MCAST

Classifying Healthy Standing Desk Posture Using Machine Learning

Noah Sultana

Supervisor: Ms Annalise Consoli

June, 2022

A dissertation submitted to the Institute of Information and Communication Technology in partial fulfillment of the requirements for the degree of B.Sc. (Hons.) Multimedia Software Development

Authorship Statement

This dissertation is based on the results of research carried out by myself, is my own composition, and has not been previously presented for any other certified or uncertified qualification.

The research was carried out under the supervision of Ms Annalise Consoli.

Noah Sultana

June 5, 2022

Copyright Statement

Acknowledgements

I hereby acknowledge and thank the following people who have supported me, not only throughout this dissertation, but also throughout my degree.

Firstly, I would like to acknowledge my mentor throughout this study Ms Annalise Consoli for her thorough guidance and support.

I also want to thank my family, especially my parents Joseph & Nathalie Sultana, for always believing in me and backing up every decision that I made throughout my journey.

Finally, with great gratitude, I want to thank my girlfriend Justine Thea Cefai for being my backbone throughout this process and my long-time friend Keith Azzopardi for his constant encouragement.

Table of Contents

List of Abbreviations

AI Artificial Intelligence. vi, 1, 6, 7, 16, 62, 67

AUC Area Under Curve. 2, 45

EF Engineered Feature. vi, x, 1, 2, 5, 32, 38, 39, 40, 42, 43, 45, 47, 50, 54, 61, 64, 66

FE Feature Engineering. 29, 42

FN False Negatives. 44, 45, 47, 50, 52, 54

FP False Positives. 44, 45, 47, 50, 52, 54

LR Logistic Regressor. vi, ix, 1, 4, 11, 33, 35, 37, 39, 42, 50, 52, 57, 59, 60, 63, 64

ML Machine Learning. vi, ix, 7, 17, 21, 62

MLP Multilayer Perceptron. v, vi, ix, 1, 4, 7, 10, 18, 21, 33, 34, 37, 39, 40, 42, 44, 45, 47, 50, 54, 57, 61, 63, 64

RF Random Forest. v, vi, ix, 1, 4, 7, 9, 17, 18, 21, 33, 34, 37, 39, 40, 42, 52, 57, 63

ROC Receiver Operating Characteristic. 44, 45

SVM Support Vector Machine. v, vi, ix, 1, 4, 7, 8, 16, 18, 21, 33, 34, 37, 39, 42, 44, 47, 50, 57, 63, 64

TN True Negatives. 47

List of Figures

List of Tables

Abstract

Bad posture raises long lasting health considerations. Studies have shown that it has a negative impact on work efficiency. For these reasons, it is critical to have a non-invasive system in place that can classify between good and bad postures when utilising a standing desk. Since standing desks are relatively new on the market, current systems for detecting postures are expensive as they come in-built with the standing desk. Other systems require the use of sensors which are not easy to come by. The objective of this study was to develop a non-invasive technology that can detect a good or bad posture from a video feed, using Artificial Intelligence (AI) algorithms to determine standing desk ergonomics. This notion makes it very affordable for the general population whilst being cost effective for enterprises to implement. The MoveNet Lightning architecture will be used to extract body keypoints in all of the models created within this project. This lightweight version of the architecture will allow for future work to implement these models in a mobile application. Within this work, a custom dataset was built to simulate different postures when using standing desks from three angles: A side view, a back view angle, and a combined view. This data is then pre-processed and supplied into four separate classifiers (MLP, SVM, RF & LR) for training. Finally, an Engineered Feature (EF) is implemented to calculate the angle between the body keypoints at run time in order to evaluate whether it improves the models' performance. Finally, all of the factors were combined to generate 24 separate models. Each model was assessed using

different metrics such as, accuracy, F1-Score and Area Under Curve (AUC). To begin with, when determining which dataset view was best, it was discovered that the side view outperforms the back view dataset. Above all, in 23 out of 24 instances, the combined view dataset outperformed the other stand-alone views. Secondly, when comparing the overall total difference between models that used the EF and the models that did not, it was found that the total overall accuracy improved by 4.36%. Finally, the best performing classifier was found to be the MLP classifier using the combined dataset without utilising the EF. The accuracy and F1-Score of this model were 98.5% and 98.5%, respectively. This study surpassed many others in the field of posture classification, laying the groundwork for future research in the field of posture detection for standing desk ergonomics. More accurate findings could be obtained by improving the dataset and using better pre-processing techniques. Above all, by enhancing the dataset, the models will be more generalisable to various scenarios such as, the distances between the user and the video input device. This appeals to many businesses that want to get the most out of their staff, such as office workers and shop assistants, whilst also keeping them safe from future health implications.

Chapter 1

Introduction

The value of good posture is not always prioritized and its awareness is sometimes only achieved after health issues arise. Aside from the apparent discomfort and soreness that bad posture can cause, research has found a link between poor work efficiency and poor posture (Kaushik & Charpe, 2008). In a corporate atmosphere, efficiency is given a high priority in the workplace, either by educating individuals to be more efficient, or by installing technologies that automate certain processes. Having said that, the majority of corporate firms have shunned upon one of the most important bi-products of efficiency, namely posture (Christy, 2019).

1.1 Problem Definition & Motivation

This study was carried out for three main reasons. To begin with, there is little to no studies on pose estimation in standing postures, particularly in the context of a standing desk environment since these have been relatively introduced lately. Secondly, current solutions, such as smart desks, either necessitate the purchase of external sensors, which can be purchased individually, or another solution is to buy a smart standing desk which

can be quite expensive. As a result, a product that detects posture using only a webcam would be ideal. Thirdly, as stated by Christy (2019), a lot of people fail to recognize the importance of proper posture thus, having a system in-place that can notify users to correct their posture, will enhance efficiency and endure less muscular pain caused by bad posture. Wearable sensors such as accelerometers and gyroscope sensors are currently used for posture assessment (Alberts et al., 2015), however they require extra setup time and physical hardware, increasing the products cost. Aside from that, recently smart desks have been launched to the market, with the potential to alert an individual when they have poor posture. However, many of them are quite costly. A supervised learning approach was used in this study, which only required a simple camera or webcam to recognise good and bad postures, making it very simple to use and accessible for anyone with a computer.

1.2 Aims & Objectives

The goal of this study is to determine if training a model to distinguish the difference between good and bad postures when utilising a standing desk can be done using the extracted body keypoints. On a custom-built dataset, this prototype leverages the MoveNet lightning pre-trained model for body keypoint extraction. This study will also determine which camera point of view angle is best when detecting this type of posture, by training four classifiers with a back angled view, a side angled view and a combined view of the two posture datasets mentioned previously. These datasets were generated from custom data acquired from videos with the different camera angles. As a result of adherence to a set of rules administered by Chandler (2022), this data was labeled as good and bad postures. Based on results from past studies, the four classifiers that were chosen to be used for training were: MLP classifier, SVM classifier, RF classifier and LR (Boateng et al., 2020). Feature engineering was composed from pre-existing model

keypoints to identify how the models would behave and if the new feature improved the classifiers' accuracy. Finally, a total of six datasets were composed: Side View dataset, Back View dataset, Combined dataset, and the same three datasets with the Engineered Feature (EF) included. All of these datasets were fed into the various classifiers to see which one produced the best results.

1.3 Paper Structure

The following is the structure that this research study follows: The second chapter examines the present methods and methodologies for posture detection classification. As a result, the literature delves into the history of what constitutes as a good and bad posture, as well as the intricacies of various classification models and frameworks for keypoint extraction. Chapter 3, entails how the data was obtained, pre-processed and used to train the different models. This section also covers the prototype pipeline as well as feature engineering. Chapter 4 analyses the data and discusses the various pipeline classifiers results, thus answering the research questions and features the test cases results. Finally, Chapter 5 closes the study by identifying future work that may be done to improve and expand on this prototype and its suggestions.

Chapter 2

Literature Review

2.1 Introduction

In today's day and age, Artificial Intelligence (AI) is increasingly being used in a variety of fields, including one of its branches, human pose estimation (Sarafianos et al., 2016). The latter has taken on challenges to improve and streamline various processes in the health care industry. Pose estimation has been employed in a vast array of applications, including posture classification, hand gesture classification, and fitness activities (Jo & Kim, 2022). Intelligent models like MoveNet and OpenPose have been trained to recognize the joint locations of a person's skeletal structure from an image or video, making this possible. This has allowed the output matrix of the position of these keypoints at every frame and use that data to train different models for various purposes.

This literature will mainly focus on the use of Artificial Intelligence (AI) in standing posture detection. This will entail the different implementations to classify different postures from research conducted throughout the years, whilst also declaring a comparison of what architectures and classifiers work best with pose estimation.

2.2 Definitions & Techniques

This section will clarify the techniques and definitions utilised in this work and will act as a reference throughout the research.

2.2.1 Machine Learning

Fundamentally, ML is a subset branch of AI. Essentially, this is when a machine or computer accomplished a task without the assistance of a human, as it can learn on its own from data (Alzubi et al., 2018). The pipeline can be visualised in Figure 2.1. RF, SVM and MLP are some instances of ML models. The pipeline of building an ML model is as follows:

- Data acquisition and pre-processing.
- Obtaining the important features from the data.
- Choosing the correct algorithm based on the problem being solved.
- Choosing the correct models and finding the best fit parameters.
- Training the models based on the acquired and pre-processed data.
- Evaluating the performance of the models against a test set.

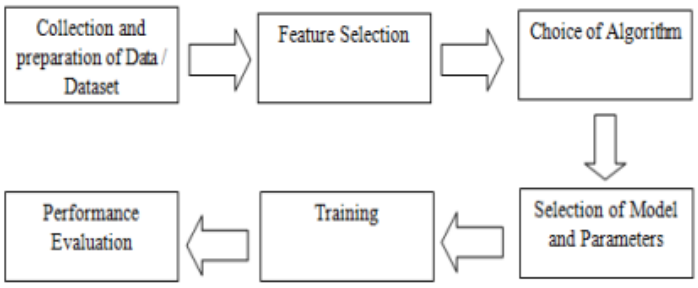

Figure 2.1: Machine Learning (ML) Pipeline
source: (Alzubi et al., 2018)

2.2.2 SVM Classifier

This classifier usually performs very well in many different contexts as it works in both linear and non-linear circumstances. The Support Vector Machine (SVM) classifiers' main purpose is to locate the optimum hyperplane with the shortest distance to the nearest support vectors or data points in order to avoid class misclassification as much as possible (Bonaccorso, 2017), as shown in Figure 2.2. This classifier features a set of hyper-parameters worth noting, as they affect the accuracy of the final models. The 'C' parameter is a regularisation parameter that tells the SVM how crucial it is to avoid classification errors. The 'Kernel' argument defines the type of kernel that will be used used to aid the separation of the observations. Finally, the 'gamma' parameter is the kernel's co-efficient which is not compatible with a linear kernel. This controls the influence that a singular support vector has; a low value indicates that more data points will be clustered (*sklearn.svm.svc*, n.d.).

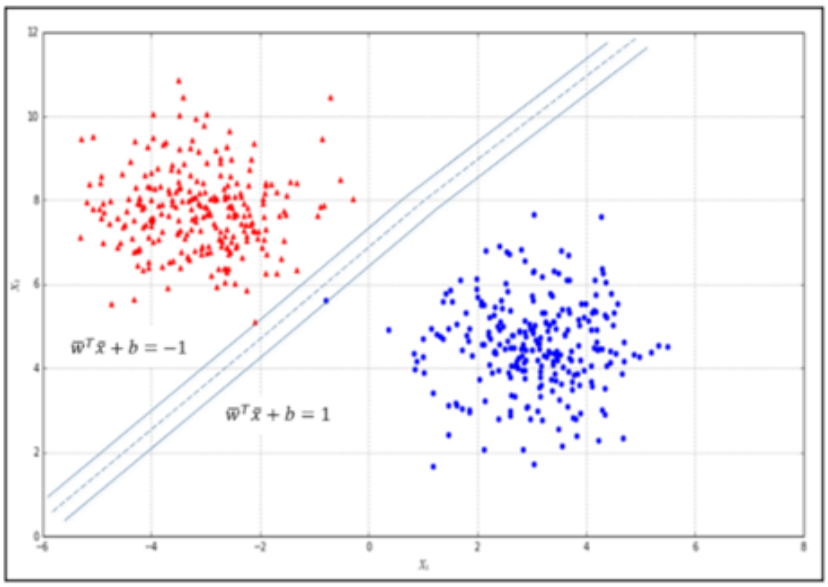

Figure 2.2: Support Vector Machine (SVM) classifier
source: (*sklearn.svm.svc*, n.d.)

2.2.3 RF Classifier

The Random Forest (RF) classifier constructs a collection of decision trees from random samples of the training data, which are subsequently divided into nodes as seen in Figure 2.3. To select the appropriate threshold for separating the data, a random subset of features for different nodes is employed. The traditional method of determining the best result is to choose the class with the most votes, but the scikit-learn implementation averages out all the results from the different nodes (Bonaccorso, 2017). 'n_estimators' which is the number of trees in the forest is one of the algorithm's most essential hyper-parameters. The 'min_samples_split' parameter controls how many samples do the nodes require to be split in. The final argument is the 'max_depth' which is used to limit the tree's growth (*ensemble methods*, n.d.).

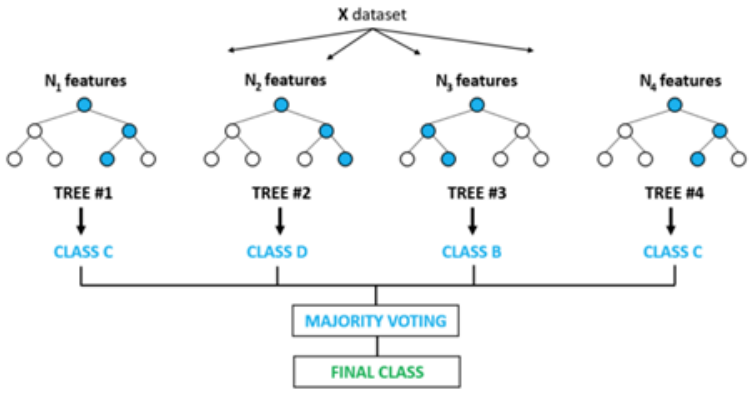

Figure 2.3: Random Forest (RF) Classifier
source: (Kirasich et al., 2018)

2.2.4 MLP Classifier

The Multilayer Perceptron (MLP) consists of an input layer, at least one hidden layer and an output layer. The size of the hidden and output layers must be equal. The varied initialization weights that exist between the input and the hidden layer are used to calculate the dot product of the inputs. These values are then pushed through an activation function, such as the sigmoid function, before being sent to the output layer (Ghate & Dudul, 2010). This is better depicted in Figure 2.4. The following are the hyper-parameters for this classifier: The 'batch_size' variable controls how many training examples are used per iteration. Secondly, the 'hidden_layer_sizes' parameter lets you choose the number of levels between input and output layers, as well as the number of nodes in each layer. Finally, 'early_stopping' will use 10% of the training data as validation data. The model will terminate training whenever the validation score stops improving. This is typically used to prevent the model from overfitting (*sklearn.neural_network.mlpclassifier*, n.d.).

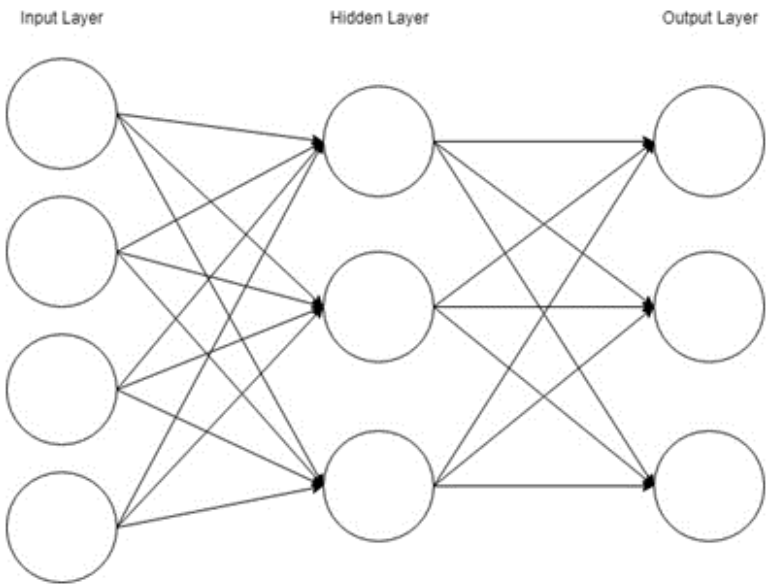

Figure 2.4: Multilayer Perceptron (MLP) Classifier
source: (stereo toolbox, 2022)

2.2.5 Logistic Regressor

In the spectrum of statistical models, the Logistic Regressor (LR) is a fairly simple and extensively used linear model. The sigmoid function is used in this model to try to determine the relationship between an input variable, X, and an output variable, Y as seen in Figure 2.5. The higher the correlation between the linear input and the outputs the more accurate it is. Due to its simplicity and capacity to communicate data in a straightforward manner, this method may outperform other nonlinear models (Kirasich et al., 2018). LR has two key hyper-parameters: 'C,' which operates similarly to SVM and is a regularization parameter. A high value indicates that the model should place more trust in the training data while a low value indicates the opposite. The second parameter is the 'penalty', which constitutes to penalizing the model for having too many variables. As a result, the coefficients that contribute the least are penalized (*sklearn.linear_model.logisticregression*, n.d.).

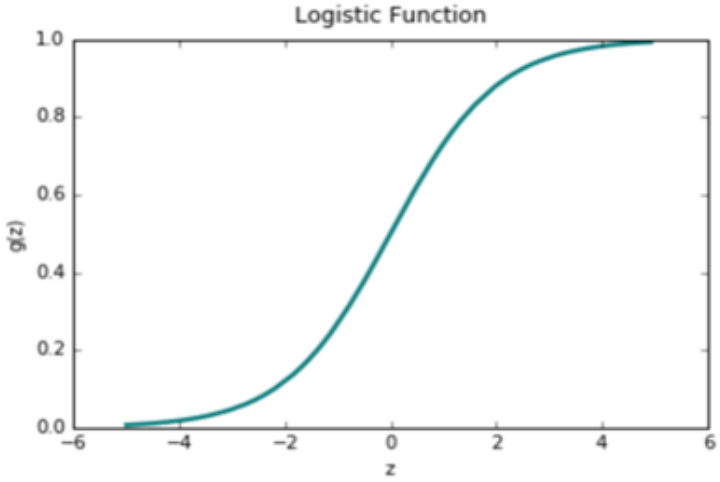

Figure 2.5: Sigmoid Function
source: (Kirasich et al., 2018)

2.2.6 Feature Engineering

As stated by Khurana et al., (2016), feature engineering is the process of extracting additional knowledge from gathered data that the model might not be able to generate on its own, in order to improve learning performance and predictive accuracy. Feature engineering is critical since it can have a significant impact on the models. In fact, feature engineering is one of the factors that distinguishes a good model from a bad model. Prior to selecting the engineered features, feature selection should be completed (Khurana et al., 2016). This will aid in the selection of the appropriate feature for further development. A feature in a dataset can be reduced to a new column of data prepared in an appropriate manner for the model to learn from.

2.3 Definition of 'Posture'

Throughout several years of research, many have studied what defines a good posture from a bad posture and what are the elements that differentiate them (Haslegrave, 1994). To comprehend this, one must first determine what the term posture actually implies. Posture is a relative arrangement of the parts of the body (Syrop & Bach, 1990). This means that posture can be defined as any position held while sitting, lying down, or standing. Posture is the consequence of a variety of habits developed over time, and it is these same posture patterns that determine whether a an individual has a good or bad posture.

2.3.1 Good Posture against Bad Posture

A statement from the American Academy of Orthopaedic Surgeons' Posture Committee was published in 1974 to describe the distinction between a good and bad posture. A healthy posture is one were "... the state of muscular and skeletal balance which protects the supporting structure of the body against injury or progressive deformity..." (AAOS, 1947), whilst a bad posture is "... a faulty relationship of the various parts of the body which produces increased strain on the supporting arches..." (AAOS, 1947). This statement has been cited in a number of research publications throughout the years, as it has been used a standard way to explain the differences between the two postures. From these distinctions it can be deduced that a good posture is one in which the body's structural position protects itself from future harm by maintaining a good balance between the muscular and skeletal systems. Whereas a bad posture is one in which the structural position at that point creates a strain on other supporting arches.

V. Korakakis et al., (2019), investigated how various physiotherapists perceived the optimal sitting and standing positions. Since the major purpose of this study is to detect various standing positions, the latter is more relevant. The study involved 544 physiotherapists who had to choose one of seven options for what they thought was the optimal standing position while also supporting their choice. The results of the survey revealed that, while two different standing postures were chosen as optimum, they shared two characteristics: they were both upright with less thoracic flexion and the head was aligned above the torso. This supports the argument made by the Posture Committee of the American Academy of Orthopaedic Surgeons, which claims that upright lordotic postures protect the muscular and skeletal system from injury (AAOS, 1947).

2.3.2 Rules That Define a Good Posture on Standing Desks

An online article by Chandler (2022), an occupational therapist, listed guidelines on how one should improve standing desk ergonomics. The following is a list of guidelines from this article:

- Maintain an "S curve" of the back by bending the knees slightly and keeping them shoulder width apart.
- Standing in one place for too long is not ideal, therefore one should stretch and move around while at the desk.
- Monitor should be placed at eye-level and around 20 to 28 inches from ones face to avoid neck strains and hunching.
- Wrists should be level with the elbow to avoid extra strain on the wrists.

2.4 Pose Estimation

Pose estimation is a subset of computer vision, whereby learning parameters of what is known as a pose, it is able to generate an appearance model (Hu, 2019). This is a technique for creating a geometric model of the person in the frame and extracting keypoints from it, which is useful for representing a pose. Apart from an appearance model, combining "spatial modelling of the human body with appearance modelling of body parts" is a typical strategy for pose estimation nowadays (Pishchulin et al., 2013). An example of pose estimation can be noted clearly in games and films. Actors frequently wear a suit with sensors, which allows motion capture to estimate their physique and apply effects during editing (Jo & Kim, 2022).

2.4.1 Pose Estimation Models

Different pose estimation models are developed utilising various algorithms and strategies to accomplish the same goal (extraction of body keypoints), however in a way that previous models failed to do. OpenPose, PoseNet, MoveNet Lightning, and MoveNet Thunder are a prominent example of keypoint extraction architectures. According to research that examined the performance of these models in mobile devices on a dataset of 1,000 photos, the higher the model's accuracy, the longer it takes to estimate postures (Jo & Kim, 2022). In fact, MoveNet Lightning required only 53.4 seconds to estimate 1,000 photos with an average accuracy of 75.1%, while PoseNet took 75.232 seconds with an average accuracy of 97.6% as seen in Tables 2.1 and 2.2.

Models	OpenPose	PoseNet	MoveNet Lightning	MoveNet Thunder
Average Time	643.536	75.232	53.458	139.974

Table 2.1: Keypoint extraction frameworks average times
source: (Jo & Kim, 2022)

Models	OpenPose	PoseNet	MoveNet Lightning	MoveNet Thunder
Group 1	78.5%	96.7%	78.7%	79.5%
Group 3	93.8%	98.4%	71.5%	81.6%
Average	86.2%	97.6%	75.1%	80.6%

Table 2.2: Keypoint extraction frameworks average accuracies
source: (Jo & Kim, 2022)

In contrast to Jo and Kim's (2022) findings, the TensorFlow documentation claims that MoveNet outperformed PoseNet on several of the datasets evaluated, particularly those including "fitness action images". At the same time, they claim that MoveNet Lightning is faster but less precise, whilst MoveNet Thunder is more accurate but takes longer to extract keypoints, which complies to the results by Jo & Kim (*pose estimation — tensorflow lite*, 2022).

2.5 Similar Research Using AI for Posture Detection

This section will focus on previous research, as well as delving into the findings and conclusions of other researchers as background information for this paper.

2.5.1 Sensor Based vs Camera Based Pose Detection

Henry Griffith et al., (2017), employed an ultrasonic echolocation sensor to classify office activities. In this study, SVM classifiers were trained using features derived from a time-series array outputted from sensor data. The goal was to create a model that could recognize numerous activities, such as typing and writing, while also having a classifier that could distinguish between seated and non-seated activities. The latter is more essential for the scope of this thesis, since it is making a binary decision, similar to deciding between good and bad standing posture. The resultant accuracy for these types of activities exceeded 85%.

In contrast, Ghazal and Khan (2018) attempted to obtain the same type of classification, that is, a classification between sitting and standing positions, by using a different approach. The approach taken was camera based by extracting skeletal pose information from 2D images using the OpenPose framework. The researchers utilised a rule-based technique to distinguish the two positions, such as the angle between the hips and knees and the distance between them. By using these features, they managed to classify between the two postures at an average accuracy of 95%.

As observed in Ghazal & Khan (2018), one can note that a simple webcam camera, which can be easily obtained from many locations, can perform skeletal point extraction and classify diverse postures from a live or recorded feed while giving high accuracy results.

2.5.2 Feature Engineering in Machine Learning (ML)

Feature engineering is the process of calculating additional data that the model was unable to construct on its own and that is dependent on the values of other features. Mathematical transformations, ratios, and differences are commonly used to construct these properties (Verdonck et al., 2021). The distances between keypoints, or the angle between them, are an example of this in pose estimation, as these tend to provide essential information in pose estimation. Depending on the mathematical capability of the ML models, these additional fields of data can improve the model's accuracy. In fact, when Heaton (2016) experimented with this by selecting a set of engineered features and combined them with different models, this was very evident. For example, a neural network's structure made it simple for it to "add, sum, and multiply," whereas a RF model needed to build multiple branches for each combination as it cannot handle two inputs at once, rendering these properties meaningless.

Gatt et al., (2019) conducted research towards training a model to recognize abnormal human behavior. PoseNet and OpenPose were used as keypoint extraction frameworks, as well as two types of autoencoders based on Convolutional Neural Networks (CNN's) and Long Short-Term Memory (LSTM). The keypoint data was pre-processed into a 1-dimensional time-series, scaled between 0 and 1, and then down sampled. The models were trained and tested on a dataset labelled as normal and abnormal behaviour. It is worth noting that the researchers stated that their prototype was able to function without any sensors because the models were trained solely on keypoint data, without any kind of feature engineering.

Sklearn, on the other hand, provides what are known as pipelines. These are utilised so that one can combine all of the stages in the prototype into a single pipeline for pre-processing to be available at runtime and make it more convenient (Garreta et al., 2017).

Pre-processing tasks like feature engineering, such as calculating the angle between two keypoints, can thus be included in the pipeline and applied directly when demoing the model. This allows a model without sensors to work with engineered features as well.

2.5.3 Classifiers for Posture Detection

When training a model for pose estimation, it is critical to select the correct model that can mathematically manage the data and solve the current problem. Kumar et al., (2021) conducted research to see which classifier compares best for posture detection where the implementation followed the pipeline in Figure 2.6. The researchers examined how data fared after being pre-processed and applying the engineered features. Secondly, they compared the enhanced data by feeding it through Deep learning models and machine learning models. Finally, the accuracy of these models was tested by predicting postures.

The findings revealed that deep learning models consistently beat numerous machine learning models, owing to their mathematical capabilities (Kumar et al., 2021). The Multilayer Perceptron (MLP), Support Vector Machine (SVM), Random Forest (RF), and KNN (K-nearest neighbour) were the best machine learning models for pose estimation, with accuracies of 0.81, 0.76, and 0.71, respectively.

These results made sense since the MLP classifier is the most complex algorithm when compared to the others. Due to the fact that the researchers used feature engineering which requires more computational capabilities, achieving the highest result proved that this is indeed true. Both the SVM and the RF were tuned to ignore noise and overfitting, so they can cope well with an unbalanced dataset while delivering results that are typically faster (Boateng et al., 2020).

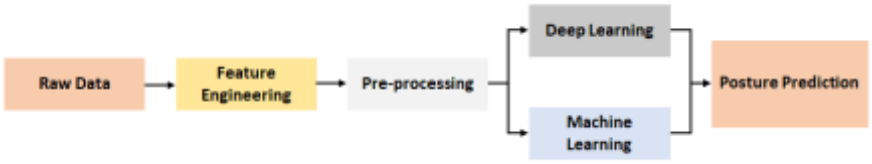

Figure 2.6: Pipeline to evaluate best classifier
source: (Kumar et al., 2021)

2.5.4 Correcting Postures with System Recommendations

Given that it can be computed from a webcam, having a pose estimation model recognize the difference between good and bad postures is already a step in the right direction. Chen and Yang (2020), conducted research to develop a model for correcting various exercise postures. The researchers did this by finding the most essential aspects of a particular workout and used those features to construct a feedback loop with the user in order to fix his posture. One of the exercises that was taken into consideration was the shoulder shrug. They discovered that if the angle between the forearm and the upper arm is too small, the range of shoulder mobility will be too low, causing the exercise to be performed incorrectly. When their pose trainer algorithm detected this, the user received a message to 'Squeeze and elevate your shoulders more during the exercise'. Raju et al., (2020), also delved into the same topic but but from a different perspective. The study's goal was to determine the difference between a good and a bad sitting posture. Instead of displaying the feedback message on screen they implemented a method to notify the user via a voice alert.

The implementation should be chosen based on the environment in which it will be used. For instance, in an office environment a voice alert might be too intrusive and, in most situations, almost pointless because the user is primarily focused on a display. On the other hand, a voice alert might make more sense in a gym environment since individuals are not focused on a monitor.

2.5.5 Optimal Camera Angle for Human Pose Estimation

One of the toughest challenges in human pose estimation is to find the correct view in which the camera or webcam will be at its most optimal position to gather the correct information for the model to make the best prediction possible. To address these challenges, Kwon et al., (2020) conducted a study to determine the ideal camera point selection based on the preferred perspective of a 3-D human stance. Three metrics were chosen to determine which camera angle would be the most appropriate as seen in Figure 2.7. To sum the three metrics, the researchers devised a viewpoint optimization problem, through which they proposed a method for view selection based on the skeleton model in frame. This demonstrated that by combining the three criteria, they were able to come up with a method for determining the ideal angle based on the individual's perspective.

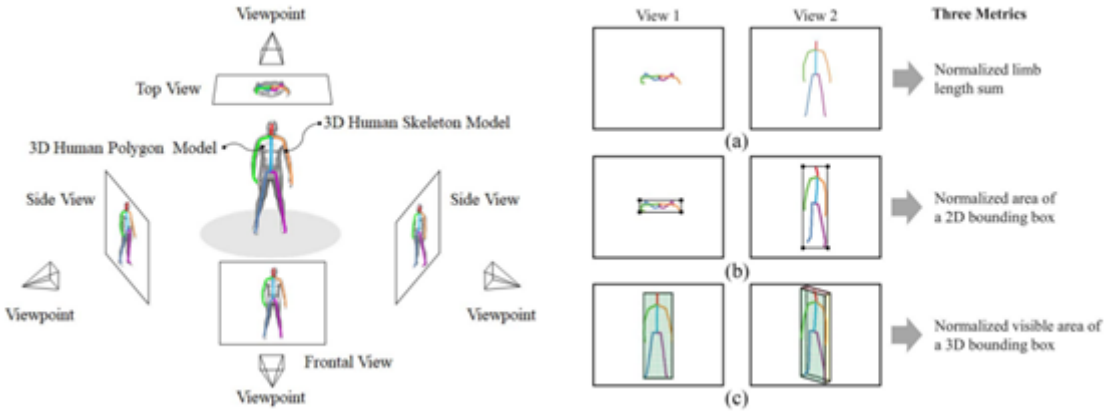

Figure 2.7: Metrics to evaluate optimal camera angle
source: (Kwon et al., 2020)

Similarly, Chang & Chen (2004), explored how pose estimation when using a multiple camera system stays consistent relatively with the known geometry. They advocated treating the multiple camera configuration as if it were a single camera and finding the best-fitting regression line that satisfied the average of all three cameras. Three images were captured at the same time, each from a pre-calibrated system that assigns each pixel to a ray with respect to the setup co-ordinates. The true ratio of the human model

had a maximum error from the estimated ratio of 0.14 and a minimum error of 0.014, according to their findings. This demonstrated that combining data from three separate cameras and using their suggested technique to calculate an average fit to create a single image produces very accurate results.

2.6 Conclusion

Since studies on posture detection for standing desks are scarce at this point in time, this literature focused on classification of standing postures in order to assess the numerous methodologies utilised over the years. In addition, various pre-processing approaches and architectures were investigated. The Multilayer Perceptron (MLP), Support Vector Machine (SVM), and Random Forest (RF) classifiers were found to be the best performing Machine Learning (ML) algorithms for pose estimation in the literature by Boateng (2020). Finally, research was conducted on different camera angles and approaches for determining the best angle for pose estimation. This literature provided the researcher with enough information to create a workable prototype for posture classification.

Chapter 3

Research Methodology

3.1 Problem Definition

The MoveNet lightning pre-trained model was implemented in this research to extract body keypoints at run-time using a supervised learning approach. This data was gathered from a regular mobile camera video stream to detect good and bad postures when using a standing desk. A custom dataset was constructed, consisting of eight videos from eight different participants. Unlike previous research, this system was designed to work without the need of any sensors, and in its current prototype version, it will only work for one person in the frame. The ankles, knees, hips, wrists, elbows, shoulders, ears, eyes, and nose were among the body keypoints extracted in this study. These keypoints and engineered features, such as angles between extracted keypoints, have to be utilised because they all contribute to standing posture detection. The models were entirely trained using these keypoints and engineered features. Given that only the body joints are retrieved from the entire video collection, there is no opportunity for error when training the models, as no additional features that are not relevant for training the models will be taken into account.

3.2 Research Questions

The main goal of this research is to attempt the detection of a good and bad posture when using a standing desk, by implementing different machine learning and computer vision techniques. The following research questions were considered in order to solve the problem:

- Which machine learning classifier responds best when creating a model to distinguish between a good and bad standing posture?
- Would feature engineering help increase the accuracy of the models?
- Which is the best point of view dataset that would achieve the highest results?

These questions were addressed by developing a variety of model combinations that fit the study's objective whilst adhering to the research topics.

3.3 Ethical Considerations

Since this research utilised a custom dataset that was developed by filming participants, all participants were given a consent form outlining how their data would be used in the study, as required by the ethical criteria. The findings of this study were derived entirely from experiments that were subjected to quantitative analysis.

3.4 System & Development Infrastructure

The development of this prototype was conducted on a desktop computer utilising a Windows 11, 64-bit operating system with the following list of specifications: NVIDIA GPU 3080 12GB VRAM, 16GB RAM and an AMD Ryzen 7 5800x 8 core. The back angle dataset was captured with an IPhone 11 while the side dataset was captured with

a Redmi Note 9a, both recording video at 30 frames per second. WSL was used to construct a docker environment on the Visual Studio Code IDE using Python 3.8 for both the data labeler and the demo. The models and dataset pre-processing were all programmed using google colab. The libraries that were used for development were: Open-CV, NumPy, Pandas, TensorFlow, Scikit-learn and PySimpleGUI.

3.5 Prototype Pipeline

The pipeline shown in Figure 3.1 is a step-by-step guide of the workflow that was undertook in this study. These are the steps taken from start to finish to develop the prototype. In the upcoming sections these steps will be laid out in more detail.

1. Acquiring the data from two perspectives (side and back).
2. Labelling the data in good and bad postures.
3. Creating three datasets (side view, back view and combined view).
4. Pre-processing the datasets such as normalization.
5. Applying the engineered feature to the datasets.
6. Developing three other datasets with these engineered features.
7. Use these datasets to train the different classifiers.
8. Fitting the models into Scikit learn pipelines.
9. Run the prototype user interface.
10. Load the trained pipelines in the prototype.
11. Input the video that needs to be classified.
12. Achieve good or bad posture classification.

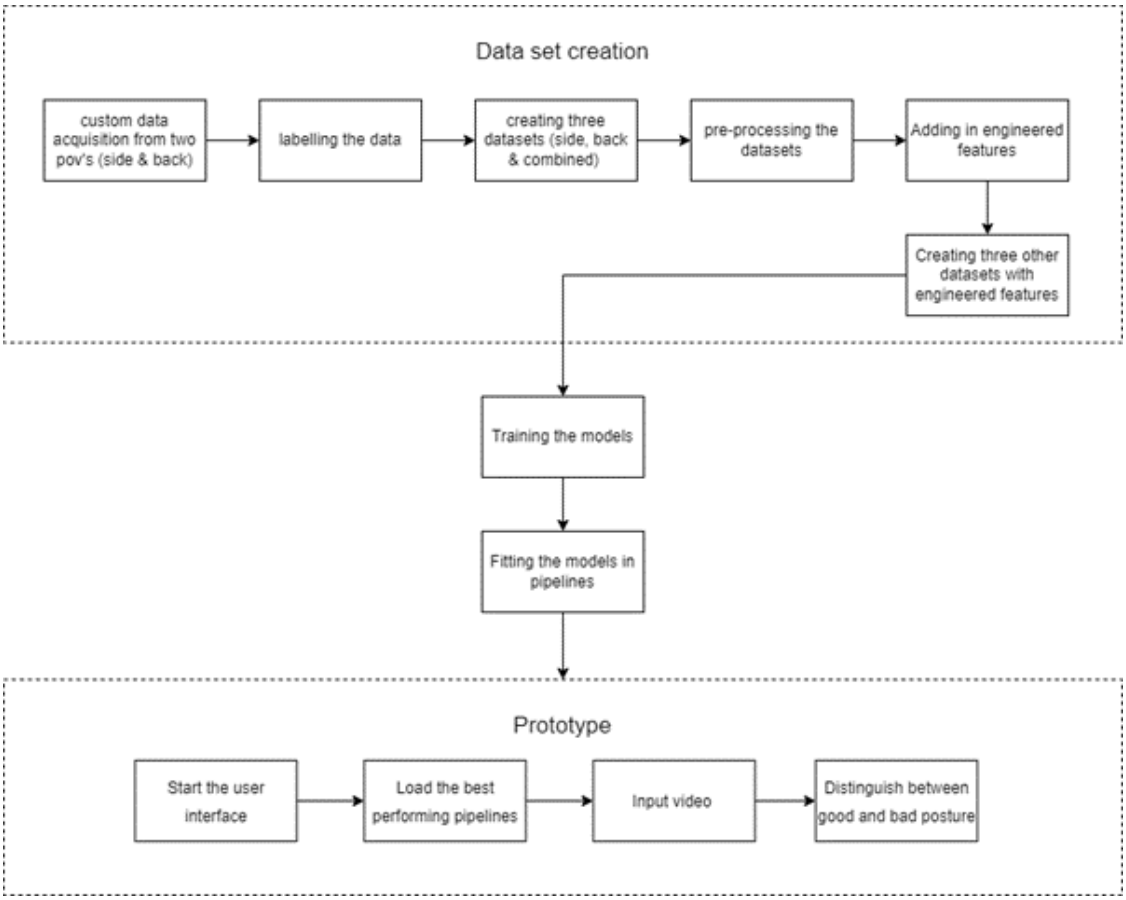

Figure 3.1: Prototype pipeline

3.6 Custom Dataset Creation

The problem that is being addressed in this research was rather specific, since posture detection in standing desks is a relatively new technology that has not yet been well explored due to its recent arrival on the market. As a result, since a ready-made dataset for research purposes could not be discovered online, the creation of a personalized dataset suited to the demands of this study had to be implemented as the first stage in the pipeline of this prototype. The data collection was collected from eight, forty-second videos, each containing eight different participants, that were situated in a setup to replicate a standing desk postured environment. All participants were instructed to switch between good and bad postures at various intervals throughout the video, based

on their understanding of what constituted as a good and bad posture. Every video was shot from two perspectives: a back angle and a side angle view as seen in Figure 3.2. In turn, this enabled the creation of three different datasets. The side view dataset, the back view dataset and the the combined view dataset, which is a combination of the the other two views in one dataset.

Figure 3.2: Dataset acquisition

3.7 Developing the Dataset Labeler

The open-cv library was used to create a data labeler tool that allowed to press the 'w' and 's' keys to label each frame as good or bad postures, respectively. This resulted in a CSV file containing the frame number, the x, y and z co-ordinates of each keypoint, as well as the posture label. This was done for each video to generate the datasets that were used to train the models.

3.7.1 Keypoint & Frame Extraction

The MoveNet model, specifically the Lightning framework, was used to extract the body keypoints, which is faster at detecting keypoints but loses some accuracy in the process. The model, which operates at 30 frames per second, retrieves 17 keypoints from the human body. This model extracts keypoints in a 3D matrix outputting the x, y and z co-ordinate of every keypoint. The video was split into frames and the key points were shown on each frame using the open-cv library. This is a crucial step as the labeler is intended to take every frame as its input.

3.7.2 Labelling the Data

When labeling the acquired data, as described in the literature (Section 2.3.2) of this research, a set of rules were strictly followed. These principles included appropriate standing desk posture ergonomics, which helped the researcher choose whether frames should be labeled as good or bad standing postures throughout the labeling process. Both video angles (back view and side view) were labelled simultaneously for every participant, since they were recorded at the same time and every frame matched in both angles. To speed up the labeling process, instead of labeling every frame, which would be time consuming, a method was devised to process the first frame every six frames and label that particular frame. Only 200 frames per video were manually tagged using this method. After this procedure was completed, a method was implemented to go back and search the empty labels in the list (the five frames that were still not tagged) and tag them with the same label that had been assigned to the previous labelled frame.

3.8 Dataset Pre-processing

3.8.1 Splitting the Data Frame Depending on Dataset

The frame count, the x, y & z co-ordinates of 17 body keypoints, the label, and the fileId were all part of the 105-column data frame. To train the models on the three datasets (back, side, and combined), the keypoints column names needed to be labeled in a way that made it straightforward to extract data based on the first word. A series of nested loops were used to label all of the data in this way (Figure 3.3). This enabled the models to be trained on a dataset based on the first word of the column names. For instance, if a model was to be trained on the back dataset, any columns that did not start with the naming convention 'back_' were dropped from the dataset, leaving only the back view data in the data frame. It is worth noting that the frame count and fileId columns were left out of the training set because they do not contribute to any learning improvements.

back_right_eye_z	...	side_right_knee_y
0.511264	...	0.507407

Figure 3.3: Dataset split according to dataset-view

3.8.2 Splitting Data Frame in Sets

The data frame was split into a training set and a test set using the train_test_split() method supplied by the Sklearn package, with a ratio of 80% to 20% respectively. From the training set another 20% was taken to create a validation set. This meant that the test split consisted of 2,400 frames out of a total of 12,000 frames, whilst the remaining 9,600 frames, were divided into 7,680 frames for training and 1,920 for the validation split.

To split the data randomly, the parameter 'shuffle' for the train and test splits was left in its default value of 'True'. This prevented the sets from containing labels from only one class, such as the train set having all the bad posture frames and the test set having all the good posture frames.

3.8.3 Normalizing the Data

This data was normalized using a standard scaler, which meant that all of the values were distributed with a mean of 0 and a standard deviation of 1. Although some of the frames contained similar information due to the small change in motion from a 30-fps camera, it was decided not to perform down sampling. This decision was made based upon the fact that it would drastically reduce the training dataset size, which would deem to be problematic to train the models with such a small dataset.

3.9 Feature Engineering

This study used Feature Engineering (FE) to analyse if adding a new variable to the dataset will increase the accuracy of the different models. This variable is the angle between two human body keypoints. Six points where specifically chosen, as these represent three major skeletal structure regions, which define a person's posture. The points being; shoulder and hip (representing the back), elbows and wrists (representing the arms) and the eyes and nose (representing a persons line of sight). These points were chosen based on an online guide by Burr (2021), which was focused on improving an individuals postures when using a standing desk. This article was reviewed by an ergonomics doctor, Dr. Bridger.

3.9.1 Angle Between Two Keypoints

The angle between two key points was achieved by using this formula:

$$math.atan2(y2[0] - y1[1], x2[0] - x1[1]) * 180/math.pi \tag{3.1}$$

The angle theta is calculated using the x and y coordinates of two keypoints. Although MoveNet generates a three-dimensional matrix for each keypoint, the z co-ordinate was ignored because the participants' movement toward the camera was negligible. The angle difference between the right hip and right shoulder, as well as the angle difference between the left hip and left shoulder, were used to compute the back. These four essential keypoints encompass the back of the body, which is very important to take into consideration when judging a posture since a slouched position indicates poor posture (Burr, 2021). The arm angle, specifically the ulna bone, was computed by calculating the angle of the wrist from the elbow keypoint, which was done for both hands. This is a crucial factor, since in a standing desk environment, the elbow is at an optimal position when it is at 180 degrees to the wrist (Burr, 2021). Finally, the angle of the nose from the eyes was used to compute the facial position, more accurately the angle of sight. This is a crucial angle in standing desk ergonomics, as in order to establish appropriate posture, the user should be facing the monitor at eye level rather than at an angle (Burr, 2021). All of these variables which can be viewed in Figure 3.4, were added to the dataset to calculate the angles at different frames. The dataset was divided into six data frames using pandas: one having the back features and back engineered features, another containing the side features and side engineered features, and a combined data frame comprising all of the previously stated data and their engineered features.

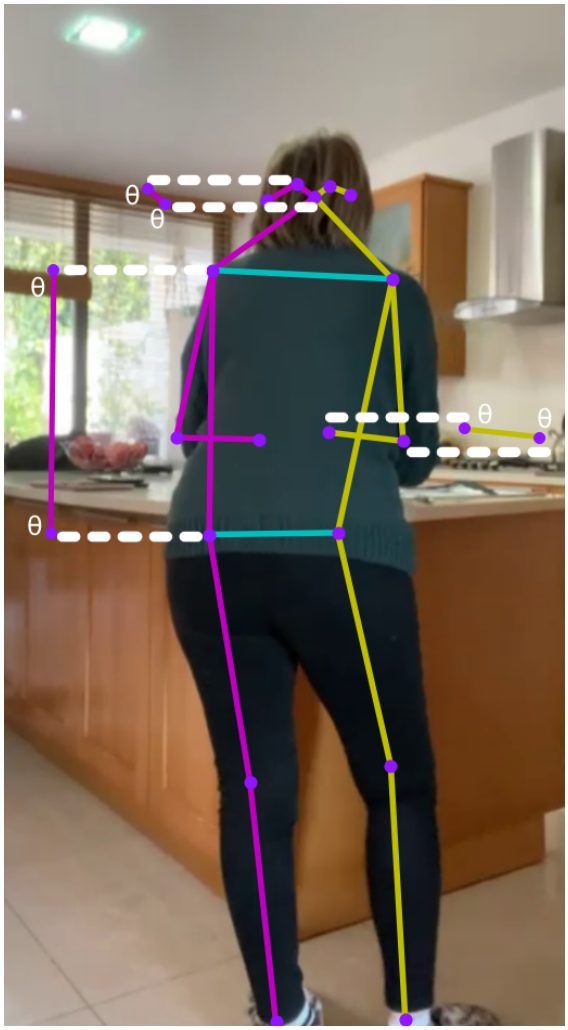

Figure 3.4: keypoints used for feature engineering

3.10 Pipeline Implementation

Pipelines were implemented mainly for the fact that the Engineered Feature (EF) needed to be extracted at runtime from a video or stream. Apart from that, it enhanced the efficiency of fitting and predicting a model with a single pipeline call and made the workflow of the preprocessing steps easier to grasp. The Scikit-learn Pipelines enabled this because they allowed for the construction of specialized transformers. These transformers are essentially pipeline steps that were run in a sequential order, that implemented a fit and a transform method. Finally, to train the models, the pipeline included a fit method. A set of custom and standard transformers were used to build the pipeline for all of the models.

Custom Transformers:

1. FeatureEngineeringTransformer(): The feature engineering tasks indicated in Section 3.9.1 were handled by this transformer. The fit method was not used in this procedure as it was not required to fit anything. On all the dataset views (back, side & combined), the transform method was utilised to calculate the angles of the provided keypoint used for feature engineering.

2. FeatureSelectionTransformer(): This transformer was created to manage which columns should be removed from the datasets prior to training the models as stated in Section 3.8.1. In the pipeline, this transformer simply used the transform function.

Standard Transformer:

1. StandardScaler(): Before training the models, this stage in the pipeline normalized the data. This was a standard transformer provided by Scikit, and it performed the preprocessing step stated in Section 3.8.3.

3.11 Training the Models

Support Vector Machine (SVM) classifier, Random Forest (RF) classifier, Multilayer Perceptron (MLP) classifier, and a Logistic Regressor (LR) were the four models chosen to be trained in this study based on their performance in pose estimation in previous studies (Boateng et al., 2020). Six datasets were constructed, as shown in Table 3.1. The first three include the data frames with the body keypoints and labels, whereas the last three included an extra column with the data frame's engineered feature. They were all used to train the four separate models stated in the beginning of this section, resulting in a total of 24 pipelines. Prior to fitting the models, a grid search was used to optimize them, in which the model tries to locate the optimal combinations of hyperparamters that would increase the estimators' performance the most. This was used on all four classifiers to get the best results from each, while still producing an unbiased outcome.

Models	df_back	df_side	df_combined	df_back_EF	df_side_EF	df_back_EF
SVM	x	x	x	x	x	x
LR	x	x	x	x	x	x
MLP	x	x	x	x	x	x
RF	x	x	x	x	x	x

Table 3.1: Models against Datasets

3.11.1 MLP classifier

The GridSearchCV() method from Scikit Learn was used to discover the best match of this model's input hyper parameters. The parameters utilised in the grid search are all listed in Section 2.2.4. The values for the 'batch_size' option were set to '8', '16', '32' and '64'. The values for the 'hidden_layer_sizes' parameter were set to '[(50,), (100,), (200,)]'. The 'early_stopping' parameter was set to 'true' to avoid overfitting.

It is worth noting that, the Scikit learn MLP classifier contains an ‘alpha’ option that was left at its default value; this is intended to minimize overfitting by decreasing the model parameters, and the ‘shuffle’ parameter was set to true to manage sample randomization throughout each iteration. The value of ‘max_iter’ was set to ‘500’.

3.11.2 SVM classifier

In Chapter 2 Section 2.2.2, the hyper-parameters that were used for the grid search for the SVM classifier were explained. The ‘C’ parameter values were set to ‘0.1’ and ‘1’. ‘Poly’ and ‘linear’ were chosen as the ‘Kernel’ parameters. Finally, the ‘gamma’ parameters were then set to ‘scale’ and ‘auto’. It is worth noting that the RBF kernel was not used as it known for overfitting a model when using small datasets. Other options such as ‘probability’, ‘shrinking’ and ‘max_iter’ were all kept at their default values.

3.11.3 RF classifier

The hyper-parameters of this classifier were all explained in depth in Chapter 2 Section 2.2.3. To identify the optimal combination of parameters, these parameter values were all run through a grid search. The first parameter ‘n_estimators’ was set to ‘120’, and ‘140’. The values for the second parameter ‘min_samples_split’ were set to ‘3’, ‘4’ and ‘5’. Finally, the ‘max_depth’ parameters were set to ‘3’ and ‘4’. Instead of leaving it at default, the ‘max_depth’ was specified to reduce overfitting in this model. This was done due to the fact that the dataset is not large enough, which meant that, if it would have been kept at default value the trees would kept expanding until the node is pure. This reduced the complexity of the trees which is beneficial in situations were training is done with a small dataset.

3.11.4 LR classifier

The parameters that were passed through the logistic regressor and used in the grid search were explained in detail in Chapter 2 Section 2.2.5. The following is a list of values that these parameters were set to. The 'C' parameter was given values of '0.1', '1' and '10'. The 'penalty' parameter was given the values 'none' and 'l2'. The rest of the hyper-parameters such as 'max_iter' and 'solver' were left in their default state, which was '100' and 'lbfgs' respectively.

3.12 Additional Test cases

3.12.1 Test Case 1: Testing the Models Against an External Dataset

The models were put to the test against a dataset recorded in a different context than the initial training and test set. The goal of this experiment was to expose how generalisable the model was in a different context. In this video, the distance between the camera and the participant differs slightly from the one indicated in Section 3.6. Two additional videos were recorded from a side view and back view angle but the cameras where placed closer to the participant to see how the models would behave. The video was then run through the data labeller described in Section 3.7 and categorised into good and bad postures using the outlined principles. Finally, the pipelines were loaded, and the accuracy as well as the F1-Score were calculated using the Scikit package 'Accuracy_score' and 'f1_score' respectively.

3.12.2 Test Case 2: Testing the Models Against an Obstructed View

The obstructed view test was another test used to evaluate the models generalisability. The dataset used in test case 1 (see Section 3.12.1) was re-used in this test. However, it was altered to include a 2D box that covered the area of the hips to the knees, as shown in Figure 3.5. This was performed to simulate an obstacle and it was maintained throughout the whole video to test the models' limits. After the video was edited, it was run through the data labeler as described in Section 3.7, and the positions were labelled as correct and incorrect depending on the posture of the shoulder, neck, and hands. The labeling was done in accordance with the guidelines laid out in Section 2.3.2. This enabled for the evaluation of accuracy, F1-Score, and a confusion matrices on a new obstructed dataset using the pipelines mentioned in Section 3.5.

Figure 3.5: External dataset with obstruction

3.13 Conclusion

The MLP classifier, SVM classifier, RF classifier, and LR were the classifiers that were chosen to be trained in this study. The MoveNet lightning framework was used to retrieve body keypoints data to train these algorithms from a custom datatset. Aside from that, two other datasets were also developed to evaluate the model's ability to generalise in a variety of scenarios. This led into the identification of the models' advantages and disadvantages, which will later be discussed in detail in Chapter 4.

Chapter 4

Analysis of Results and Discussion

4.1 Introduction

This chapter will explain the findings that were obtained following the implementation stated in Chapter 3. This will also allow the following research questions to be answered:

- Which is the best point of view dataset that would achieve the highest results?
- Would feature engineering help increase the accuracy of the models?
- Which machine learning classifier responds best when creating a model to distinguish between a good and bad standing posture?

The following section will be broken down into three primary parts. The first section will be an evaluation of metrics such as accuracy, precision, recall, and F1-Score in order to answer the research questions. These will be compared to the performance of the different classifiers indicated in Section 3.11, the three datasets which were mentioned in Section 3.6 and the EF that was explained in Section 3.9.1. All of these findings were based on a custom dataset that was divided into three splits: training split, validation split, and test split. The second section will focus on how the best models fared on a

dataset in a completely different environment in order to eliminate any bias and validate the data gathered from the first section. This was further addressed in Section 3.12.1. The third portion is a test case described in Section 3.12.2, in which the previously mentioned external dataset was altered to include a 2D box covering certain keypoints of the individual in the frame to mimic an obstruction. Moreover, a naming convention for the model names will be noted in this chapter multiple times. An instance of this convention is 'combined_engFalse_MLP'. The first part entails which dataset (side, back & combined) was used. The second part shows if the EF was included or not in the dataset (engFalse meaning that it was not included). Finally, the last part specifies which classifier (MLP, SVM, RF and LR) has been used.

4.2 Best Performing Dataset

When the back and side view datasets were compared using the four distinct classifiers, a pattern emerged. The side datasets outperformed the back datasets in every classifier, with the top performing one having a validation set accuracy of 97.8%. Moreover, the achieved test set accuracy of this model was 98% with an F1-Score of 98% when using the MLP classifier. The best performing back dataset accuracy was also achieved by the latter, resulting in 97.6% accuracy on the test set with an F1-score of 97.6%. Other classifiers, such as the RF classifier, showed a greater distinction between the two. The latter resulted in a 2.6% difference in accuracy between the best side and best back classifiers, with the side dataset proving to be the better of the two.

When testing out the models in the prototype to visualise how they would function in a video, it was discovered that in certain instances the keypoint estimation was highly inaccurate. This occurred when the participants had an arm hidden in front of their torso when considering the back view models. As a result, the keypoint extraction framework

MoveNet tried to guess where the participants' hands were at that particular instance. In some frames the model failed to assume the correct posture of the hands therefore it made an incorrect assumption as seen in Figure 4.1. Since the MoveNet model could observe all of the participants' body joints when using the side view, it made a more informed assumption on where the body joints are on the opposing side, as shown in Figure 4.2. One cannot simply draw a reliable conclusion since the keypoint extraction from the back view dataset was erroneous in some cases. As a result, the following is an informed assumption as to why the side view dataset outperformed the back view dataset:

- Since the back view dataset had a certain limitation of assuming an incorrect keypoint location at certain frames (as previously explained), this might have hindered its performance when evaluated against the test set, thus producing poorer results in comparison to the side view dataset.

The final dataset was constructed with a concatenation of the side view and back view datasets, resulting in the combined view dataset mentioned in Section 3.6. This dataset outperformed the stand-alone datasets in 23 out of 24 models, probably due to its complexity, as the ratio of the number of columns (keypoints data) to rows (frames) was twofold. In fact, the best performing models overall were all using the combined dataset, except for one case in the RF classifier, were the side view dataset outperformed the combined view. The best MLP classifier model resulted in an accuracy of 98.5% with an F1-score of 98.5%, whereas the worst performing model using the combined dataset was the RF classifier. This resulted in an accuracy of 91.9% and an F1-Score of 92.3%. These outcomes were all predicated on the prior implementation of the EF, as these features varied these results.

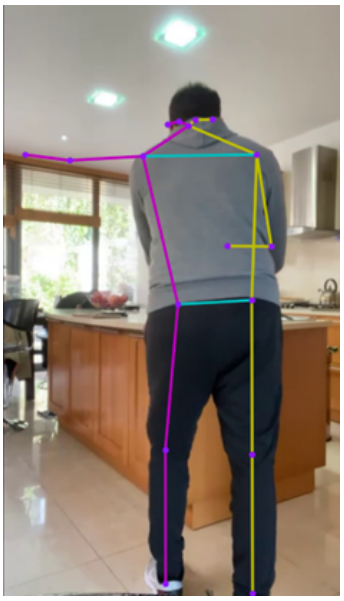

Figure 4.1: Keypoint extraction from back view

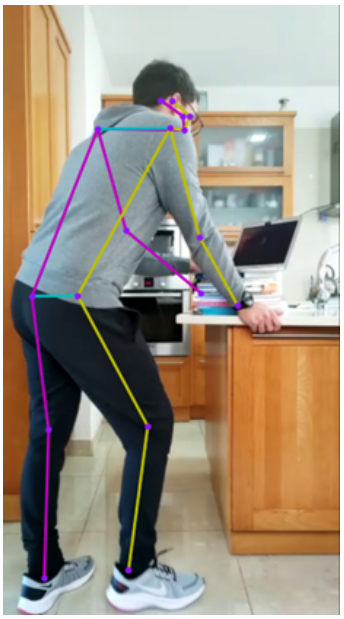

Figure 4.2: Keypoint extraction from side view

4.3 Models Performance Using the EF

To explain the results of the Engineered Feature (EF) at a high level, a Table seen in Figure 4.1, shows how the classifiers reacted before and after applying this feature. Certain classifiers such as the Multilayer Perceptron (MLP) classifier, referred to as MLP in the table, clearly revealed that all the models took a loss in terms of accuracy upon having the EF applied. In fact, when averaging out the discrepancies in the MLP classifier, it resulted in a 0.25% loss. The Random Forest (RF) classifier, on the other hand, reacted to the feature best, with an overall average gain in accuracy of 1.21%. This was followed by the Logistic Regressor (LR) with an average increase of 0.36%, and finally the Support Vector Machine (SVM) classifier with an average increase of 0.12%. When the total accuracy of all the classifiers with their respective datasets was calculated, it resulted in a 4.36% overall accuracy increase, as shown in Figure 4.2. Another pattern worth noting in these results is that when the EF was applied to the combined datasets, the best results were obtained, with the overall accuracy increasing by 3.36% whilst only losing 0.21% accuracy.

This can be explained by the fact that this dataset had two columns of EFs since it contained features from both the back and side view datasets, thus doubling in terms of rows and columns. In fact, if there had been a larger dataset, the discrepancies between the model with and without FE would probably have been even greater (Woodall et al., 2014). This is due to the fact that, when an extra column is added to the dataset, in this case the EF, the model accuracy improves when more rows are added, in this case more frames of data, increasing the gap between these models.

Aside from that, the EF was chosen after evaluating what differs a good posture from a bad one. Although this information appears to be valid at face value, it did not have a significant impact on the models. As previously stated, the problem could arise due to the models not having enough data to make certain discrepancies. When comparing the models that used and did not use the EF across the 24 trained models, it was noticeable that integrating them yielded a positive result.

Dataset	SVM	LR	RF	MLP
Back	93.38%	91.31%	91.30%	97.66%
Back_EF	93.25%	91.63%	91.63%	97.46%
Percentage Difference	-0.13%	0.32%	-0.33%	-0.20%
Side	96.83%	93.34%	90.88%	98.04%
Side_EF	97.17%	93.30%	92.46%	97.71%
Percentage Difference	0.34%	-0.04%	1.58%	-0.33%
Combined	97.41%	94.92%	91.96%	98.50%
Combined_EF	97.58%	95.73%	94.34%	98.29%
Percentage Difference	0.17%	0.81%	2.38%	-0.21%

Table 4.1: Percentage difference of models using and not using EF

negative value: Percentage loss

positive value: Percentage gain

Total difference using FE	5.6%
Total difference not using FE	-1.24%
Total Improvement	4.36%

Table 4.2: Total difference from models using EF

4.4 Classifier Performance

The MLP classifier achieved the highest accuracy on the test set and the highest F1-score, ranging from 97.4% to 98.5% accuracy, based on the collected data. This was an expected result as the mathematical capabilities of this classifier compared to the three others is greater due to its complexity (Boateng et al., 2020). Apart from that, it is worth noting that the SVM classifier performed admirably. In fact, with an accuracy of 97.5% and an F1-score of 97.6%, the top ranked model in this classifier, 'combined_engTrue_SVM', ranked higher than the lowest ranked model in the MLP classifier. The next section will provide a more extensive explanation of the model's performance, including the evaluation of a Receiver Operating Characteristic (ROC) curve and confusion matrices. The classifiers will be arranged in descending order from best to worst, depending on the accuracy performance.

4.4.1 MLP Classifier

The MLP classifier managed to achieve a low outcome of False Positives (FP) and False Negatives (FN), resulting in highly accurate models. When analysing the best performing model in this classifier, 'combined_engFalse_MLP' (shown in Table 4.3) and visualising the results in a confusion matrix (shown in Figure 4.3), it can be identified that 0.62% of the 2403 frames in the test set were FN, and 0.87% were FP. This meant that the model only identified 15 frames in which a bad posture was mistaken for a good posture, and 21 frames in which a good posture was mistaken for a bad posture. As stated by Boateng et al., (2020), a more 'complex' dataset should yield better outcomes in a Neural Network model than a less intricate one.

These results suggest that this statement is somewhat correct, as the model performed better on the combined datasets, which are more 'complex' since the data has doubled

as it is provided from two camera angle perspectives. The MLP classifier, on the other hand, did not appear to respond well to the EF. This could be due to two factors:

- The classifier had already learnt enough data and the Engineered Feature (EF) did not yield any extra information to the model, that it already did not know.
- The model did not benefit from the chosen EF specified in Section 3.9.1 since it may have already been taken into account by the model.

A Receiver Operating Characteristic (ROC) curve was generated from a prediction on the test set to back up the results, as shown in Figure 4.4. Since accuracy reflects the ability to classify data at a specific threshold, it may give better results by chance due to the chosen threshold. This is solved with the Area Under Curve (AUC) of a ROC curve as it reflects the models ability to separate one class from the other with a predefined threshold. For each model, the default threshold was set to 0.5. The AUC for the 'combined_engFalse_MLP' model yielded a result of 99.7%, which was higher then the resultant accuracy, but still relatively similar. This demonstrated that the models were quite proficient in classifying data. Since this study dealt with posture classification which is detrimental to ones health, the FN rates and FP rates for detecting this posture are crucial. Due to this reason a better metric, the F1-score was calculated, which penalizes these values more than accuracy. As shown in Table 4.3, all of the computed F1-scores were nearly identical to the test accuracy, confirming the test accuracy results.

Model	Dataset	FE	Val_Accuracy	F1_Val	Test_Accuracy	F1_Test
MLP	combined	False	98.13%	98.11%	98.50%	98.50%
MLP	combined	True	98.34%	98.34%	98.29%	98.30%
MLP	side	False	97.81%	97.80%	98.04%	98.04%
MLP	side	True	97.55%	97.54%	97.71%	97.70%
MLP	back	False	97.61%	97.61%	97.67%	97.66%
MLP	back	True	97.45%	97.45%	97.46%	97.47%

Table 4.3: MLP classifier results

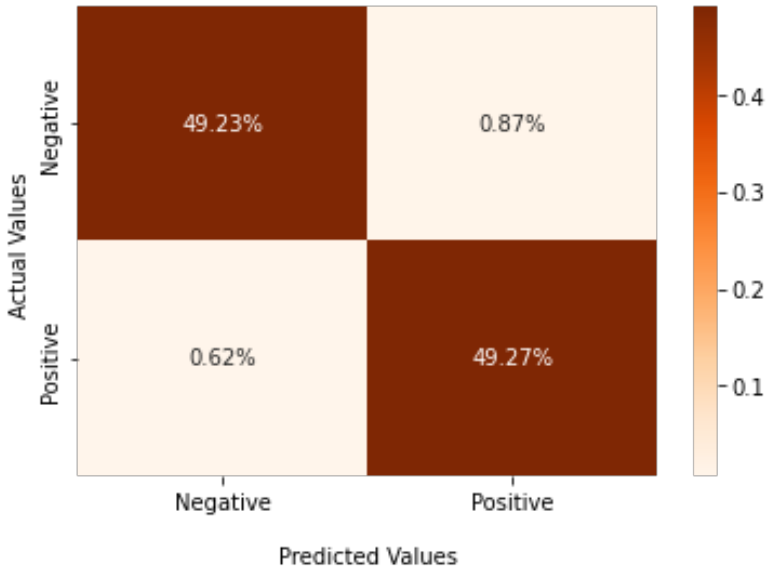

Figure 4.3: Confusion matrix of the best performing model using the MLP classifier

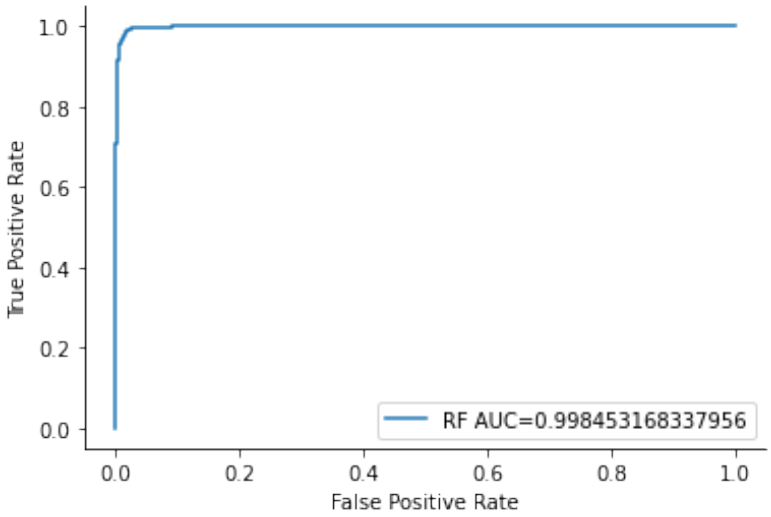

Figure 4.4: ROC curve of the best performing model using the MLP classifier

4.4.2 SVM Classifier

The SVM classifier resulted in a more dispersed manner than the MLP classifier. In fact, the accuracy variance between the best and worst performing model was of 4%. When comparing the various models that were developed with this classifier it was evident that the EF on the validation set accuracy (labeled as 'Val_Accuracy' in Table 4.4), had no significance. On the other hand, although yielding a minor percentage difference, when used on the test set, the models with the EF fared better overall. This indicated that these characteristics aided the SVM in distinguishing between the two classes. Although the variance was negligible, it is possible that the classifier needed additional data to show larger disparities or a different EF to perform better.

When examining the confusion matrix in Figure 4.5, the FP amount nearly tripled the amount in the MLP classifier, resulting in 53 frames being categorized as FP's. On the other hand, the FN rate was about three times lower than that of the MLP classifier, with 5 frames classified as FN cases. This reduced the number of FP cases, resulting in a nearly identical outcome between the two highest performing models in the SVM and MLP classifiers. When observing the confusion matrix of 'back_engTrue_SVM' in Figure 4.6, the FP's rise to 6.2%, implying that the model mistook around 149 frames in the bad posture class as good postures. This in turn reduced the True Negatives (TN)'s to 43.9%, which is an exact difference of 4% from the best model, 'combined_engTrue_SVM'. All of the models' F1-scores were close to the test accuracies, implying that the data is very well balanced when formulating a prediction.

Model	Dataset	FE	Val_Accuracy	F1_Val	Test_Accuracy	F1_Test
SVM	combined	True	97.29%	97.32%	97.59%	97.63%
SVM	combined	False	97.29%	97.33%	97.42%	97.47%
SVM	side	True	96.36%	96.38%	97.17%	97.21%
SVM	side	False	96.36%	96.39%	96.84%	96.89%
SVM	back	False	93.76%	94.02%	93.38%	93.72%
SVM	back	True	93.24%	93.53%	93.26%	93.61%

Table 4.4: SVM classifier results

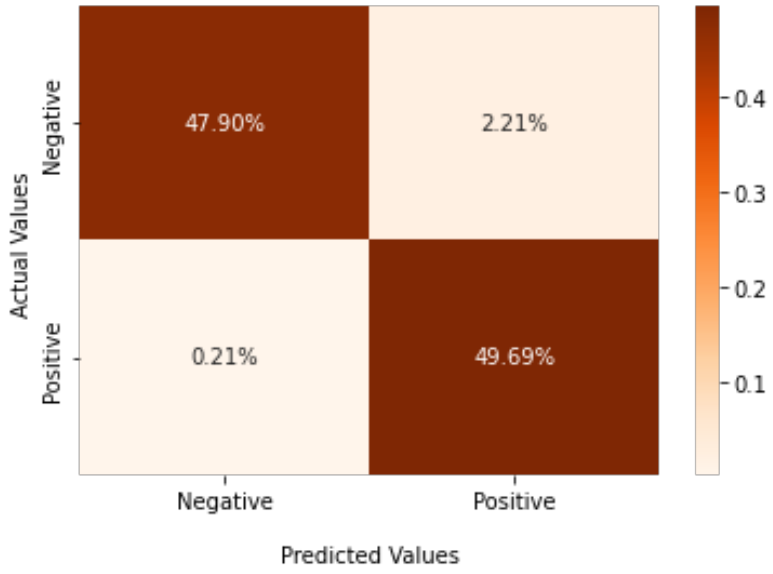

Figure 4.5: Confusion Matrix of the best performing model using the SVM classifier

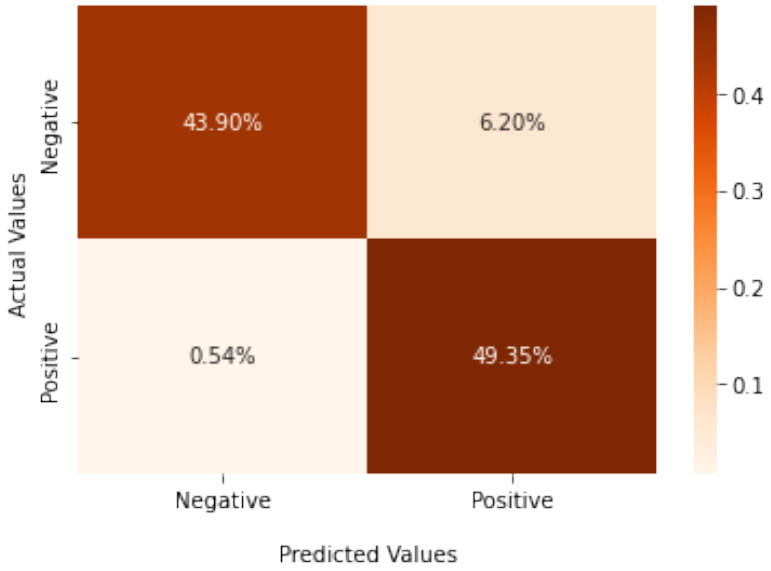

Figure 4.6: Confusion Matrix of the worst performing model using the SVM classifier

4.4.3 LR Classifier

Considering the Logistic Regressor (LR) classifier's simplicity, the results were still fairly good. In fact, the best performing model achieved 95.6% accuracy on the test set, whilst the worst performing model was 91.3% accurate. Both of the side dataset LR models were 93% accurate. When comparing the confusion matrix in Figure 4.7 to the two other classifiers (MLP & SVM), one can note that both the FP and FN rates increased. In fact, 3% of the test data consisted of 72 frames of incorrect postures categorized as correct postures and around 33 frames of correct postures classified as bad postures. As can be seen in Figure Table 4.5, upon calculating the overall difference between the models that utilised the EF and the models that did not, it showed that two out of three models performed slightly better. Figure 4.8 depicts the confusion matrix for the 'combined_engFalse_LR,' where the number of FN increased to 44 frames and the number of FP increased to 78 frames. Given the relevance of the F1-score in this study, it was calculated for this model as well as the others, and the test accuracy was found to be similar to this score.

Model	Dataset	FE	Val_Accuracy	F1_Val	Test_Accuracy	F1_Test
LR	combined	True	94.75%	94.74%	95.63%	95.70%
LR	combined	False	93.96%	93.96%	94.92%	94.98%
LR	side	False	93.29%	93.33%	93.34%	93.44%
LR	side	True	93.50%	93.56%	93.30%	93.39%
LR	back	True	91.78%	91.91%	91.64%	91.77%
LR	back	False	91.31%	91.42%	91.30%	91.43%

Table 4.5: LR classifier results

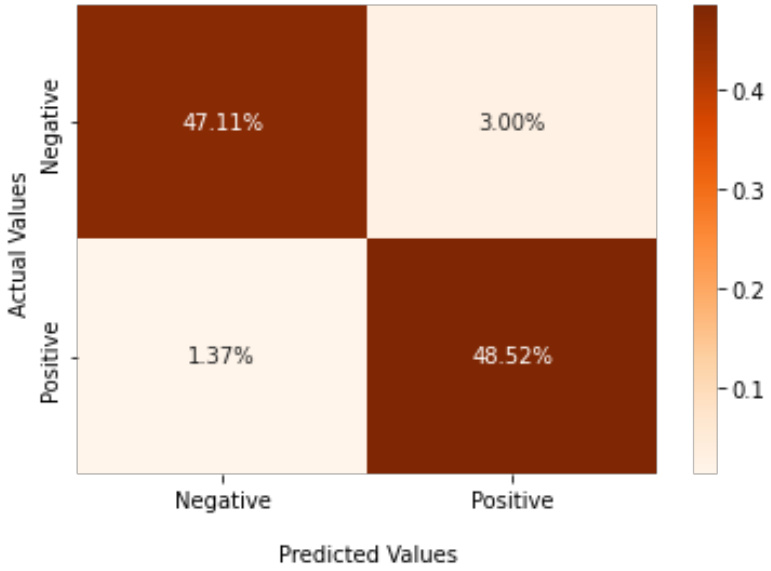

Figure 4.7: Confusion Matrix of the best performing model using the LR classifier

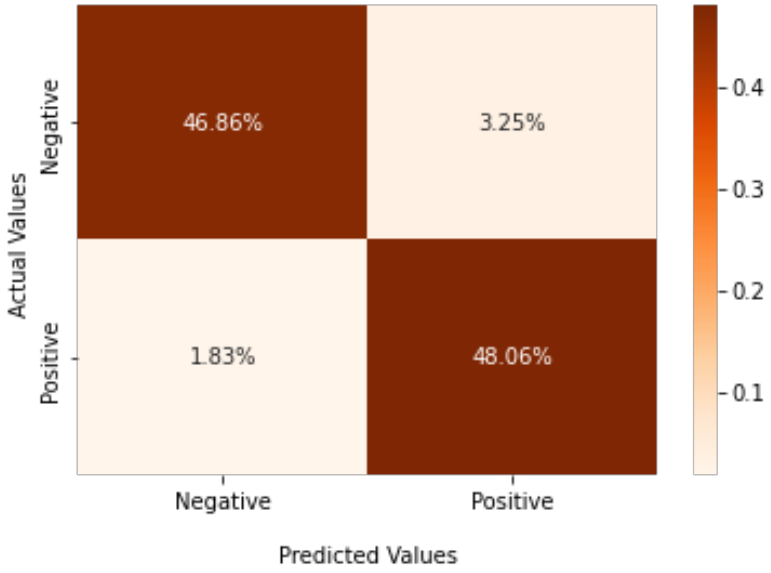

Figure 4.8: Confusion Matrix of the second best performing model using the LR classifier

4.4.4 RF Classifier

When compared to the other three classifiers, it was clear that the Random Forest (RF) classifier scored the worst. The results in Table 4.6 show that the range of accuracies between the best and worst models was roughly 6%, which is the highest variance between all the classifiers and the only one with results in the 80% accuracy range. The confusion matrix of the best model, depicted in Figure 4.9, demonstrated that this model had the highest number of FP's, with 4.37%, when compared to best ranked models from the other classifiers. This meant that 105 frames were labeled as good postures when they were in fact bad postures. The LR had a greater rate of FN's when looking at the False Negatives (FN) class. 1.29% translated to about 31 frames of incorrect predictions of frames that should have been predicted as bad postures but were instead forecasted as good postures by the algorithm. When evaluated, the confusion matrix of the worst performing model, 'back_engFalse_RF', depicted in Figure 4.10, it was clear that the number of FN's is around the same that of the best model, with 1.21% or 29 frames. The FP's rate also rose to 10.36%, or 249 frames, which is more then double that of the best performing model. These results showed a considerable difference in accuracy when compared to the other classifiers.

Model	Dataset	FE	Val_Accuracy	F1_Val	Test_Accuracy	F1_Test
Rf	combined	True	94.12%	94.27%	94.34%	94.50%
Rf	side	True	93.03%	93.25%	92.47%	92.79%
Rf	combined	False	92.35%	92.70%	91.97%	92.36%
RF	side	False	91.83%	92.22%	90.89%	91.37%
RF	back	True	88.97%	89.82%	89.85%	90.59%
RF	back	False	87.83%	88.87%	88.43%	89.38%

Table 4.6: RF classifier results

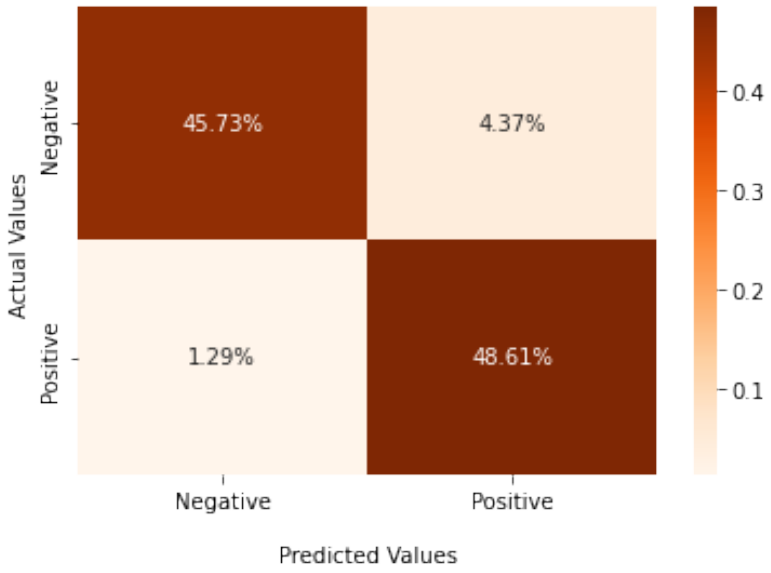

Figure 4.9: Confusion Matrix of the best performing model using the RF classifier

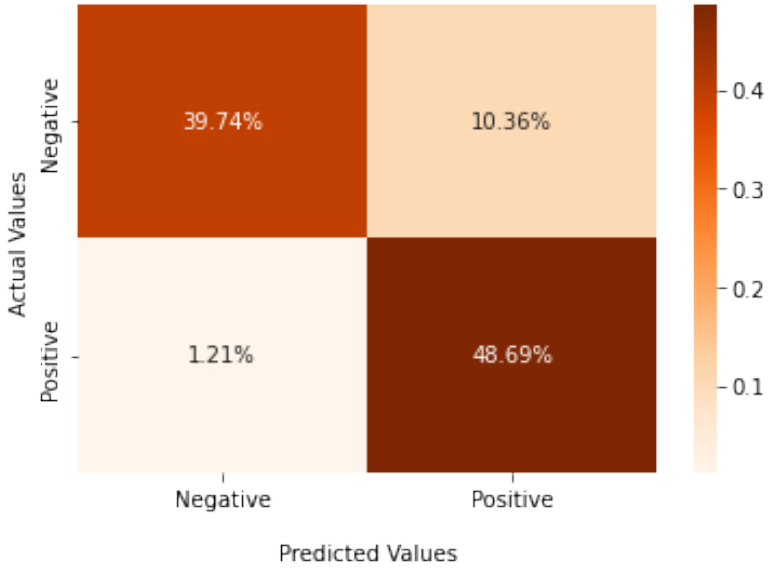

Figure 4.10: Confusion Matrix of the worst performing model using the RF classifier

4.5 Performance of Models on External Dataset

To evaluate the best performing models for each point of view dataset, the back view, side view, and combined view datasets, the models were tested against a dataset that they were never exposed to before, as detailed in Section 3.12.1. Since the MLP classifier produced the best performing models for each view, all the view dependant models using this classifier were used to evaluate their performance against this dataset. The models that were tested included two distinct training set sizes for this test instance, as shown in Table 4.7. The combined and side results on the training set size of 7,680, which is approximately the size utilised for the results reported in Section 4.4, revealed that they were similar to one another as seen in Table 4.7. They differed by roughly 3% which indicated that the initial results might have resulted in such a high accuracy as a consequence to overfitting the models. The side model indicated that it performed better in this situation then the combined model. The back model, on the other hand, differed by 46%, which showed a significant discrepancy when compared to the original result.

The training set size was reduced by two videos, resulting in 6,144 frames of training data to test if the back model was overfitted. The 'back_engFalse_MLP' model still performed poorly with this training set but the 'back_engTrue_MLP' model with the smaller data sample performed as expected at an accuracy of 96.08%. When utilising the smaller training set models to evaluate the confusion matrix of the models 'back_engFalse_MLP' and 'back_engTrue_MLP,' it was discovered that the model not utilising the EF had a very high FN rate of 45.6% as seen in Figure 4.12. The model that used the EF, on the other hand, resulted in a FP rate of 3.75%, as shown in the Figure 4.13. This might be attributed to the EF providing the model with more critical information on a video it had never been exposed to before. This also hints that the models using the back dataset were being overfitted when trained with a larger dataset size. Furthermore, when visually examining how the two models perform on the video, one can notice that the

'back_engFalse_MLP' detected a bad posture at instances where it should have detected a good one, as shown in Figure 4.11. The 'side_engTrue_MLP' model, on the other hand, performed considerably better and is consistent with the combined dataset model's judgment.

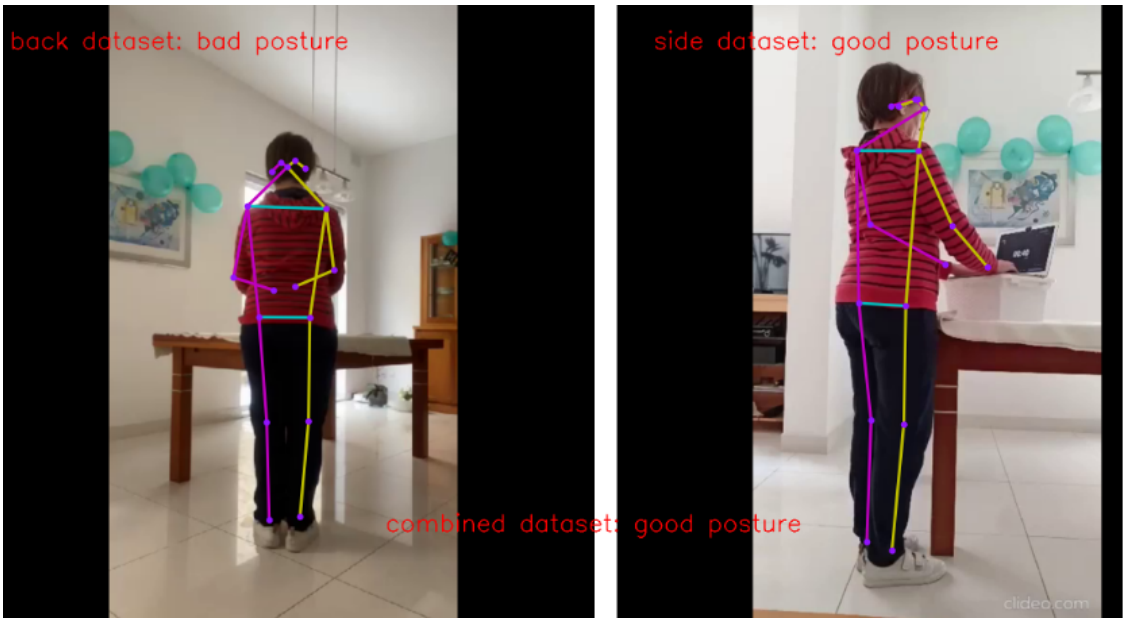

Figure 4.11: Model prediction on external dataset

Model	Training Set Size: 7680	Training Set Size: 6144
combined_engFalse_MLP	94.86%	83.27%
side_engFalse_MLP	95.10%	86.78%
back_engFalse_MLP	51.63%	54.40%
combined_engTrue_MLP	94.37%	87.43%
side_engTrue_MLP	95.00%	93.80%
back_engTrue_MLP	51.63%	96.08%

Table 4.7: MLP classifier results on external dataset

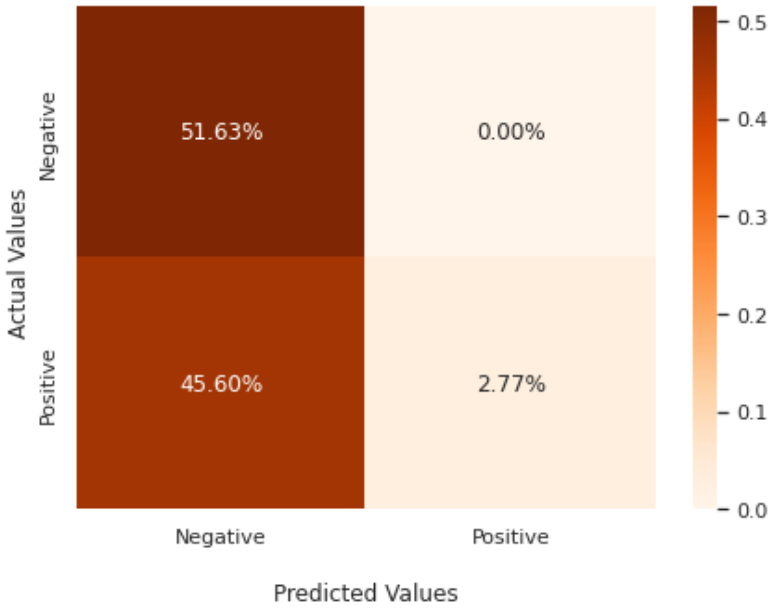

Figure 4.12: back_engFalse_MLP model with Training set size: 6144

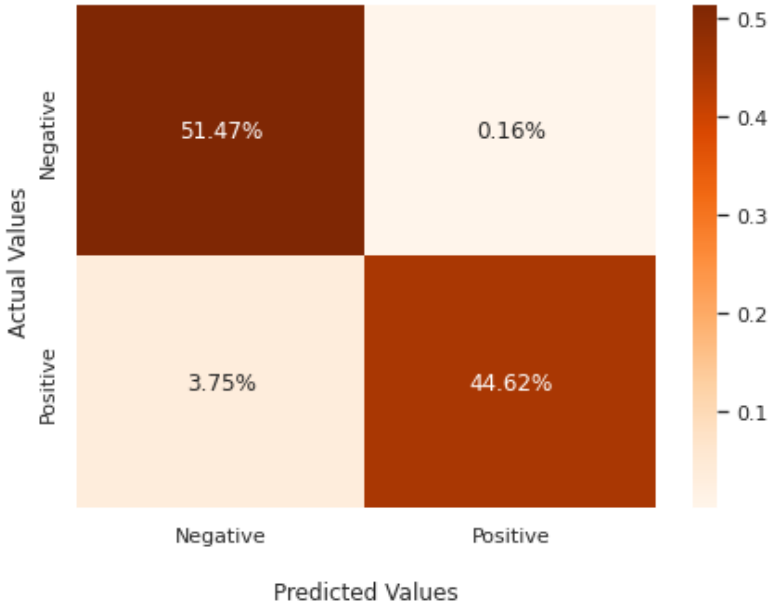

Figure 4.13: back_engTrue_MLP model with Training set size: 6144

4.6 Performance of Models on Dataset With Obstacle

A 2D square was edited throughout the video, covering the individuals hips and knees keypoints on both camera views, to test how the models would perform when there is an obstacle between the user and the camera. This was further detailed in Section 3.12.2. This was an extreme situation where the user was obstructed for the duration of the video to test the models' limits. Since the models were not trained to account for an obstacle, it was expected that their would be a drop in performance. In fact, when testing the models, the outcomes differed a lot.

As can be seen in Table 4.8, the results of the SVM, RF and the MLP classifiers all performed poorly on the obstructed dataset with an average accuracy of 54.25%, 55.12% and 52.99% respectively. The logistic regressor models, on the other hand, had an average accuracy of 72.24%, with the 'combined_engFalse_LR' model having the best accuracy at 93.06%. When the highest rated models for each view (back, side, and combined) were applied on the video stream, it was apparent that the combined and side models were able to categorize good postures whilst the back view could not, as can be seen in Figure 4.14. It is possible that the LR discovered a linear correlation between the inputs and outputs that the other models did not exploit due to their complexity. A comparison of the confusion matrices for the best models utilising the Logistic Regressor (LR) was done in Figures 4.15, 4.16 and 4.17 to better evaluate this data. To begin with, it was noted that the back view model (Figure 4.15) failed to accomplish any classification for the true positive class, implying that it was not able to make a final prediction, thus rendering dysfunctional. Alternately, the side model (Figure 4.16) showed more potential, as it correctly classified 37% true negatives and 46% true positives. The concern with this model was that it still had a high rate of false positives (16%). The combined model (Figure 4.17) once again proved to be more prominent as it generated the best results, with a true negative rate of 50% and a true positive rate of 42%.

Dataset	SVM	LR	RF	MLP
Back	52.03%	54.00%	55.47%	53.58%
Back_EF	54.32%	53.87%	54.27%	53.58%
Side	46.41%	83.44%	52.29%	47.63%
Side_EF	46.49%	79.93%	52.89%	55.87%
Combined	59.46%	93.06%	57.68%	53.58%
Combined_EF	66.80%	69.16%	58.76%	53.67%

Table 4.8: Models performance on obstructed datatset

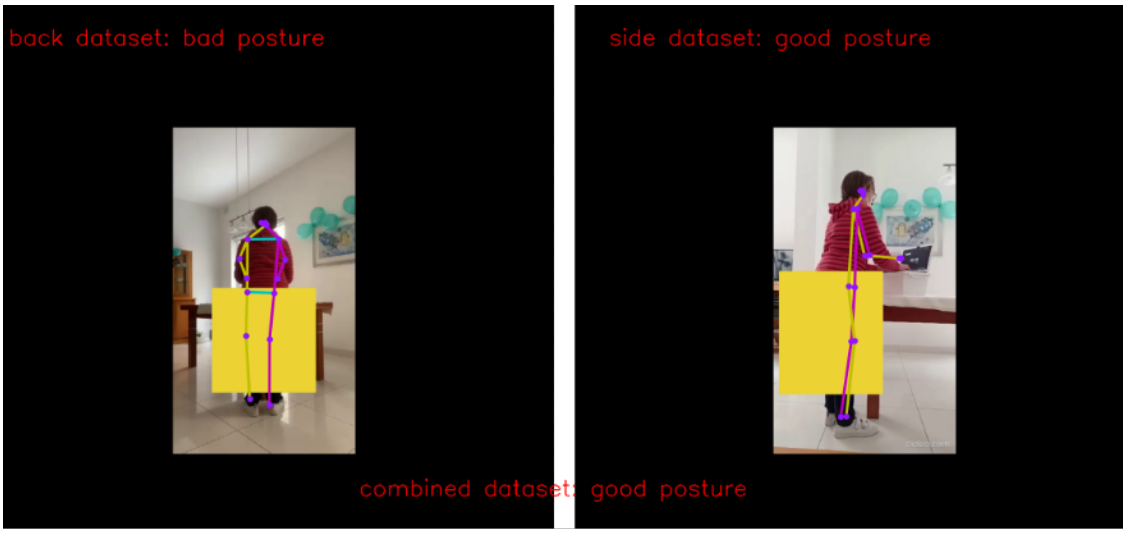

Figure 4.14: Prediction of models on obstructed dataset

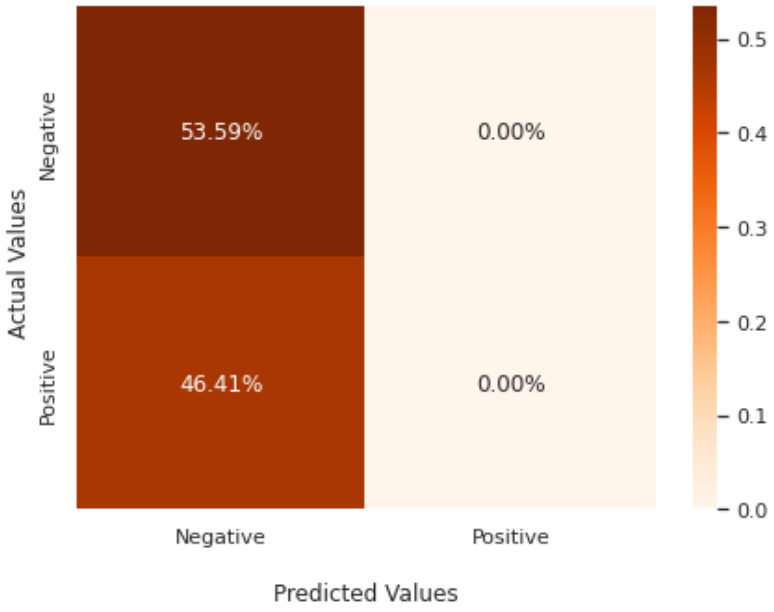

Figure 4.15: LR back obstructed dataset confusion matrix

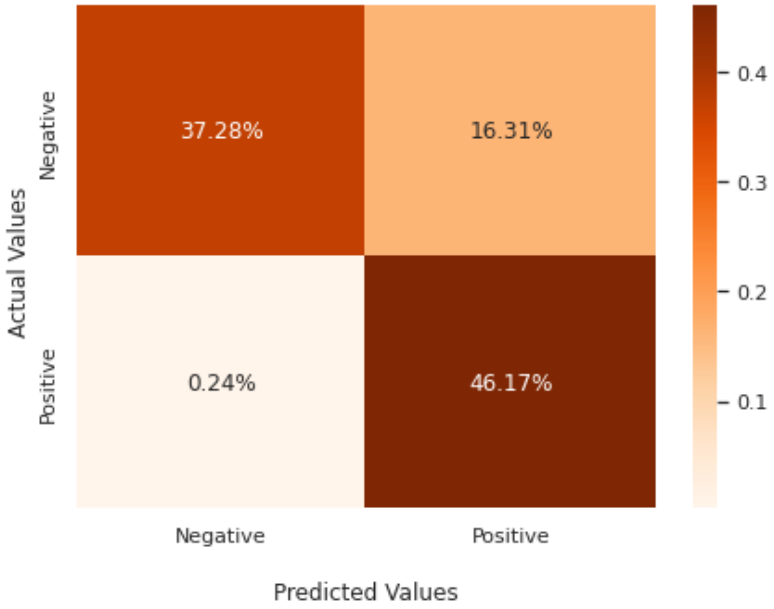

Figure 4.16: LR side obstructed dataset confusion matrix

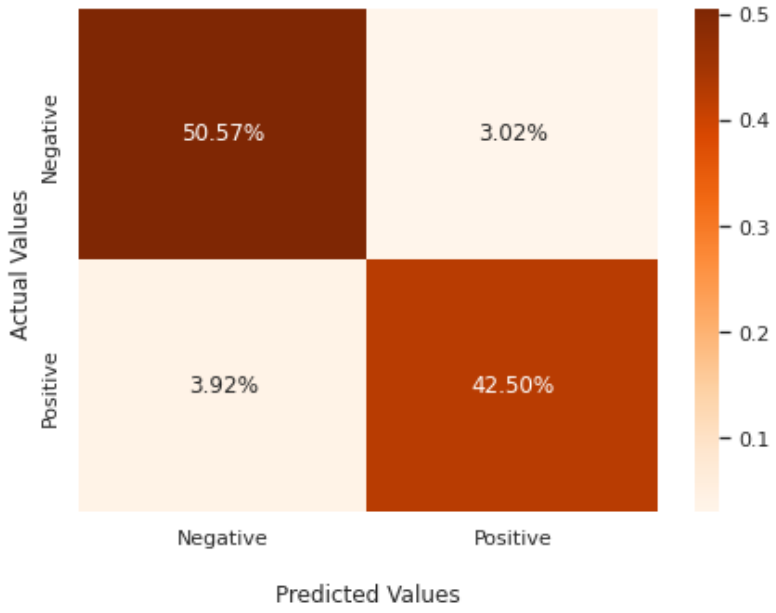

Figure 4.17: LR combined obstructed dataset confusion matrix

4.7 Conclusion

This chapter encompassed the evaluation of different metrics to be able to answer the research questions. Two test cases were also performed to stress test the model's generalisability. Through these evaluations and tests certain weaknesses and future improvements have surfaced. The following list will simplify the answers for the research questions referenced in 4.1 in the same order:

- The models trained with the side view dataset generated better results when compared to the back view dataset. When the two views were combined to produce the combined view dataset and used to train the models, 23 out of 24 models performed better with this dataset.
- When calculating the overall difference on all 24 models the EF produced an increase of 4.6% accuracy. When averaged on all the models, this resulted in a 0.19% accuracy increase. Essentially the answer to the research question is yes, the models did improve although having an almost negligible increase.
- The best classifier when tested against the originally created custom dataset was the MLP classifier. This was somewhat expected since it is essentially the most complex classifier between the four chosen algorithms. Saying that, the test cases showed that this does not always suffice.

Nonetheless, compared to studies done in the field of pose estimation and posture recognition, such as Gatt et al., (2019) and Raju et al., (2020) respectively, the results are up to standard and in some cases even higher than others.

Chapter 5

Conclusions and Recommendations

This research looked into integrating Artificial Intelligence (AI) and pre-processing techniques to distinguish between good and bad standing postures when using standing desks. A total of 24 models were trained and tested against a custom dataset using four different Machine Learning (ML) classifiers. Since there has been a dearth of research in this sector especially when using the MoveNet architecture for skeleton point extraction, this study highlighted what this architecture is capable of. In addition to that, the research questions were all satisfied. The classifiers were chosen based on previous research performed by Kumar et al., (2021), which showed that they performed well with tabular data in pose estimation . After the models were trained and validated against the datasets validation and test splits, two more tests were conducted to assess the models' generalisability. The following were the steps used to conduct this study:

- Current technologies were analyzed to identify what was missing, and the findings can be seen in the Section 2.5.1.
- In Chapter 2 Section 2.5.3, the performance of various classifiers using tabular data for pose estimation was investigated.

- As noted in Chapter 2 Section 2.5, research was done on previous papers methodologies on how to develop posture classification systems in order to collect knowledge on different methods..
- Training the four chosen classifiers (Multilayer Perceptron (MLP), Support Vector Machine (SVM), Random Forest (RF) and Logistic Regressor (LR)) on the custom dataset as documented in Chapter 3 Section 3.11.
- Training the chosen classifiers on the different views' datasets (side-view, back-view and combined-view) as explained in Chapter 3 Section 3.8.1.
- Training the classifiers with different dataset perspectives utilising the engineered feature which can be found in Chapter 3 Section 3.9.1.
- As mentioned in Chapter 4 Section 4.4, the models were tested on the validation and test splits using various metrics.
- Performing two additional test cases to see how the models perform on an external datasets, as documented in Chapter 4 Section 4.5 & 4.6.

After a thorough evaluation of the different metrics in Chapter 4, it was evident that the MLP classifier produced the best results on the custom created dataset out of the four classifiers. The MLP classifier that used the combined dataset without utilising the engineered feature, which was detailed in Section 4.4.1, was in fact, the best performing model out of the 24 models. The accuracy and F1-Score of this model were both 98.5%. The SVM came in close second, with the best model obtaining 97.5% accuracy, followed by the LR with 95.6% accuracy and finally, the RF classifier with an accuracy of 94.3%. In addition, 23 out of 24 models seemed to perform better when trained with the combined dataset.

In Section 4.2, it was also stated that when comparing the back view and side view datasets, the latter performed better on all accounts. Furthermore, although there was not a significant difference between the models that included the EF and those that did not, the EF appeared to improve the total collective accuracy by 4.36%. This was covered in further depth in Chapter 4 Section 4.3. The two test cases that were performed on the models sought out interesting outcomes. The first test case which was detailed in Section 4.5 of the results chapter, showed that the MLP classifier still outperformed other classifiers on the external dataset. The best resultant model achieving an accuracy of 95.1%. The back view model was unable to produce any predictions, but when using a model that was trained with less data, the back view model was able to achieve an accuracy of 96%. The second test case scenario, described in Section 4.6, made use of a dataset in which certain parts of the participant were obscured. The LR resulted to be the most accurate in this experiment, with the best model achieving a 93% accuracy. When compared with the performance of the other models, only one model using the SVM classifier achieved an accuracy of above 60%, whilst the other classifiers were all in the 50% range in this test case. This indicated that all the classifiers except the LR classifier were unable to classify obstructed postures.

5.1 Improvements

5.1.1 Larger Dataset

As previously stated, the dataset used in this investigation included 12,000 frames of data. Due to the magnitude of the dataset, certain pre-processing approaches, such as down sampling could not be applied, since it would drastically reduce the training data. Removing data with highly correlated values would have allowed the models to learn much more since redundant data would be discarded, hence creaing more diversity. In-

creasing the size of the dataset means that the test split would also increase thus giving more reliable results.

5.1.2 Making the Models More Generalisable

The custom dataset was constructed from videos all recorded from the same distance between the person and the camera, which is directly related to the improvement indicated in 5.1.1. This meant that in order for it to be more accurate in a real-life scenario the cameras and the participant needed to be situated at the same distance the data was recorded from for optimal accuracy. This implementation would not be convenient as it is quite constrained. As a result, the custom dataset must be recorded from various distances in order for the models to perform well even if the configuration is modified.

5.1.3 Better Setup for Data Acquisition

It is recommended that the data acquisition is captured from individuals who are actually utilising a standing desk to obtain more accurate results. This was reproduced in this study using a tabletop. This is not ideal since the heights and distances of items such as peripherals might not be up to par with a professional standing desk. As a result, having this data captured on these types of workstations would result in a much more accurate representation of a real-life event.

5.2 Future Work

5.2.1 Creating an Application From the Models

Since the MoveNet lightning architecture for keypoint extraction was used in this work, it created a good opportunity to design a mobile application using these models. This is due to the fact that the lightning model was developed specifically for applications,

sacrificing some accuracy in exchange for faster performance, making it the lightweight version of the MoveNet architecture. This would make the system much more accessible to a wider audience, given the likelihood of owning a smartphone is presently very high and increasing fast (Poushter et al., 2016).

5.2.2 Multi Person Classification

Unlike this study, which can only detect body keypoints for one person in the frame, other models, such as OpenPose, can extract body keypoints from multiple individuals in the frame (Osokin, 2018). This is referred to as multi-person classification, and it sets out an opportunity to train the models using a dataset composed of more than one person. The results can then be compared with the ones produced in this study. Above all, having an implementation that can detect multiple individuals' postures with one camera would make it more viable for enterprises.

5.2.3 Adding Different Engineered Features

In this paper only one EF was taken into consideration, that being the angle between the two skeletal keypoints. It is suggested that in the future, alternative engineered characteristics would be tested against the models to determine if they perform differently. A suggestion for an Engineered Feature (EF) would be implementing an equation that calculates the distance between two keypoints at run time. This makes sense since when considering a bad posture, the resultant body keypoints anatomy shortens in terms of distance between each other and vice versa for good postures.

5.3 Final Words

This research sought out the impact that Artificial Intelligence (AI) approaches can have on our daily lives, particularly in something as vital as posture. The findings suggested that the models were quite capable of identifying standing ergonomics when utilising a standing desk. With a few minor tweaks, this prototype can be easily implemented in settings like offices or retail outlets to guide individuals to improve their posture. For years to come, this research will serve as the foundation for something bigger in the field of posture detection when using a standing desk.

Appendices

A Git project

https://github.com/noasulMSD61A/thesis-code

B Visual Studio Code

This is the code that was developed in visual studio code. Here is were the labeller was developed to label the dataset and the visualizer which takes the trained models and runs them on a video.

B.1 Labeller.py

```python
import cv2
import argparse
import csv
import tensorflow as tf
import tensorflow_hub as hub
import numpy as np
from matplotlib import pyplot as plt
from matplotlib.collections import LineCollection
import matplotlib.patches as patches


class VideoLabeller():
    def __init__(self):
        self.input_size = 192
        self.module = hub.load("https://tfhub.dev/google/movenet/singlepose/lightning/4")
        # Dictionary that maps from joint names to keypoint indices.
        self.KEYPOINT_DICT = {
                'nose': 0,
                'left_eye': 1,
                'right_eye': 2,
                'left_ear': 3,
                'right_ear': 4,
                'left_shoulder': 5,
                'right_shoulder': 6,
                'left_elbow': 7,
                'right_elbow': 8,
                'left_wrist': 9,
                'right_wrist': 10,
                'left_hip': 11,
                'right_hip': 12,
                'left_knee': 13,
                'right_knee': 14,
                'left_ankle': 15,
                'right_ankle': 16
        }
        # Maps bones to a matplotlib color name.
        self.KEYPOINT_EDGE_INDS_TO_COLOR = {
                (0, 1): 'm',
                (0, 2): 'c',
                (1, 3): 'm',
                (2, 4): 'c',
                (0, 5): 'm',
                (0, 6): 'c',
                (5, 7): 'm',
                (7, 9): 'm',
                (6, 8): 'c',
                (8, 10): 'c',
                (5, 6): 'y',
                (5, 11): 'm',
                (6, 12): 'c',
                (11, 12): 'y',
                (11, 13): 'm',
                (13, 15): 'm',
                (12, 14): 'c',
                (14, 16): 'c'
        }

    def _keypoints_and_edges_for_display(self, keypoints_with_scores,
                                                                      height,
                                                                      width,
                                                                      keypoint_threshold=0.11):
        """Returns high confidence keypoints and edges for visualization.

        Args:
            keypoints_with_scores: A numpy array with shape [1, 1, 17, 3] representing
                the keypoint coordinates and scores returned from the MoveNet model.
```

```
            height: height of the image in pixels.
            width: width of the image in pixels.
            keypoint_threshold: minimum confidence score for a keypoint to be
                visualized.

        Returns:
            A (keypoints_xy, edges_xy, edge_colors) containing:
                * the coordinates of all keypoints of all detected entities;
                * the coordinates of all skeleton edges of all detected entities;
                * the colors in which the edges should be plotted.
        """
        keypoints_all = []
        keypoint_edges_all = []
        edge_colors = []
        num_instances, _, _, _ = keypoints_with_scores.shape
        for idx in range(num_instances):
            kpts_x = keypoints_with_scores[0, idx, :, 1]
            kpts_y = keypoints_with_scores[0, idx, :, 0]
            kpts_scores = keypoints_with_scores[0, idx, :, 2]
            kpts_absolute_xy = np.stack(
                    [width * np.array(kpts_x), height * np.array(kpts_y)], axis=-1)
            kpts_above_thresh_absolute = kpts_absolute_xy[
                    kpts_scores > keypoint_threshold, :]
            keypoints_all.append(kpts_above_thresh_absolute)

            for edge_pair, color in self.KEYPOINT_EDGE_INDS_TO_COLOR.items():
                if (kpts_scores[edge_pair[0]] > keypoint_threshold and
                        kpts_scores[edge_pair[1]] > keypoint_threshold):
                    x_start = kpts_absolute_xy[edge_pair[0], 0]
                    y_start = kpts_absolute_xy[edge_pair[0], 1]
                    x_end = kpts_absolute_xy[edge_pair[1], 0]
                    y_end = kpts_absolute_xy[edge_pair[1], 1]
                    line_seg = np.array([[x_start, y_start], [x_end, y_end]])
                    keypoint_edges_all.append(line_seg)
                    edge_colors.append(color)
        if keypoints_all:
            keypoints_xy = np.concatenate(keypoints_all, axis=0)
        else:
            keypoints_xy = np.zeros((0, 17, 2))

        if keypoint_edges_all:
            edges_xy = np.stack(keypoint_edges_all, axis=0)
        else:
            edges_xy = np.zeros((0, 2, 2))
        return keypoints_xy, edges_xy, edge_colors

    def draw_prediction_on_image(self,
            image, keypoints_with_scores, crop_region=None, close_figure=False,
            output_image_height=None):
        """Draws the keypoint predictions on image.

        Args:
            image: A numpy array with shape [height, width, channel] representing the
                pixel values of the input image.
            keypoints_with_scores: A numpy array with shape [1, 1, 17, 3] representing
                the keypoint coordinates and scores returned from the MoveNet model.
            crop_region: A dictionary that defines the coordinates of the bounding box
                of the crop region in normalized coordinates (see the init_crop_region
                function below for more detail). If provided, this function will also
                draw the bounding box on the image.
            output_image_height: An integer indicating the height of the output image.
                Note that the image aspect ratio will be the same as the input image.

        Returns:
            A numpy array with shape [out_height, out_width, channel] representing the
            image overlaid with keypoint predictions.
```

```
        height, width, channel = image.shape
        aspect_ratio = float(width) / height
        fig, ax = plt.subplots(figsize=(12 * aspect_ratio, 12))
        # To remove the huge white borders
        fig.tight_layout(pad=0)
        ax.margins(0)
        ax.set_yticklabels([])
        ax.set_xticklabels([])
        plt.axis('off')

        im = ax.imshow(image)
        line_segments = LineCollection([], linewidths=(4), linestyle='solid')
        ax.add_collection(line_segments)
        # Turn off tick labels
        scat = ax.scatter([], [], s=60, color='#FF1493', zorder=3)

        (keypoint_locs, keypoint_edges,
        edge_colors) = self._keypoints_and_edges_for_display(
                keypoints_with_scores, height, width)

        line_segments.set_segments(keypoint_edges)
        line_segments.set_color(edge_colors)
        if keypoint_edges.shape[0]:
            line_segments.set_segments(keypoint_edges)
            line_segments.set_color(edge_colors)
        if keypoint_locs.shape[0]:
            scat.set_offsets(keypoint_locs)

        if crop_region is not None:
            xmin = max(crop_region['x_min'] * width, 0.0)
            ymin = max(crop_region['y_min'] * height, 0.0)
            rec_width = min(crop_region['x_max'], 0.99) * width - xmin
            rec_height = min(crop_region['y_max'], 0.99) * height - ymin
            rect = patches.Rectangle(
                    (xmin,ymin),rec_width,rec_height,
                    linewidth=1,edgecolor='b',facecolor='none')
            ax.add_patch(rect)

        fig.canvas.draw()
        image_from_plot = np.frombuffer(fig.canvas.tostring_rgb(), dtype=np.uint8)
        image_from_plot = image_from_plot.reshape(
                fig.canvas.get_width_height()[::-1] + (3,))
        plt.close(fig)
        if output_image_height is not None:
            output_image_width = int(output_image_height / height * width)
            image_from_plot = cv2.resize(
                    image_from_plot, dsize=(output_image_width, output_image_height),
                        interpolation=cv2.INTER_CUBIC)
        return image_from_plot

    def movenet(self,input_image):
        """Runs detection on an input image.

        Args:
            input_image: A [1, height, width, 3] tensor represents the input image
                pixels. Note that the height/width should already be resized and match the
                expected input resolution of the model before passing into this function.

        Returns:
            A [1, 1, 17, 3] float numpy array representing the predicted keypoint
            coordinates and scores.
        """
        model = self.module.signatures['serving_default']
```

```
        # SavedModel format expects tensor type of int32.
        input_image = tf.cast(input_image, dtype=tf.int32)
        # Run model inference.
        outputs = model(input_image)
        # Output is a [1, 1, 17, 3] tensor.
        keypoints_with_scores = outputs['output_0'].numpy()
        return keypoints_with_scores

    def get_keypoints(self,frame):
        image_rgb = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)

        input_image = tf.expand_dims(image_rgb, axis=0)
        input_image = tf.image.resize_with_pad(input_image, self.input_size, self.input_size)
        keypoints_with_scores = self.movenet(input_image)
        return keypoints_with_scores

    def draw_keypoints(self,frame, keypoints):
        display_image = tf.expand_dims(frame, axis=0)
        display_image = tf.cast(tf.image.resize_with_pad(
            display_image, 1280, 1280), dtype=tf.int32)
        output_overlay = self.draw_prediction_on_image(
            np.squeeze(display_image.numpy(), axis=0), keypoints)
        return output_overlay

    def engineer_features_mock(self,keypoints):
        return list(keypoints)

if __name__ == '__main__':

    parser = argparse.ArgumentParser(description='Process some integers.')
    parser.add_argument('--back', type=str,
                        help='path of the back view video')
    parser.add_argument('--side', type=str,
                        help='path of the side view video')
    parser.add_argument('--out', type=str,
                        help='path of the output CSV')
    args = parser.parse_args()


    lab = VideoLabeller()
    print("\n\n\n\n Finished Initialization \n\n\n\n")
    vidcap1 = cv2.VideoCapture(args.back)
    success1,image1 = vidcap1.read()
    len1 = int(vidcap1.get(cv2.CAP_PROP_FRAME_COUNT))
    vidcap2 = cv2.VideoCapture(args.side)
    success2,image2 = vidcap2.read()
    len2 = int(vidcap2.get(cv2.CAP_PROP_FRAME_COUNT))
    frame_count = np.min([len1,len2])

    arr_labels = [None] * frame_count
    arr_features = [None] * frame_count

    # first loop - manually label 1 out of every 6 frames
    count = 0
    while success1 and success2:

        if count % 6 == 0:
            print(f'labelling frame {count+1}')

            back_keypoints = lab.get_keypoints(image1)
            back_image = lab.draw_keypoints(image1, back_keypoints)

            side_keypoints = lab.get_keypoints(image2)
            side_image = lab.draw_keypoints(image2, side_keypoints)

            image_with_keypoints = cv2.hconcat([back_image, side_image])
            cv2.imshow('jpg', image_with_keypoints)
```

```
        if k == 119: #up arrow - good posture
            print("good posture")
            label = 1
        elif k == 115: #down arrow - bad posture
            print("bad posture")
            label = 0
        else:
            print(f'ERROR: please label again ... you pressed key {k}')
            continue

        features = list(back_keypoints.flatten()) + list(side_keypoints.flatten())
        features = lab.engineer_features_mock(features)

        arr_features[count] = features
        arr_labels[count] = label
        #row = [count] + row + [ label ]
        #dataset.append(row)

    success1,image1 = vidcap1.read()
    success2,image2 = vidcap2.read()
    count += 1

cv2.destroyAllWindows()
print(arr_labels)
print(arr_features)

#post processing loop
vidcap1 = cv2.VideoCapture(args.back)
vidcap2 = cv2.VideoCapture(args.side)
for i in range(len(arr_labels)):
    if arr_labels[i] is None:
        arr_labels[i] = arr_labels[i-1]

        vidcap1.set(cv2.CAP_PROP_POS_FRAMES,i)
        success1,image1 = vidcap1.read()
        vidcap2.set(cv2.CAP_PROP_POS_FRAMES,i)
        success2,image2 = vidcap2.read()

        if success1 and success2:
            back_keypoints = lab.get_keypoints(image1)
            back_image = lab.draw_keypoints(image1, back_keypoints)

            side_keypoints = lab.get_keypoints(image2)
            side_image = lab.draw_keypoints(image2, side_keypoints)

            features = list(back_keypoints.flatten()) + list(side_keypoints.flatten())
            features = lab.engineer_features_mock(features)

            arr_features[i] = features

        else:
            print("error on frame "+ i)


print(arr_labels)
print(arr_features)

dataset = []
for i in range(len(arr_labels)):
    dataset.append( [i] + arr_features[i] + [arr_labels[i]] )

with open(f'{args.out}', "w", newline="") as f:
    writer = csv.writer(f)
    writer.writerows(dataset)
```

B.2 visualizer.py

```
import pickle
import cv2
import argparse
import csv
import tensorflow as tf
import tensorflow_hub as hub
import numpy as np
from matplotlib import pyplot as plt
from matplotlib.collections import LineCollection
import matplotlib.patches as patches
import pandas as pd
import math
import PySimpleGUI as sg


feature_names = [
    "back_nose_x",
    "back_nose_y",
    "back_nose_z",
    "back_left_eye_x",
    "back_left_eye_y",
    "back_left_eye_z",
    "back_right_eye_x",
    "back_right_eye_y",
    "back_right_eye_z",
    "back_left_ear_x",
    "back_left_ear_y",
    "back_left_ear_z",
    "back_right_ear_x",
    "back_right_ear_y",
    "back_right_ear_z",
    "back_left_shoulder_x",
    "back_left_shoulder_y",
    "back_left_shoulder_z",
    "back_right_shoulder_x",
    "back_right_shoulder_y",
    "back_right_shoulder_z",
    "back_left_elbow_x",
    "back_left_elbow_y",
    "back_left_elbow_z",
    "back_right_elbow_x",
    "back_right_elbow_y",
    "back_right_elbow_z",
```

```
"back_left_wrist_x",
"back_left_wrist_y",
"back_left_wrist_z",
"back_right_wrist_x",
"back_right_wrist_y",
"back_right_wrist_z",
"back_left_hip_x",
"back_left_hip_y",
"back_left_hip_z",
"back_right_hip_x",
"back_right_hip_y",
"back_right_hip_z",
"back_left_knee_x",
"back_left_knee_y",
"back_left_knee_z",
"back_right_knee_x",
"back_right_knee_y",
"back_right_knee_z",
"back_left_ankle_x",
"back_left_ankle_y",
"back_left_ankle_z",
"back_right_ankle_x",
"back_right_ankle_y",
"back_right_ankle_z",
"side_nose_x",
"side_nose_y",
"side_nose_z",
"side_left_eye_x",
"side_left_eye_y",
"side_left_eye_z",
"side_right_eye_x",
"side_right_eye_y",
"side_right_eye_z",
"side_left_ear_x",
"side_left_ear_y",
"side_left_ear_z",
"side_right_ear_x",
"side_right_ear_y",
"side_right_ear_z",
"side_left_shoulder_x",
"side_left_shoulder_y",
"side_left_shoulder_z",
"side_right_shoulder_x",
"side_right_shoulder_y",
"side_right_shoulder_z",
```

```
    "side_left_elbow_x",
    "side_left_elbow_y",
    "side_left_elbow_z",
    "side_right_elbow_x",
    "side_right_elbow_y",
    "side_right_elbow_z",
    "side_left_wrist_x",
    "side_left_wrist_y",
    "side_left_wrist_z",
    "side_right_wrist_x",
    "side_right_wrist_y",
    "side_right_wrist_z",
    "side_left_hip_x",
    "side_left_hip_y",
    "side_left_hip_z",
    "side_right_hip_x",
    "side_right_hip_y",
    "side_right_hip_z",
    "side_left_knee_x",
    "side_left_knee_y",
    "side_left_knee_z",
    "side_right_knee_x",
    "side_right_knee_y",
    "side_right_knee_z",
    "side_left_ankle_x",
    "side_left_ankle_y",
    "side_left_ankle_z",
    "side_right_ankle_x",
    "side_right_ankle_y",
    "side_right_ankle_z",
]

def calcAngle(x1,y1,x2,y2):
  angles = []
  for i in range(len(x1)):
    angles.append(math.atan2(y2[i] - y1[i], x2[i] - x1[i]) * 180/math.pi)
  return angles

class FeatureEngineeringTransformer():
  def __init__(self):
    self.angles_joints = {
        'left_torso': ('left_hip','left_shoulder'),
        'right_torso': ('right_hip','right_shoulder'),
        'left_ulna': ('left_wrist','left_elbow'),
        'right_ulna': ('right_wrist','right_elbow'),
```

```
        'left_los': ('left_eye','nose'),#line of sight
        'right_los': ('right_eye','nose')
    }

  def fit(self,x, y=None):
    return self

  def transform(self,x):
    for angle in self.angles_joints:
      joint1, joint2 = self.angles_joints[angle]
      x[f'back_{angle}_angle'] = calcAngle(
          x1 = x[f'back_{joint1}_x'].values,
          y1 = x[f'back_{joint1}_y'].values,
          x2 = x[f'back_{joint2}_x'].values,
          y2 = x[f'back_{joint2}_y'].values
      )
      x[f'side_{angle}_angle'] = calcAngle(
          x1 = x[f'side_{joint1}_x'].values,
          y1 = x[f'side_{joint1}_y'].values,
          x2 = x[f'side_{joint2}_x'].values,
          y2 = x[f'side_{joint2}_y'].values
      )
    return x

class FeatureSelectionTransformer():
  def __init__(self, mode):
    self.mode = mode

  def fit(self,x, y=None):
    return self

  def transform(self, x, y=None):
    if self.mode == "back":
      cols_to_drop = [c for c in x.columns if "side_" in c]
    elif self.mode == "side":
      cols_to_drop = [c for c in x.columns if "back_" in c]
    elif self.mode == "combined":
      cols_to_drop = []

    x = x[[c for c in x.columns if c not in cols_to_drop]]
    return x




class VideoLabeller():
    def __init__(self):
        self.input_size = 192
        self.module = hub.load("https://tfhub.dev/google/movenet/singlepose/lightning/4")
        # Dictionary that maps from joint names to keypoint indices.
        self.KEYPOINT_DICT = {
                'nose': 0,
                'left_eye': 1,
                'right_eye': 2,
                'left_ear': 3,
                'right_ear': 4,
                'left_shoulder': 5,
                'right_shoulder': 6,
                'left_elbow': 7,
                'right_elbow': 8,
                'left_wrist': 9,
                'right_wrist': 10,
                'left_hip': 11,
                'right_hip': 12,
                'left_knee': 13,
                'right_knee': 14,
```

```
            'left_ankle': 15,
            'right_ankle': 16
        }
        # Maps bones to a matplotlib color name.
        self.KEYPOINT_EDGE_INDS_TO_COLOR = {
            (0, 1): 'm',
            (0, 2): 'c',
            (1, 3): 'm',
            (2, 4): 'c',
            (0, 5): 'm',
            (0, 6): 'c',
            (5, 7): 'm',
            (7, 9): 'm',
            (6, 8): 'c',
            (8, 10): 'c',
            (5, 6): 'y',
            (5, 11): 'm',
            (6, 12): 'c',
            (11, 12): 'y',
            (11, 13): 'm',
            (13, 15): 'm',
            (12, 14): 'c',
            (14, 16): 'c'
        }

    def _keypoints_and_edges_for_display(self, keypoints_with_scores,
                                         height,
                                         width,
                                         keypoint_threshold=0.11):
        """Returns high confidence keypoints and edges for visualization.

        Args:
            keypoints_with_scores: A numpy array with shape [1, 1, 17, 3] representing
                the keypoint coordinates and scores returned from the MoveNet model.
            height: height of the image in pixels.
            width: width of the image in pixels.
            keypoint_threshold: minimum confidence score for a keypoint to be
                visualized.

        Returns:
            A (keypoints_xy, edges_xy, edge_colors) containing:
                * the coordinates of all keypoints of all detected entities;
                * the coordinates of all skeleton edges of all detected entities;
                * the colors in which the edges should be plotted.
        """
        keypoints_all = []
        keypoint_edges_all = []
        edge_colors = []
        num_instances, _, _, _ = keypoints_with_scores.shape
        for idx in range(num_instances):
            kpts_x = keypoints_with_scores[0, idx, :, 1]
            kpts_y = keypoints_with_scores[0, idx, :, 0]
            kpts_scores = keypoints_with_scores[0, idx, :, 2]
            kpts_absolute_xy = np.stack(
                [width * np.array(kpts_x), height * np.array(kpts_y)], axis=-1)
            kpts_above_thresh_absolute = kpts_absolute_xy[
                kpts_scores > keypoint_threshold, :]
            keypoints_all.append(kpts_above_thresh_absolute)

            for edge_pair, color in self.KEYPOINT_EDGE_INDS_TO_COLOR.items():
                if (kpts_scores[edge_pair[0]] > keypoint_threshold and
                        kpts_scores[edge_pair[1]] > keypoint_threshold):
                    x_start = kpts_absolute_xy[edge_pair[0], 0]
                    y_start = kpts_absolute_xy[edge_pair[0], 1]
                    x_end = kpts_absolute_xy[edge_pair[1], 0]
```

```
                    y_end = kpts_absolute_xy[edge_pair[1], 1]
                    line_seg = np.array([[x_start, y_start], [x_end, y_end]])
                    keypoint_edges_all.append(line_seg)
                    edge_colors.append(color)
        if keypoints_all:
            keypoints_xy = np.concatenate(keypoints_all, axis=0)
        else:
            keypoints_xy = np.zeros((0, 17, 2))

        if keypoint_edges_all:
            edges_xy = np.stack(keypoint_edges_all, axis=0)
        else:
            edges_xy = np.zeros((0, 2, 2))
        return keypoints_xy, edges_xy, edge_colors

    def draw_prediction_on_image(self,
            image, keypoints_with_scores, crop_region=None, close_figure=False,
            output_image_height=None):
        """Draws the keypoint predictions on image.

        Args:
            image: A numpy array with shape [height, width, channel] representing the
                pixel values of the input image.
            keypoints_with_scores: A numpy array with shape [1, 1, 17, 3] representing
                the keypoint coordinates and scores returned from the MoveNet model.
            crop_region: A dictionary that defines the coordinates of the bounding box
                of the crop region in normalized coordinates (see the init_crop_region
                function below for more detail). If provided, this function will also
                draw the bounding box on the image.
            output_image_height: An integer indicating the height of the output image.
                Note that the image aspect ratio will be the same as the input image.

        Returns:
            A numpy array with shape [out_height, out_width, channel] representing the
            image overlaid with keypoint predictions.
        """
        height, width, channel = image.shape
        aspect_ratio = float(width) / height
        fig, ax = plt.subplots(figsize=(12 * aspect_ratio, 12))
        # To remove the huge white borders
        fig.tight_layout(pad=0)
        ax.margins(0)
        ax.set_yticklabels([])
        ax.set_xticklabels([])
        plt.axis('off')

        im = ax.imshow(image)
        line_segments = LineCollection([], linewidths=(4), linestyle='solid')
        ax.add_collection(line_segments)
        # Turn off tick labels
        scat = ax.scatter([], [], s=60, color='#FF1493', zorder=3)

        (keypoint_locs, keypoint_edges,
        edge_colors) = self._keypoints_and_edges_for_display(
                keypoints_with_scores, height, width)

        line_segments.set_segments(keypoint_edges)
        line_segments.set_color(edge_colors)
        if keypoint_edges.shape[0]:
            line_segments.set_segments(keypoint_edges)
            line_segments.set_color(edge_colors)
        if keypoint_locs.shape[0]:
            scat.set_offsets(keypoint_locs)

        if crop_region is not None:
            xmin = max(crop_region['x_min'] * width, 0.0)
```

```
            ymin = max(crop_region['y_min'] * height, 0.0)
            rec_width = min(crop_region['x_max'], 0.99) * width - xmin
            rec_height = min(crop_region['y_max'], 0.99) * height - ymin
            rect = patches.Rectangle(
                    (xmin,ymin),rec_width,rec_height,
                    linewidth=1,edgecolor='b',facecolor='none')
            ax.add_patch(rect)

        fig.canvas.draw()
        image_from_plot = np.frombuffer(fig.canvas.tostring_rgb(), dtype=np.uint8)
        image_from_plot = image_from_plot.reshape(
                fig.canvas.get_width_height()[::-1] + (3,))
        plt.close(fig)
        if output_image_height is not None:
            output_image_width = int(output_image_height / height * width)
            image_from_plot = cv2.resize(
                    image_from_plot, dsize=(output_image_width, output_image_height),
                        interpolation=cv2.INTER_CUBIC)
        return image_from_plot

    def movenet(self,input_image):
        """Runs detection on an input image.

        Args:
            input_image: A [1, height, width, 3] tensor represents the input image
                pixels. Note that the height/width should already be resized and match the
                expected input resolution of the model before passing into this function.

        Returns:
            A [1, 1, 17, 3] float numpy array representing the predicted keypoint
            coordinates and scores.
        """
        model = self.module.signatures['serving_default']

        # SavedModel format expects tensor type of int32.
        input_image = tf.cast(input_image, dtype=tf.int32)
        # Run model inference.
        outputs = model(input_image)
        # Output is a [1, 1, 17, 3] tensor.
        keypoints_with_scores = outputs['output_0'].numpy()
        return keypoints_with_scores

    def get_keypoints(self,frame):
        image_rgb = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)

        input_image = tf.expand_dims(image_rgb, axis=0)
        input_image = tf.image.resize_with_pad(input_image, self.input_size, self.input_size)
        keypoints_with_scores = self.movenet(input_image)
        return keypoints_with_scores

    def draw_keypoints(self,frame, keypoints):
        display_image = tf.expand_dims(frame, axis=0)
        display_image = tf.cast(tf.image.resize_with_pad(
            display_image, 1280, 1280), dtype=tf.int32)
        output_overlay = self.draw_prediction_on_image(
            np.squeeze(display_image.numpy(), axis=0), keypoints)
        return output_overlay

    def engineer_features_mock(self,keypoints):
        return list(keypoints)
```

```python
if __name__ == '__main__':

    parser = argparse.ArgumentParser(description='Process some integers.')
    parser.add_argument('--back', type=str,
                        help='path of the back view video')
    parser.add_argument('--side', type=str,
                        help='path of the side view video')
    args = parser.parse_args()

    with open('./pipelines/side_engFalse_LR.pkl', 'rb') as file:
        model_side = pickle.load(file)

    with open('./pipelines/back_engFalse_LR.pkl', 'rb') as file:
        model_back = pickle.load(file)

    with open('./pipelines/combined_engFalse_LR.pkl', 'rb') as file:
        model_combined = pickle.load(file)

    lab = VideoLabeller()
    print("\n\n\n\n Finished Initialization \n\n\n\n")
    vidcap1 = cv2.VideoCapture(args.back)
    success1,image1 = vidcap1.read()
    len1 = int(vidcap1.get(cv2.CAP_PROP_FRAME_COUNT))
    vidcap2 = cv2.VideoCapture(args.side)
    success2,image2 = vidcap2.read()
    len2 = int(vidcap2.get(cv2.CAP_PROP_FRAME_COUNT))
    frame_count = np.min([len1,len2])

    # first loop - manually label 1 out of every 6 frames
    count = 0
    while success1 and success2:

        back_keypoints = lab.get_keypoints(image1)
        back_image = lab.draw_keypoints(image1, back_keypoints)

        side_keypoints = lab.get_keypoints(image2)
        side_image = lab.draw_keypoints(image2, side_keypoints)

        image_with_keypoints = cv2.hconcat([back_image, side_image])

        features = list(back_keypoints.flatten()) + list(side_keypoints.flatten())
        features = lab.engineer_features_mock(features)

        pipeline_input = pd.Series(features).to_frame().transpose()
        pipeline_input.columns = feature_names

        pred_side = model_side.predict(pipeline_input)[0]
        pred_side_str = "side dataset: good posture" if pred_side==1 else "side dataset: bad posture"
        cv2.putText(
            image_with_keypoints, #numpy array on which text is written
            pred_side_str, #text
            (1300,100), #position at which writing has to start
            cv2.FONT_HERSHEY_SIMPLEX, #font family
            1.5, #font size
            (0, 0, 255, 255), #font color
            2) #font stroke

        pred_back = model_back.predict(pipeline_input)[0]
        pred_back_str = "back dataset: good posture" if pred_back==1 else "back dataset: bad posture"
        cv2.putText(
            image_with_keypoints, #numpy array on which text is written
            pred_back_str, #text
            (100,100), #position at which writing has to start
            cv2.FONT_HERSHEY_SIMPLEX, #font family
```

```
            1.5, #font size
            (0, 0, 255, 255), #font color
            2), #font stroke


        pred_combined = model_combined.predict(pipeline_input)[0]
        pred_combined_str = "combined dataset: good posture" if pred_combined==1 else "combined dataset: bad posture"
        cv2.putText(
            image_with_keypoints, #numpy array on which text is written
            pred_combined_str, #text
            (800,1000), #position at which writing has to start
            cv2.FONT_HERSHEY_SIMPLEX, #font family
            1.5, #font size
            (0, 0, 255, 255), #font color
            2) #font stroke

        cv2.imshow('jpg', image_with_keypoints)
        cv2.waitKey(1)

        success1,image1 = vidcap1.read()
        success2,image2 = vidcap2.read()
        count += 1



cv2.destroyAllWindows()
```

C Google Colab Code

This section includes the code that was used to pre-process, split data and train the models. Apart from that the code that was used to test the models against different datasets is also included.

C.1 pipelines.py

```
from google.colab import drive
drive.mount('/content/gdrive')
test_path = 'gdrive/My Drive/Colab Notebooks/NoahTrim_Proper.csv'

from sklearn.model_selection import train_test_split
import pandas as pd
import glob

from sklearn.linear_model import LogisticRegression

from sklearn import svm
from sklearn.model_selection import GridSearchCV
from sklearn.neural_network import MLPClassifier
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
import pickle

import tensorflow as tf

from tensorflow.keras import datasets, layers, models
import matplotlib.pyplot as plt

from sklearn.ensemble import RandomForestClassifier
import matplotlib.pyplot as plt
from matplotlib import rcParams
rcParams['figure.figsize'] = 14, 7
rcParams['axes.spines.top'] = False
rcParams['axes.spines.right'] = False
import math
from sklearn.metrics import f1_score
from sklearn.metrics import accuracy_score

from sklearn.pipeline import Pipeline
from sklearn.model_selection import StratifiedShuffleSplit
import numpy as np
from sklearn import metrics
from sklearn.metrics import precision_recall_curve
from sklearn.metrics import confusion_matrix
import seaborn as sns
from sklearn.metrics import precision_score, recall_score
from sklearn.metrics import classification_report

pd.set_option('display.max_columns', None)
```

```
Mounted at /content/gdrive
```

```
!pip install interpret
from interpret import show
from interpret.blackbox import LimeTabular
```

```
path = 'gdrive/My Drive/Colab Notebooks/'
all_files = {

     "p1": "p1.csv",
     "p2": "p2.csv",
     "p3": "p3.csv",
     "p4": "p4.csv",
     "p5": "p5.csv",
     "p6": "p6.csv",
     "p7": "p7.csv",
     "p8": "p8.csv",
     "p9": "p9.csv",
     "p10": "p10.csv"

}

li = []

for alias in all_files:
    temp_df = pd.read_csv(path+all_files[alias], index_col=None, header=0)
    temp_df["file_id"] = alias
    li.append(temp_df)

    kp_names = [
      'nose',
      'left_eye',
      'right_eye',
      'left_ear',
      'right_ear',
      'left_shoulder',
      'right_shoulder',
      'left_elbow',
      'right_elbow',
      'left_wrist',
      'right_wrist',
      'left_hip',
      'right_hip',
      'left_knee',
      'right_knee',
      'left_ankle',
      'right_ankle',
    ]

    colnames = []
    for cam in ["back", "side"]:
      for i in kp_names:
        for pt in ["x", "y", "z"]:
          colnames.append(f"{cam}_{i}_{pt}")
    colnames = ["frame_count"] + colnames + ["label", "file_id"]
    temp_df.columns = colnames

df = pd.concat(li, axis=0, ignore_index=True)
dftest = df.apply(lambda x : True
            if x['label'] == 1 else False, axis = 1)
```

Splitting up the Dataset

```
X = df[[c for c in df.columns if c not in ("frame_count","label", "file_id")]]
y = df["label"]
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.20, random_state=0)#test set
X_train, X_val, y_train, y_val = train_test_split(X_train, y_train, test_size=0.20, random_state=0)
```

```
for c in X_train.columns: print(c)
```

Creating the Custom Transformers

```
def calcAngle(x1,y1,x2,y2):
  angles = []
  for i in range(len(x1)):
    angles.append(math.atan2(y2[i] - y1[i], x2[i] - x1[i]) * 180/math.pi)
  return angles

class FeatureEngineeringTransformer():
  def __init__(self):
    self.angles_joints = {
        'left_torso': ('left_hip','left_shoulder'),
        'right_torso': ('right_hip','right_shoulder'),
        'left_ulna': ('left_wrist','left_elbow'),
        'right_ulna': ('right_wrist','right_elbow'),
        'left_los': ('left_eye','nose'),#line of sight
        'right_los': ('right_eye','nose')
    }

  def fit(self,x, y=None):
    return self

  def transform(self,x):
    for angle in self.angles_joints:
      joint1, joint2 = self.angles_joints[angle]
      x[f'back_{angle}_angle'] = calcAngle(
          x1 = x[f'back_{joint1}_x'].values,
          y1 = x[f'back_{joint1}_y'].values,
          x2 = x[f'back_{joint2}_x'].values,
          y2 = x[f'back_{joint2}_y'].values
      )
      x[f'side_{angle}_angle'] = calcAngle(
          x1 = x[f'side_{joint1}_x'].values,
          y1 = x[f'side_{joint1}_y'].values,
          x2 = x[f'side_{joint2}_x'].values,
          y2 = x[f'side_{joint2}_y'].values
      )
    return x
```

```
class FeatureSelectionTransformer():
  def __init__(self, mode):
    self.mode = mode

  def fit(self,x, y=None):
    return self

  def transform(self, x, y=None):
    if self.mode == "back":
      cols_to_drop = [c for c in x.columns if "side_" in c]
    elif self.mode == "side":
      cols_to_drop = [c for c in x.columns if "back_" in c]
    elif self.mode == "combined":
      cols_to_drop = []

    x = x[[c for c in x.columns if c not in cols_to_drop]]
    return x

df
```

Creating the Pipeline

```
[ ]  results = {}

for dataset in ["combined", "back", "side"]:
  for engineering in [True, False]:
    for model in ["SVM", "LR", "RF", "NN"]:
      pipeline_name = f"{dataset}_{'engTrue' if engineering else 'engFalse'}_{model}"
      print(pipeline_name)
      pipeline_steps = []
      if engineering:
        pipeline_steps.append(('eng', FeatureEngineeringTransformer()))

      pipeline_steps.append(('sel', FeatureSelectionTransformer(dataset)))
      pipeline_steps.append(('scale', StandardScaler()))


      if model == "SVM":
        params={
            'C':[0.1,1],
            'kernel':['poly','linear'],
            'gamma': ['scale', 'auto']
        }
        classifier_svm = svm.SVC()
        clf = GridSearchCV(classifier_svm, param_grid=params)
```

```
elif model == "RF":
    params={
    'n_estimators':[120,140],
    'min_samples_split':[3,4,5],
    'max_depth': [3,4]
    }
    RN = RandomForestClassifier()
    clf=GridSearchCV(RN,param_grid=params)

elif model == "NN":
  parameters = {'batch_size': [8,16,32, 64], 'hidden_layer_sizes' : [(50,), (100,) , (200,)], 'early_stopping': [True]}
  MLP = MLPClassifier(random_state=1, max_iter=500)
  clf = GridSearchCV(MLP, parameters)
  #clf = MLPClassifier(random_state=1, max_iter=500, batch_size=32, hidden_layer_sizes=(100,))
pipeline_steps.append(('clf', clf))

temp_pipeline = Pipeline(pipeline_steps).fit(X_train.copy(), y_train.copy())

y_pred = temp_pipeline.predict(X_val.copy())#X_test
y_predtest = temp_pipeline.predict(X_test.copy())
results[pipeline_name] = {
    'model': model,
    'dataset': dataset,
    'feature_engineering': engineering,
    'pipeline': temp_pipeline,
    'accuracy': accuracy_score(y_val, y_pred) * 100,
    'F1': f1_score(y_val,y_pred) * 100, #y-test instead of y val in test set
    'test_accuracy': accuracy_score(y_test, y_predtest) * 100,
    'F1_test': f1_score(y_test,y_predtest) * 100
}

with open('gdrive/MyDrive/Colab Notebooks/Pipelines_Latest/'+pipeline_name+'.pkl', 'wb') as file:
  pickle.dump(temp_pipeline,file)
```

```
with open('gdrive/My Drive/Colab Notebooks/Pipelines_Latest/back_engFalse_RF.pkl', 'rb') as file:
        model_side = pickle.load(file)
```

```
[ ] y_predtest = model_side.predict(X_test.copy())
    #y_pred = model_side.predict(X)
    accuracy = accuracy_score(y_test, y_predtest)

    y_pred_proba = model_side.predict_proba(X_test)[::,1]

    fpr, tpr, _ = metrics.roc_curve(y_test, y_pred_proba)
    auc = metrics.roc_auc_score(y_test, y_pred_proba)

    plt.plot(fpr,tpr,label="RF AUC="+str(auc))
    plt.ylabel('True Positive Rate')
    plt.xlabel('False Positive Rate')
    plt.legend(loc=4)
    plt.show()
    #f1_score(y_pred,y)
```

```
[ ] y_predtest = model_side.predict(X_test.copy())
    cf_matrix = confusion_matrix(y_test, y_predtest)
    print(cf_matrix)

    ax = sns.heatmap(cf_matrix/np.sum(cf_matrix), annot=True, fmt='.2%', cmap='Oranges')
    #ax.set_title('back_engFalse_LR\n\n');
    ax.set_xlabel('\nPredicted Values')
    ax.set_ylabel('Actual Values ');
    ax.xaxis.set_ticklabels(['Negative','Positive'])
    ax.yaxis.set_ticklabels(['Negative','Positive'])
    plt.show()

    print(classification_report(y_test, y_predtest))
```

```
[ ] y_predtest = model_side.predict(X_test.copy())
    ps = precision_score(y_test, y_predtest)
    print(ps)
    rs = recall_score(y_test, y_predtest)
    print(rs)
```

C.2 testset.py

```
from google.colab import drive
drive.mount('/content/gdrive')
test_path = 'gdrive/My Drive/Colab Notebooks/NoahTrim_Proper.csv'

from sklearn.model_selection import train_test_split
import pandas as pd
import glob

from sklearn.linear_model import LogisticRegression

from sklearn import svm
from sklearn.model_selection import GridSearchCV
from sklearn.neural_network import MLPClassifier
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
import pickle

import tensorflow as tf

from tensorflow.keras import datasets, layers, models
import matplotlib.pyplot as plt

from sklearn.ensemble import RandomForestClassifier
import matplotlib.pyplot as plt
from matplotlib import rcParams
rcParams['figure.figsize'] = 14, 7
rcParams['axes.spines.top'] = False
rcParams['axes.spines.right'] = False
import math
from sklearn.metrics import f1_score
from sklearn.metrics import accuracy_score

from sklearn.pipeline import Pipeline
from sklearn.model_selection import StratifiedShuffleSplit
import numpy as np
from sklearn import metrics
from sklearn.metrics import precision_recall_curve
from sklearn.metrics import confusion_matrix
import seaborn as sns
from sklearn.metrics import precision_score, recall_score
from sklearn.metrics import classification_report

pd.set_option('display.max_columns', None)
import plotly.express as px
```

```
X = df[[c for c in df.columns if c not in ("frame_count","label", "file_id")]]
y = df["label"]
```

```
def calcAngle(x1,y1,x2,y2):
  angles = []
  for i in range(len(x1)):
    angles.append(math.atan2(y2[i] - y1[i], x2[i] - x1[i]) * 180/math.pi)
  return angles

class FeatureEngineeringTransformer():
  def __init__(self):
    self.angles_joints = {
        'left_torso': ('left_hip','left_shoulder'),
        'right_torso': ('right_hip','right_shoulder'),
        'left_ulna': ('left_wrist','left_elbow'),
        'right_ulna': ('right_wrist','right_elbow'),
        'left_los': ('left_eye','nose'),#line of sight
        'right_los': ('right_eye','nose')
    }

  def fit(self,x, y=None):
    return self

  def transform(self,x):
    for angle in self.angles_joints:
      joint1, joint2 = self.angles_joints[angle]
      x[f'back_{angle}_angle'] = calcAngle(
          x1 = x[f'back_{joint1}_x'].values,
          y1 = x[f'back_{joint1}_y'].values,
          x2 = x[f'back_{joint2}_x'].values,
          y2 = x[f'back_{joint2}_y'].values
      )
      x[f'side_{angle}_angle'] = calcAngle(
          x1 = x[f'side_{joint1}_x'].values,
          y1 = x[f'side_{joint1}_y'].values,
          x2 = x[f'side_{joint2}_x'].values,
          y2 = x[f'side_{joint2}_y'].values
      )
    return x

class FeatureSelectionTransformer():
  def __init__(self, mode):
    self.mode = mode

  def fit(self,x, y=None):
    return self
```

```
  def transform(self, x, y=None):
    if self.mode == "back":
      cols_to_drop = [c for c in x.columns if "side_" in c]
    elif self.mode == "side":
      cols_to_drop = [c for c in x.columns if "back_" in c]
    elif self.mode == "combined":
      cols_to_drop = []

    x = x[[c for c in x.columns if c not in cols_to_drop]]
    return x
```

```
with open('gdrive/My Drive/Colab Notebooks/pipelines/back_engFalse_NN.pkl', 'rb') as file:
        model_side = pickle.load(file)
```

```
y_pred = model_side.predict(X.copy())

accuracy_score(y, y_pred)
```

```
0.9608482871125612
```

```
with open('gdrive/My Drive/Colab Notebooks/Pipelines_Latest/back_engTrue_LR.pkl', 'rb') as file:
        model_side = pickle.load(file)
```

Bibliography

AAOS, P. (1947), 'Posture and its relationship to orthopedic disabilities', *A Report of the Posture Committee of the American Academy of Orthopedic Surgeons* .

Alberts, J. L., Hirsch, J. R., Koop, M. M., Schindler, D. D., Kana, D. E., Linder, S. M., Campbell, S. & Thota, A. K. (2015), 'Using accelerometer and gyroscopic measures to quantify postural stability', *Journal of athletic training* **50**(6), 578–588.

Alzubi, J., Nayyar, A. & Kumar, A. (2018), Machine learning from theory to algorithms: an overview, *in* 'Journal of physics: conference series', Vol. 1142, IOP Publishing, pp. 5–6.

Boateng, E. Y., Otoo, J. & Abaye, D. A. (2020), 'Basic tenets of classification algorithms k-nearest-neighbor, support vector machine, random forest and neural network: a review', *Journal of Data Analysis and Information Processing* **8**(4), 341–357.

Bonaccorso, G. (2017), *Machine learning algorithms*, Packt Publishing Ltd.

Burr, R. (2021), 'Guide to proper sitting and standing desk ergonomics - start standing'. **URL:** *https://www.startstanding.org/proper-workplace-ergonomics/*

Chandler, M. (2022), 'Standing desk ergonomics guide: The best practices in 2022'. **URL:** *https://ergonomicshealth.com/standing-desk-ergonomics/*

Chang, W.-Y. & Chen, C.-S. (2004), Pose estimation for multiple camera systems, *in* 'Proceedings of the 17th International Conference on Pattern Recognition, 2004. ICPR 2004.', Vol. 3, IEEE, pp. 262–265.

Chen, S. & Yang, R. R. (2020), 'Pose trainer: correcting exercise posture using pose estimation', *arXiv preprint arXiv:2006.11718* .

Christy, D. V. (2019), 'Ergonomics and employee engagement', *International Journal of Mechanical Engineering and Technology* **10**(2), 105–109.

ensemble methods (n.d.).
URL: *https://scikit-learn.org/stable/modules/ensemble.htmlforest*

Garreta, R., Moncecchi, G., Hauck, T. & Hackeling, G. (2017), *Scikit-learn: machine learning simplified: implement scikit-learn into every step of the data science pipeline*, Packt Publishing Ltd.

Gatt, T., Seychell, D. & Dingli, A. (2019), Detecting human abnormal behaviour through a video generated model, *in* '2019 11th International Symposium on Image and Signal Processing and Analysis (ISPA)', IEEE, pp. 264–270.

Ghate, V. N. & Dudul, S. V. (2010), 'Optimal mlp neural network classifier for fault detection of three phase induction motor', *Expert Systems with Applications* **37**(4), 3468–3481.

Ghazal, S. & Khan, U. S. (2018), Human posture classification using skeleton information, *in* '2018 International Conference on Computing, Mathematics and Engineering Technologies (iCoMET)', IEEE, pp. 1–4.

Griffith, H., Hajiaghajani, F. & Biswas, S. (2017), Office activity classification using first-reflection ultrasonic echolocation, *in* '2017 39th Annual International Conference

of the IEEE Engineering in Medicine and Biology Society (EMBC)', IEEE, pp. 4451–4454.

Haslegrave, C. M. (1994), 'What do we mean by a 'working posture'?', *Ergonomics* **37**(4), 781–799.

Heaton, J. (2016), An empirical analysis of feature engineering for predictive modeling, *in* 'SoutheastCon 2016', IEEE, pp. 1–6.

Hu, X. (2019), *Intelligent Biomechatronics in Neurorehabilitation*, Academic Press.

Jo, B. & Kim, S. (2022), 'Comparative analysis of openpose, posenet, and movenet models for pose estimation in mobile devices', *Traitement du Signal* **39**(1), 119–124.

Kaushik, V. & Charpe, N. A. (2008), 'Effect of body posture on stress experienced by worker', *Studies on Home and Community Science* **2**(1), 1–5.

Khurana, U., Turaga, D., Samulowitz, H. & Parthasrathy, S. (2016), Cognito: Automated feature engineering for supervised learning, *in* '2016 IEEE 16th International Conference on Data Mining Workshops (ICDMW)', IEEE, pp. 1304–1307.

Kirasich, K., Smith, T. & Sadler, B. (2018), 'Random forest vs logistic regression: binary classification for heterogeneous datasets', *SMU Data Science Review* **1**(3), 6–9.

Korakakis, V., O'Sullivan, K., O'Sullivan, P. B., Evagelinou, V., Sotiralis, Y., Sideris, A., Sakellariou, K., Karanasios, S. & Giakas, G. (2019), 'Physiotherapist perceptions of optimal sitting and standing posture', *Musculoskeletal Science and Practice* **39**, 24–31.

Kumar, S. S., Dashtipour, K., Gogate, M., Ahmad, J., Assaleh, K., Arshad, K., Imran, M., Abbai, Q. & Ahmad, W. (2021), Comparing the performance of different classifiers for posture detection, *in* 'EAI'.

Kwon, B., Huh, J., Lee, K. & Lee, S. (2020), 'Optimal camera point selection toward the most preferable view of 3-d human pose', *IEEE Transactions on Systems, Man, and Cybernetics: Systems* **52**(1), 533–553.

Osokin, D. (2018), 'Real-time 2d multi-person pose estimation on cpu: Lightweight openpose', *arXiv preprint arXiv:1811.12004* .

Pishchulin, L., Andriluka, M., Gehler, P. & Schiele, B. (2013), Strong appearance and expressive spatial models for human pose estimation, *in* 'Proceedings of the IEEE international conference on Computer Vision', pp. 3487–3494.

pose estimation — tensorflow lite (2022).

URL: *https://www.tensorflow.org/lite/examples/pose*$_e$*stimation/overview*

Poushter, J. et al. (2016), 'Smartphone ownership and internet usage continues to climb in emerging economies', *Pew research center* **22**(1), 1–44.

Raju, J., Reddy, Y. C. et al. (2020), Smart posture detection and correction system using skeletal points extraction, *in* 'Advances in Decision Sciences, Image Processing, Security and Computer Vision', Springer, pp. 177–181.

Sarafianos, N., Boteanu, B., Ionescu, B. & Kakadiaris, I. A. (2016), '3d human pose estimation: A review of the literature and analysis of covariates', *Computer Vision and Image Understanding* **152**, 2.

sklearn.linear_model.logisticregression (n.d.).

URL: *https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html*

sklearn.neural_network.mlpclassifier (n.d.).

URL: *https://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPClassifier.html*

sklearn.svm.svc (n.d.).

URL: *https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html*

stereo toolbox (2022).

URL: *https://eo.belspo.be/en/stereo-toolbox*

Syrop, M. & Bach, J. R. (1990), 'A study of sitting posture variations using the nose and mandible as reference points', *CRANIO®* **8**(3), 258–263.

Verdonck, T., Baesens, B., Óskarsdóttir, M. et al. (2021), 'Special issue on feature engineering editorial', *Machine Learning* pp. 1–12.

Woodall, P., Borek, A., Gao, J., Oberhofer, M. & Koronios, A. (2014), An investigation of how data quality is affected by dataset size in the context of big data analytics.