphy (EMG) signal based hand gesture recognition using artificial neural network (ANN)," Kuala Lumpur, 2011.

[48] T. Dietterich , "Ensemble Methods in Machine Learning," in *Multiple Classifier Systems*, Springer, Berlin, Heidelberg, 2000, pp. 1-15.

[49] H. Butler , M. Friend , K. Bauer and T. Bihl, "The effectiveness of using diversity to select multiple classifier systems with varying classification thresholds," *Journal of Algorithms & Computational Technology,* pp. 187-199, 2018.

[50] A. Joshi , C. Monnier, M. Betke and S. Sclaroff, "A random forest approach to segmenting and classifying gestures," Ljubljana, 2015.

[51] R. Shrivastava, "A hidden Markov model based dynamic hand gesture recognition system using OpenCV," Ghaziabad, 2013.

[52] C. Yu, X. Wang, H. Huang, J. Shen and K. Wu, "Vision-Based Hand Gesture Recognition Using Combinational Features," Darmstadt, 2010.

[53] F. Amici, A. Filippo and J. Call, "Response facilitation in the four great apes: is there a role?," *Primates,* vol. 55, pp. 113-118, 2013.

[54] "Stratified K Fold Cross Validation," 29 May 2021. [Online]. Available: https://www.geeksforgeeks.org/stratified-k-fold-cross-validation/.

[55] J. Brownlee, "Recursive Feature Elimination (RFE) for Feature Selection in Python," 25 May 2020. [Online]. Available: https://machinelearningmastery.com/rfe-feature-selection-in-python/.

[56] O. Markos, "Feature selection using Scikit-learn," 28 January 2020. [Online]. Available: https://medium.com/analytics-vidhya/feature-selection-using-scikit-learn-5b4362e0c19b.

[57] J. Brownlee, "Feature Selection in Python with Scikit-Learn," 14 July 2014. [Online]. Available: https://machinelearningmastery.com/feature-selection-in-python-with-scikit-learn/.

[58] J. Brownlee, "Discrete Probability Distributions for Machine Learning," 20 September 2019. [Online]. Available: https://machinelearningmastery.com/discrete-probability-distributions-for-machine-learning/.

[59] J. Brownlee, "A Gentle Introduction to Imbalanced Classification," 23 December 2019. [Online]. Available: https://machinelearningmastery.com/what-is-imbalanced-classification/.

[60] J. Brownlee, "4 Types of Classification Tasks in Machine Learning," 8 April 2020. [Online]. Available: https://machinelearningmastery.com/types-of-classification-in-machine-learning/.

[61] J. Brownlee, "How To Use Classification Machine Learning Algorithms in Weka," 25 July 2016. [Online]. Available: https://machinelearningmastery.com/use-classification-machine-learning-algorithms-weka/.

[62] J. Brownlee, "Linear Discriminant Analysis for Machine Learning," 6 April 2016. [Online]. Available: https://machinelearningmastery.com/linear-discriminant-analysis-for-machine-learning/.

[63] T. Boyle, "Feature Selection and Dimensionality Reduction," 25 March 2019. [Online]. Available: https://towardsdatascience.com/feature-selection-and-dimensionality-reduction-f488d1a035de.

[64] A. Band, "Multi-class Classification — One-vs-All & One-vs-One," 9 May 2020. [Online]. Available: https://towardsdatascience.com/multi-class-classification-one-vs-all-one-vs-one-94daed32a87b.

[65] J. Brownlee, "One-vs-Rest and One-vs-One for Multi-Class Classification," 13 April 2020. [Online]. Available: https://machinelearningmastery.com/one-vs-rest-and-one-vs-one-for-multi-class-classification/.

[66] J. Brownlee, "What is a Confusion Matrix in Machine Learning," 18 November 2016. [Online]. Available: https://machinelearningmastery.com/confusion-matrix-machine-learning/.

[67] J. Brownlee, "TPOT for Automated Machine Learning in Python," 9 September 2020. [Online]. Available: https://machinelearningmastery.com/tpot-for-automated-machine-learning-in-python/.

[68] J. Brownlee, "Ordinal and One-Hot Encodings for Categorical Data," 12 June 2020. [Online]. Available: https://machinelearningmastery.com/one-hot-encoding-for-categorical-data/.

[69] J. Brownlee, "Scikit-Optimize for Hyperparameter Tuning in Machine Learning," 4 September 2020. [Online]. Available: https://machinelearningmastery.com/scikit-optimize-for-hyperparameter-tuning-in-machine-learning/.

[70] S. Raschka and V. Mirjalili, Python Machine LEarning, Third ed., Packt Publishing Ltd., 2019.

[71] J. Brownlee, "Bagging and Random Forest for Imbalanced Classification," 2020. [Online]. Available: https://machinelearningmastery.com/bagging-and-random-forest-for-imbalanced-classification/.

[72] J. Brownlee, "Random Oversampling and Undersampling for Imbalanced Classification," 2020. [Online]. Available: https://machinelearningmastery.com/random-oversampling-and-undersampling-for-imbalanced-classification/.

[73] J. Brownlee, "Boosting and AdaBoost for Machine Learning," 2020. [Online]. Available: https://machinelearningmastery.com/boosting-and-adaboost-for-machine-learning/.

[74] J. Brownlee, "How to Use StandardScaler and MinMaxScaler Transforms in Python," 2020. [Online]. Available: https://machinelearningmastery.com/standardscaler-and-minmaxscaler-transforms-in-python/.

[75] L. Chen, "Support Vector Machine — Simply Explained," 7 January 2019. [Online]. Available: https://towardsdatascience.com/support-vector-machine-simply-explained-fee28eba5496.

[76] J. Brownlee, "Crash Course On Multi-Layer Perceptron Neural Networks," Machine Learning Mastery, 17 May 2016. [Online]. Available: https://machinelearningmastery.com/neural-networks-crash-course/.