ORTHODOX

Technical Guide

Killian Connolly - 17303116
William John O’Hanlon - 17477494
Supervisor - Prof Tomas Ward

Table of Contents

1. Introduction

1.1 Overview

Orthodox MMA trainer is a web application which utilises pose estimation and computer vision to predict a user's technique / form in regards to certain strikes used within the sport of mixed martial arts, such as a jab. The goal of this project is to provide an inexpensive way for MMA enthusiasts and beginners alike to practice their striking technique. This is mainly aimed at beginners as it is important that they have good fundamentals before moving on to more complex techniques. It is important to note that the goal of this project is not to replace in person coaching, but to act as a tool for people with limited access to a MMA gym regularly or to be used in conjunction with in person training.

The web application accepts video as input and utilises Google's latest lightweight pose estimation solution, BlazePose [1], to abstract the position of the user's body in said video. These body landmarks are then fed into multiple classifiers, based on the inputted strike, and a feedback report is generated and displayed to the user.

The web app is hosted on AWS at the domain https://orthodoxmma.com.

1.2 Motivation

This project was motivated by many things. The first motivation came with the closing of gyms and sports clubs during the first lockdown. We had to find alternative ways of exercising that didn't involve going to the gym or leaving the home outside of the 5km radius restrictions. Mixed martial arts (MMA) training provided an exercise outlet which was both cardio-intensive and achievable at home. The strikes and movements required little to no equipment, as one could shadow box in place of hitting a bag or pads. Normally, this sort of training would take place in a MMA or kickboxing gym, with a trainer supervising technique. However, this sort of oversight was not possible due to the then ongoing pandemic, and so form and technique had to be derived from online resources without professional direction and clouded by personal bias. This led, along with other things, to the formation of the idea of producing a machine learning tool which could be trained to evaluate / predict striking form effectively, in place of in-person coaching.

Moreover, MMA is a sport of great interest to us, and one which has grown massively in the past few years. This project presented a great opportunity to learn more about the sport as well as improve upon our machine learning capabilities. This motivation was further cemented by the public release of Google's pose estimation pipeline Mediapipe BlazePose [] in late 2020, as it provided a cutting edge machine learning technology to learn and implement.

1.3 Glossary

API - Application Programming Interface
Golang - Open source programming language developed by Google.
BlazePose - Open source human pose estimation solution provided by MediaPipe. Also referred to as MediaPipe Pose.
ML - Machine Learning.

2. Research

2.1 Literature Review

In order to fully comprehend the task at hand, we examined the importance of proper technique in regards to striking in combat sports such as MMA and boxing. A survey into injuries in Thai Boxing identified incorrect technique as the leading cause of injury and identified it as an underrated problem in combat sports [2]. Moreover, proper striking technique was found to be one of the main injury prevention measures in preventing hand injury in boxing [3].

Most of the research examined centred around injuries sustained from striking technique in training, sparring and competitions, and not shadowboxing [4]. However, the goal of this project is to provide users with striking technique feedback, to prevent injuries in in-person training, such as pad work and sparring. This research reinforced our emphasis on technique prediction, as improving the user's striking technique will indirectly protect them from injury when applied in a gym in person setting.

Research was also conducted into similar projects. It was found that inertial sensors have been used to successfully classify striking types and strike technique [5]. These are sensors which measure the acceleration and angular velocity of an object along three mutually perpendicular axes. Two sensors were placed within the participants gloves, another was placed in a specially designed sports harness on the participants' backs. The data collected was used to train machine learning classifiers which achieved an overall accuracy of up to 96%. While this is an impressive metric, a motivation behind our project was to provide an inexpensive way for individuals to evaluate their strike technique, and inertial sensors is not a technology widely available to the average person.

2.2 Requirements Gathering

Before starting work on our project we had to gather requirements for each of the project's components. The research done is outlined in this section.

2.2.1 Striking

In order for us to classify whether a user is performing a certain strike correctly or incorrectly, we first need to gather information on what exactly is considered correct and incorrect technique.

To do this, we started by contacting a few people who are involved in the MMA community in our local vicinity. One of the places we reached out to for information was the Black Dragon Kickboxing Gym in Cavan. We conducted an informal interview with a representative at the gym. We inquired about general errors in technique common among newcomers in the sport, in regards to the jab punch, the cross / straight punch and stance. They provided us with a plethora of valuable information. Importantly, they emphasised that beginners should learn good fundamentals first, in regards to stance and striking, ensuring to effectively protect themselves by covering the chin and body and not overcommitting to strikes, as beginners have a tendency to do. For example, in regards to the jab, it is important to cover the chin with the shoulder of the jabbing arm when striking.

We also made use of available online resources to learn more about improving technique for beginners. There is a lot of information and online resources available, and these resources coupled with the information we received from the Black Dragon Kickboxing Gym helped us make informed decisions for building datasets and classifiers.

2.2.2 Pose Estimation

Prior to starting the project, we concluded that existing pose estimation technology could be used to facilitate our project. This is a very important aspect of the project as without it we would be unable to determine the position of a user’s body landmarks effectively in order to predict their striking technique.

After some research on the available technology, various pose estimation solutions were discovered, such as OpenPose [6], AlphaPose [7] and BlazePose [1]. We concluded on using BlazePose over the aforementioned solutions. The deciding factor was that BlazePose is a new lightweight solution which can be run on mobile devices, allowing for offline app usage. Also with BlazePose only being released last year it seemed like an exciting bit of technology that we really wanted to learn more about.

2.2.3 Datasets

Intuitively, when training supervised machine learning algorithms it is necessary to have labelled datasets. As such, research was conducted on publicly available human action datasets which would contain labelled data similar to our use case. Datasets such as the KTH dataset [8] and the G3Di dataset [9] were considered. However, while some of these datasets contained labelled boxing video data, generally the form within the video dataset was not optimal, as the datasets were not built with this project’s use case in mind. Therefore, it was concluded that custom datasets had to be created for predicting each individual technique e.g. separate dataset for jab and separate dataset for stance. This would allow us to have full control over the machine learning process.

2.2.4 Web Application

Originally we wished to create a mobile application, however, after consulting with our supervisor, Tomas Ward, we inferred that creating a progressive web application would be best suited for our project. In doing so, the same application could be used on both mobile and desktop environments. Moreover, this enabled more time to be spent focusing on creating accurate models.

We researched the best possible way to implement a web application with our criteria and concluded that a Python Flask application would be an ideal solution for us. This would allow for a seamless connection between our backend classifier handlers and our web application framework, as both would be written in Python. This web application could then be stored and deployed with a cloud computing provider such as an Amazon Web Services (AWS), Microsoft Azure or Google Cloud. Neither of us had experience in either cloud computing solution, so after some deliberation we decided on deploying on an AWS Elastic Compute Cloud (EC2) instance given that it is more widely used and has a substantial amount of documentation.

A discussion was also had about which web server provider we should use. The two main contenders were Apache and Nginx. We weren’t expecting our application to attract much traffic so the decision came down to ease of use between the two. Nginx seemed to be the superior choice.

2.2.5 Database

Being able to store users information and progress is vital for the application. After some analysis we deduced that creating an Amazon Relational Database Service(RDS) instance would be more efficient given that our Web Application will be deployed using the same Cloud Platform. SQLite database was also discussed which is a client based database. This would be able to store the users feedback report. However for the likes of users credentials this would not be ideal. Therefore, we decided against using SQLite. We also considered using other cloud based databases, such as Firebase, however, we dismissed this idea due to wanting to learn the more widely used Amazon Web Service which would integrate well with our EC2 instance.

3. Design

3.1 High Level System Architecture

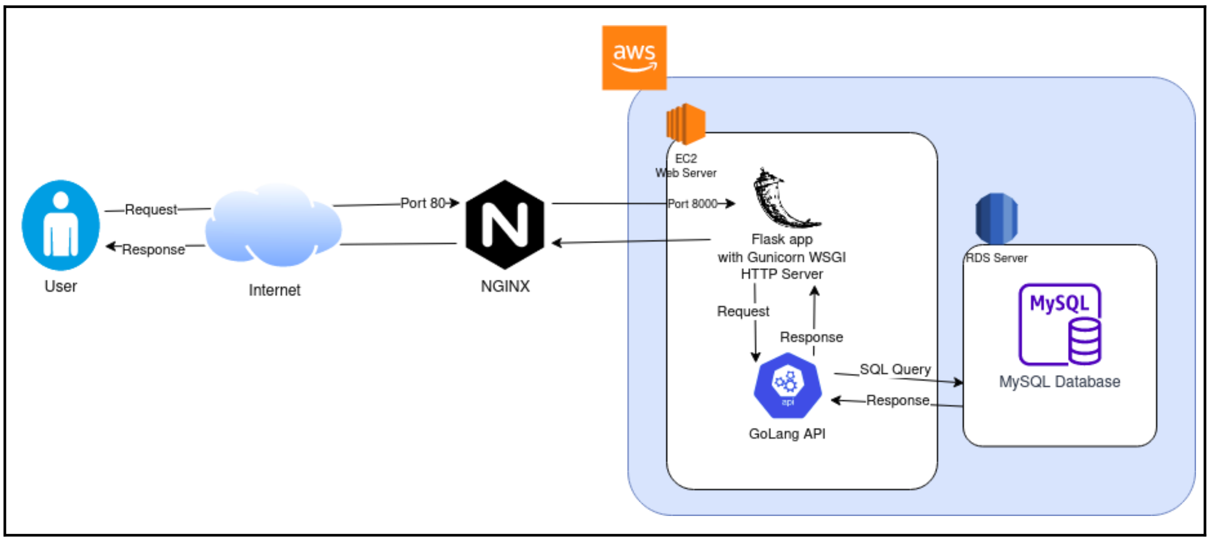


Fig. 3.1.1 High level system architecture of web application components

3.2 Architecture of Form Prediction Feature

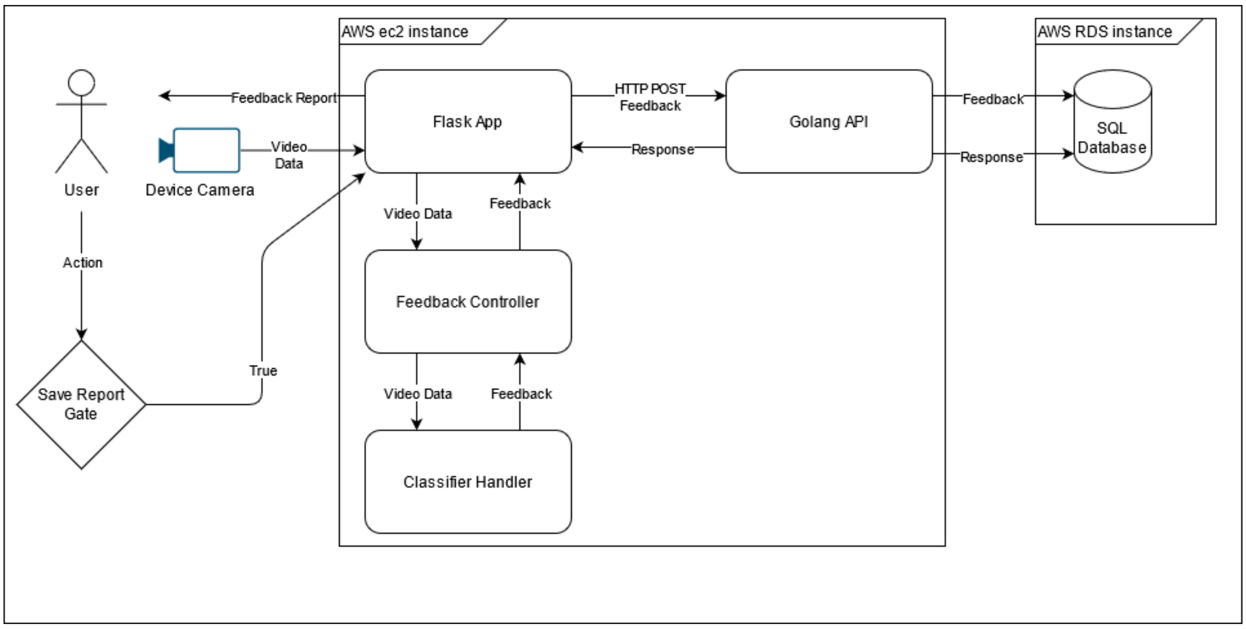


Fig. 3.2.1 High level architecture of form prediction feature within application

The main component of the project is the form prediction function. The architecture of this function is displayed above. The user submits a video to the flask app through the frontend, either by way of a file upload or through a live recording facilitated by the web application. If the file upload option is chosen the file extension is checked to ensure that the file is a .mp4 file. This video is passed to the feedback controller, which feeds the data to the classifier handler.

The classifier handler is a program which is specific to the strike technique in which it is predicting. This program uses BlazePose to extract the body landmarks from the user submitted video frames, converting them to numpy arrays and passing them to the classifiers. This subsequently returns the predictions to the feedback controller in the form of a feedback dictionary. The feedback controller uses this dictionary to build a feedback report, which is displayed to the user through the Flask App. A more detailed diagram of how the feedback controller and classifier handler interact is displayed below.

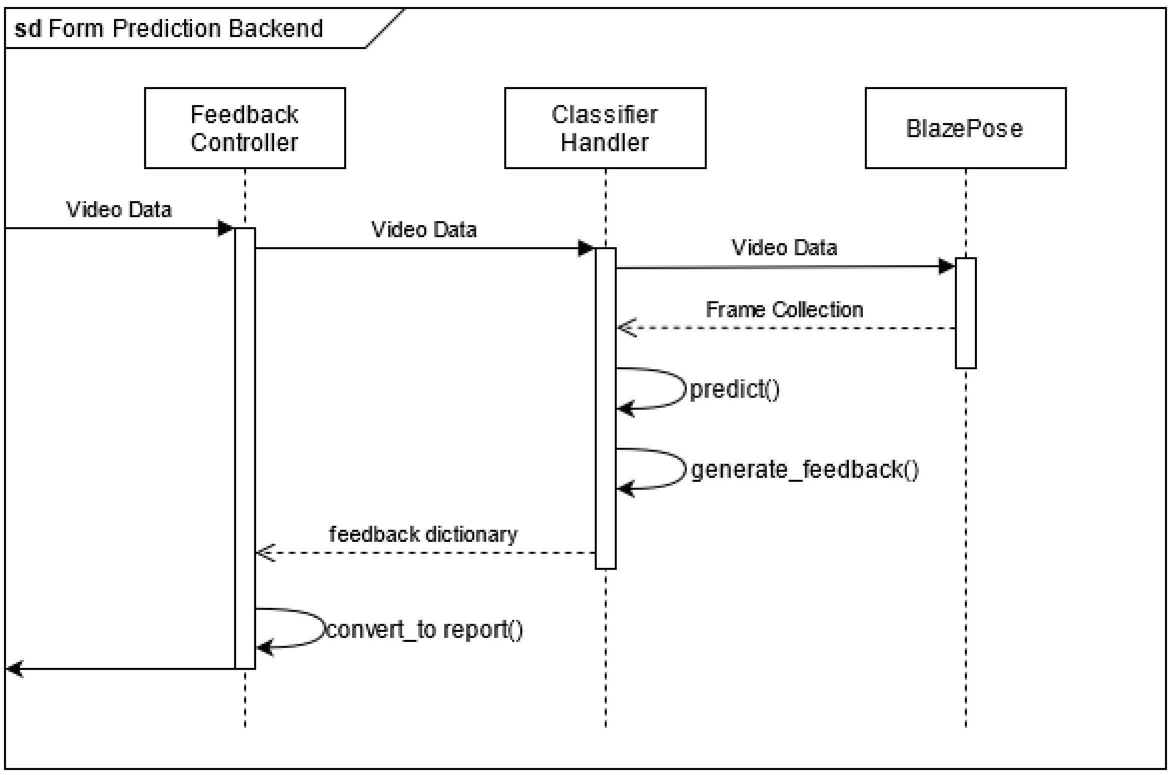


Fig. 3.2.2 Sequence diagram of form prediction

4. Implementation

4.1 Datasets

4.1.1 Dataset Dimensionality

Before the datasets were created, we evaluated what type and format of data the dataset would contain. There are many factors to consider when evaluating the form of a strike, such as a jab. For this reason, there were many possible classes and combinations of classes to be contained with a dataset. For example, we identified five main contributors to bad jabbing technique, namely:

- Failing to sufficiently cover the chin with the left shoulder when striking.
- Not protecting the body with the non jabbing arm.
- Leaving the chin exposed on the non jabbing arm side, by failing to cover the chin with lead hand.
- Overcommitting to the jab by leaning too far forward.
- Allowing the jabbing arm elbow to come in from the side and not from underneath the jab.

Therefore there are 120 (5!) classes which contain somewhat incorrect form, meaning the jab dataset would have to contain balanced data on 121 classes, including the correct form class, to train a single multi-label classifier. Building a dataset of this dimensionality and complexity would be a project in itself, and so the dimensionality had to be reduced.

To reduce the dimensionality and complexity, datasets were broken up based on the contributing factors to the technique of the strike the dataset represents. In other words, a dataset was produced for each of the technique contributors, with a positive class representing the contributor performed correctly, and the negative class representing the contributor performed incorrectly. Compartmentalising the dataset this way allowed for a single classifier to be replaced with a binary classifier for each contributing factor of a strike's technique. This method seemed more feasible, and allowed for models to be trained on less data. As such, the main contributors of a strike had to be determined, and a labelled dataset created for each of these contributors.

4.1.2 Dataset Creation Process

When actually creating the dataset, many steps had to be followed. The raw form of the datasets consists of .mp4 video files filmed by ourselves, at a side on angle in the orthodox boxing stance. These videos were shot over a number of days, in slightly different environments, allowing for inter-session variability in the dataset.

The video files had to be placed within a certain folder structure for the programs to correctly locate the data. Once called, the dataset creation program (`make_dataset_numpy.py`) passes the local path of the video files, taken off the command line, to BlazePose, which subsequently returns a Frame Collection object after pose estimation is completed. A Frame Collection object is a custom object which contains rows of 33 tuples corresponding to the 33 body landmarks predicted by BlazePose for each frame in the inputted video. The tuples contain the x and y coordinates of the predicted body landmark, as well as a visibility metric which indicates the predicted likelihood of the landmark being visible. Any rows which contain landmarks with unsatisfactory levels of visibility are removed from the object (below 80% visibility).

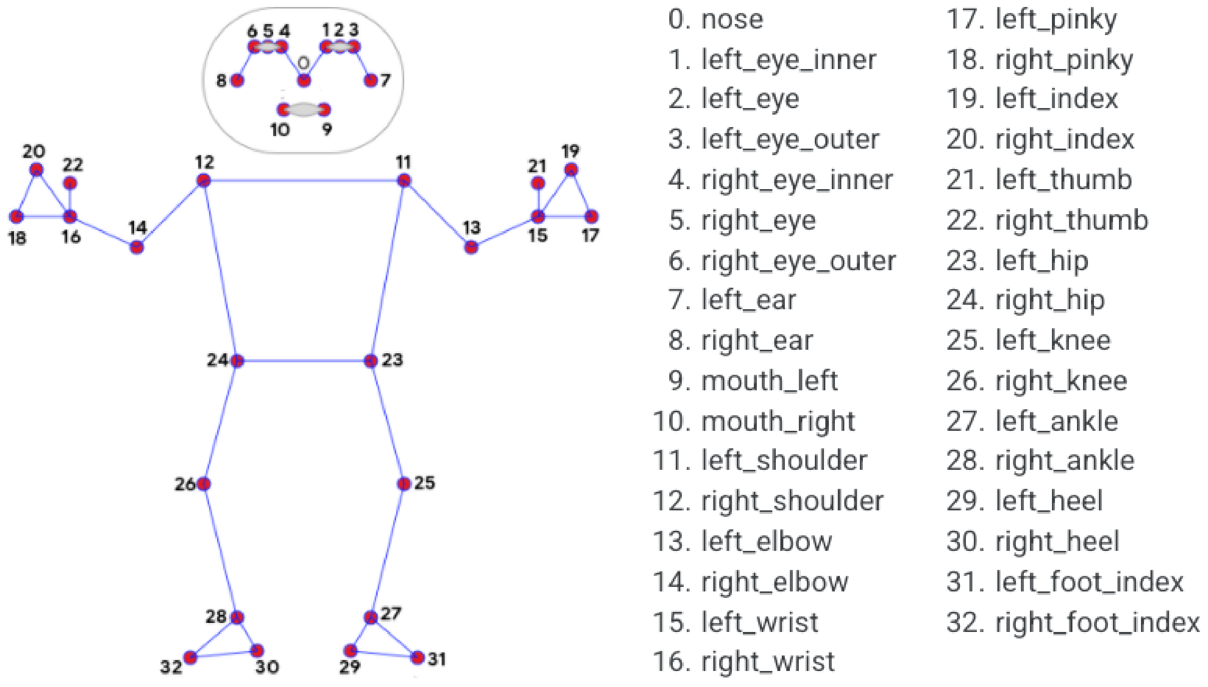

0. nose
1. left_eye_inner
2. left_eye
3. left_eye_outer
4. right_eye_inner
5. right_eye
6. right_eye_outer
7. left_ear
8. right_ear
9. mouth_left
10. mouth_right
11. left_shoulder
12. right_shoulder
13. left_elbow
14. right_elbow
15. left_wrist
16. right_wrist
17. left_pinky
18. right_pinky
19. left_index
20. right_index
21. left_thumb
22. right_thumb
23. left_hip
24. right_hip
25. left_knee
26. right_knee
27. left_ankle
28. right_ankle
29. left_heel
30. right_heel
31. left_foot_index
32. right_foot_index

Fig. 4.1.1 Body landmarks predicted by BlazePose

Moreover, another step which may be necessary when making a dataset is to remove the unwanted body landmarks. When the dataset creation program is called, a list of landmark indices can be optionally passed to the program. If this is the case only the body landmarks which correspond to these indices will be added to the dataset. This way tailored datasets can be created for predicting certain aspects of technique. For example, when creating the dataset for whether the shoulder of the jabbing arm protects the chin throughout a jab strike, only the body landmark values which correspond to the shoulders and the lower half of the face were added to the dataset. This ensures that the classifiers learn from the relevant features instead of other features which may be inconsistent due to them not being the focus of the dataset. For a reference as to which indices correspond to which landmarks, please refer to the figure above. This removal of landmarks occurs asynchronously with the removal of frames with low visibility.

After this step, each row in the Frame Collection object is then concatenated with its respective label, taken from the name of the directory the video was stored in, and added to a multi-dimensional numpy array. The numpy array is then corrected for class imbalance. As the datasets were produced from video data, it is unlikely that both class labels in a dataset contain the same number of examples so this is a necessary step. If the dataset was not corrected for class imbalance then the data would be biased / skewed in the direction of the oversampled class. This could hinder the classifiers ability to learn properly in training and result in the model having a poorer predictive accuracy over the minority class, in comparison to the majority class.

Furthermore, these x and y coordinates had to be normalised, as they were reliant on the dimensions of the inputted frames. For example, if part of the dataset was abstracted from a 1920x1080 video, then the x coordinates could be in the range of 0 to 1920, while another part of the dataset could be abstracted from a 1280x720 video with x coordinate values ranging from 0 to 1280. Besides from being good practice, normalization allows for our classifiers to handle video input from a wide range of camera dimensions. The values outputted from normalisation are between -100 and 100.

Finally, the dataset has been fully prepared and serialised to file. A summary of the process is described in the diagram below.

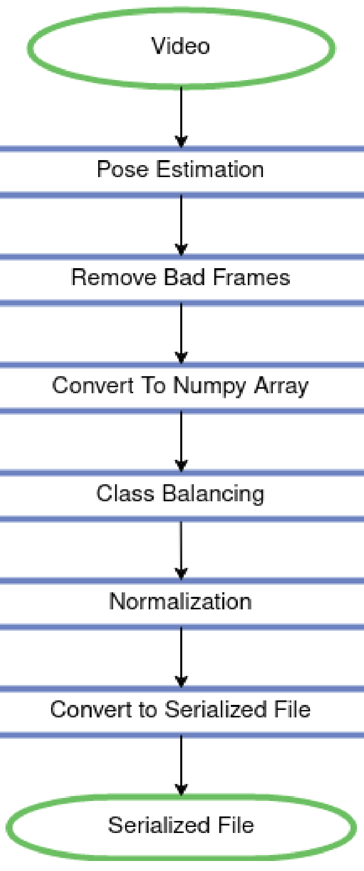


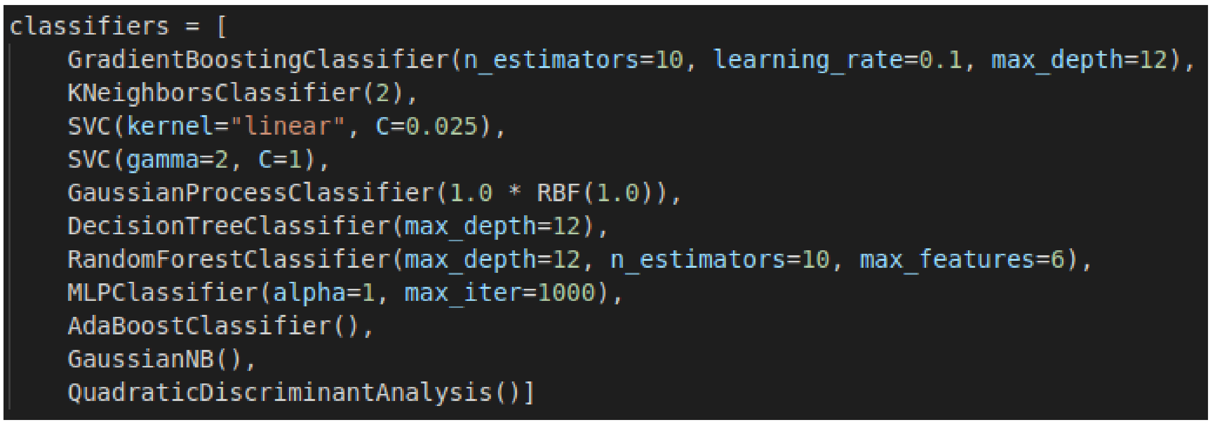
Fig. 4.1.2 FlowChart of Dataset Creation Process

4.2 Classifiers

We require machine learning classifiers for our system to predict whether a user is performing a strike correctly. This section will describe the process we took to training our classifiers and what classification algorithms we used.

4.2.1 Training Classifiers

A model benchmark program was produced to determine which of the different classifiers provided by scikit-learn had the highest validation set accuracy after training on a particular dataset. The model(s) with the highest validation accuracy were tested with an unseen testing set. This testing dataset was taken in a different environment across different days for intersession variability. This process was repeated for each contributing feature of a strike's form. Once the best classifier was determined for predicting a dataset, the dataset was loaded into a different program which retrained the selected scikit-learn classifier on the data and saved the model to file after validation.



```
classifiers = [
    GradientBoostingClassifier(n_estimators=10, learning_rate=0.1, max_depth=12),
    KNeighborsClassifier(2),
    SVC(kernel="linear", C=0.025),
    SVC(gamma=2, C=1),
    GaussianProcessClassifier(1.0 * RBF(1.0)),
    DecisionTreeClassifier(max_depth=12),
    RandomForestClassifier(max_depth=12, n_estimators=10, max_features=6),
    MLPClassifier(alpha=1, max_iter=1000),
    AdaBoostClassifier(),
    GaussianNB(),
    QuadraticDiscriminantAnalysis()]
```

Fig. 4.2.1 All Scikit-Learn classification algorithms tested within our benchmark.

Using Jab Classification as an example, it required five binary classifiers. Namely, arm protection, elbows out, chin protection, elbow protection and finally overcommitting detection. To decide on which algorithm to use, we used the aforementioned model benchmark. After training the five binary classifiers separately, it was clear that the k-Nearest Neighbour and Gradient Boosting classification algorithms received the highest prediction accuracy across the classifiers. Additionally, we took into account F1 scores of the model. The results from these model benchmarks are available within our Gitlab repository: `src\Model_Results` , as well as in our testing document.

4.2.2 Classifiers Results

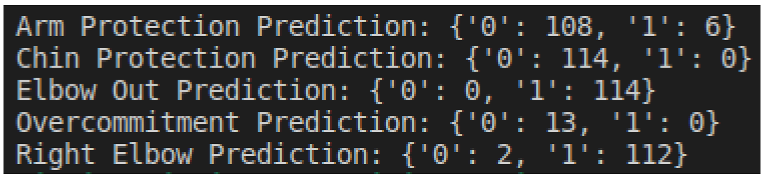
The resulting binary classifiers for the jab strike is summarised below.

	Classifier	Validation Accuracy	F1 Scores	Testing Accuracy
Jab Elbow	KNN	0.98	0.98	0.6111
Over Commitment	GradientBoosting	1.0	1.0	1.0
Arm Protection	KNN	1.0	1.0	1.0
Chin Protected	KNN	1.0	1.0	1.0
Right Elbow In	KNN	0.99	1.0	1.0

Fig.4.2.2 Table of all classifiers and their results

These resulting binary classifiers were combined into a single program which could take in a video frame, along with the label of the strike being performed, and output a prediction of the strike form in the frame.

However, the classifiers were trained on the body landmark data extracted from video frames and not a complete video sequence of a strike. As such, the program had to be modified to allow the classifiers to predict sequential video frames. A simple form of moving average filtering was implemented, whereby each frame of a strike video sequence is fed to each binary classifier and the results saved to a dictionary. The median label prediction of the classifier is taken to be the overall prediction. Typically, the simple moving average is taken to be the mean of the data-points, however as the output of the binary classifiers is either 0 or 1, the median had to be taken to ensure that the overall prediction was one of these labels. This filtering insured against model flickering and also facilitated video classification.



```
Arm Protection Prediction: {'0': 108, '1': 6}
Chin Protection Prediction: {'0': 114, '1': 0}
Elbow Out Prediction: {'0': 0, '1': 114}
Overcommitment Prediction: {'0': 13, '1': 0}
Right Elbow Prediction: {'0': 2, '1': 112}
```

Fig. 4.2.3 Outputted predictions from jab classification showing predictions pre averaging.

Furthermore, some classifiers required preprocessing in practice. For example, the classifier responsible for detecting overcommitment for a jab only predicts on certain frames in which the user has fully extended their arm.

4.3 Application Programming Interface

The implementation of our API was created in Go language (Golang). We chose this language because we were interested in learning a different language that we haven't programmed in before and this project provided a great opportunity to learn the language.

Our API contains two routes groups, one for handling user credentials and another for handling user feedback reports. We decided on implementing a Data Access Object(DAO) Pattern approach using Golangs Object Relational Mapping(ORM) library as an instance of our database. The HTTP web framework was supplied via the gin library.

The database was created on an AWS Relational Database Service(RDS) instance and contains two tables, credentials and feedback. The API has two separate routing groups for these groups. When requests are made, golang structures are instantiated, using the aforementioned ORM library, for the respective requests. These structures can then be used to communicate with the database.

4.3.1 Credentials Database

This database was required to hold login credentials for the user. A simplistic approach was taken, meaning, solely email and password attributes are logged in the database, with email being the Primary Key. Requests to the database are under the routing group `user-api`. The Credentials Data Access Object deals with all communication to the Credentials database.

Firstly, when a user registers an account with the web application, a POST request is sent via the CreateUser method, given json of their submitted email and encrypted password. In order for a user to successfully register, a unique email address is required. This is checked by way of a GET request to `GetUserByEmail`. If a 404 response is returned from this request, no account exists with this email, and registration can continue. After this, the password is checked to ensure that an empty string password has not been submitted. If anything unexpected happens during the registration process, the process is aborted and a custom message is displayed to the user on the registration page explaining the error.

Moreover, once registered the user can navigate to the settings page to change their password or delete their account, managed by a PUT and DELETE request respectively.

Command	URL	Explanation
GET	/user-api/user	GetUsers
POST	/user-api/user	CreateUser
GET	/user-api/{email}	GetUserByEmail
PUT	/user-api/{email}	UpdateUser
DELETE	/user-api/{email}	DeleteUser

Fig. 4.3.1 Credentials Database Requests

4.3.2 Feedback Database

Feedback database contains the feedback reports received from the report generator. Once again a simplistic approach was taken in the way that email address and the report itself are the two attributes associated with the database. The API has a second routing group for requests to this database, namely, `user-api-feedback`. GetAllFeedbacks is a GET request which obtains all feedback reports given the email address in the url of the request. The POST request, CreateFeedback, is also required for when the user wishes to save the report they received.

Command	URL	Explanation
GET	/user-api-feedback/{email}	GetAllFeedbacks
POST	/user-api-feedback/user	CreateFeedback

Fig. 4.3.1 Feedback Database Requests

4.4 Web Application

4.4.1 Flask application

We implemented the web application in python using Flask, a python micro web framework. The web application follows a RESTful architecture, with each component communicating through HTTP requests and responses. The app comprises a total of 16 web pages including the HTTP error handler pages. The routing between these pages is all handled within the main `app.py` program. Each endpoint accepts HTTP request methods and returns either the appropriate web page or JSON response depending, along with a HTTP status code. Requests to databases are also handled in the main program.

The aim of the overall aesthetic of the website was to be minimalistic and easy to read and navigate, from a frontend perspective. Moreover, being able to have the same experience on desktop as well as mobile was something we wanted to achieve.

As previously mentioned, we intended on making the app a progressive web application (PWA). While we did achieve this in some respects, such as the application working on any platform which has a standards compliant browser, the application is still not installable as a progressive web app should be.

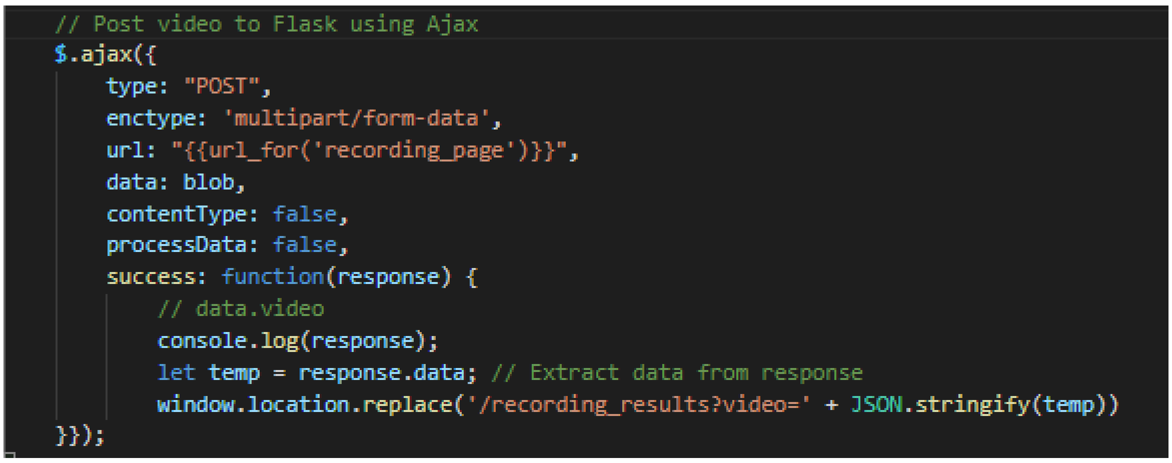
4.4.2 Security

Prioritising the security is of utmost importance to us as developers. Firstly, users' access to the web application is reliant on whether they have the correct login credentials. The password for the credentials is encrypted via werkzeug security. Flask contains a session secret key which is used to sign session cookies for protection against cookie data tampering, this key is generated randomly. Using this we were able to apply session based authentication to our application.

Moreover, the configuration credentials for the access of our RDS instance must be kept hidden from the public. For this reason we concluded on privatising these configuration files since it would be difficult for us to encrypt these files without showing the source code for the decryption process either on GitLab or on our instance.

4.4.3 Web Camera

Again, the web application gives the user the option to either upload pre-recorded video or send a live recording from their local device camera for form prediction. To facilitate this functionality, the MediaStream Recording API [10] was used. This is a web API which allows for the capture of data streams from a device’s microphone and / or cameras given the user’s consent. This was implemented in Javascript. Once the recording begins, a MediaRecorder object is created and passed the video stream. This object handles saving the stream data to a blob (binary large object), once video data is available. On recording completion, the blob is sent to the Flask app by way of an AJAX POST request. Once the data is received on the server-side, it is cached on the server and sent to the feedback controller which returns the feedback to the user post prediction.

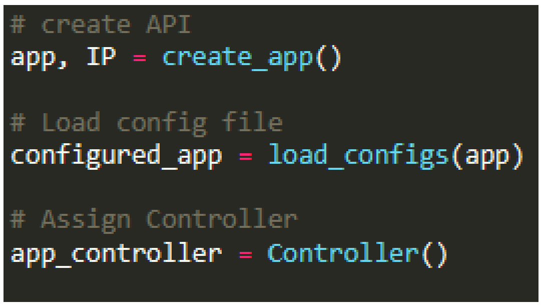


```
// Post video to Flask using Ajax
$.ajax({
    type: "POST",
    enctype: 'multipart/form-data',
    url: "{{url_for('recording_page')}}",
    data: blob,
    contentType: false,
    processData: false,
    success: function(response) {
        // data.video
        console.log(response);
        let temp = response.data; // Extract data from response
        window.location.replace('/recording_results?video=' + JSON.stringify(temp))
}});
```

Fig. 4.4.1 AJAX POST request responsible for sending video data to flask application

4.4.4 Feedback Controller

This component of the system is responsible for facilitating the prediction of the technique in a submitted video. An instance of the `Controller` class is initialised once the app is started and the config file has been loaded.



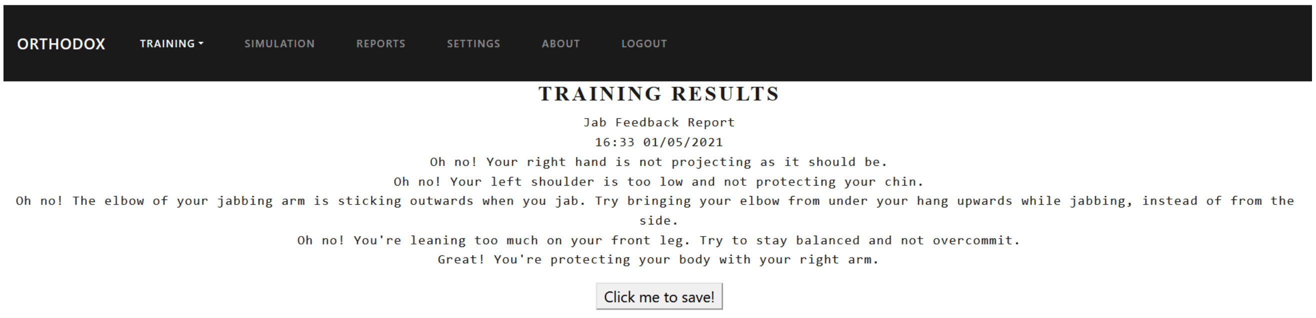
```
# create API
app, IP = create_app()

# Load config file
configured_app = load_configs(app)

# Assign Controller
app_controller = Controller()
```

Fig. 4.4.2 Web Application set up

Regardless if the video has been submitted through the live recording functionality or the upload feature, the video is temporarily cached on the server given that the video is of the right format and no errors occur. After the video data is saved, the feedback analysis method within the app feedback controller is passed the video location, as well as the location of the respective classifiers for the technique to be predicted. This method calls another program which extracts the body landmarks from the video data, using BlazePose, and subsequently inputs the landmark data into the binary classifiers. The predictions are returned to the feedback controller, in the form of a feedback dictionary. The feedback controller converts this dictionary into a readable feedback report, which is displayed to the user.



TRAINING RESULTS

Jab Feedback Report
16:33 01/05/2021

```
Oh no! Your right hand is not projecting as it should be.
Oh no! Your left shoulder is too low and not protecting your chin.
Oh no! The elbow of your jabbing arm is sticking outwards when you jab. Try bringing your elbow from under your hang upwards while jabbing, instead of from the side.
Oh no! You're leaning too much on your front leg. Try to stay balanced and not overcommit.
Great! You're protecting your body with your right arm.
```

Fig. 4.4.3 Example of returned feedback report

4.5 Deployment to Cloud Services

In order for us to successfully deploy our application, we needed to set up a web server. Firstly, we needed a cloud computing platform to set up our web server on. For this, as mentioned previously, we choose an AWS Elastic Compute Cloud (EC2) instance. We instantiated Nginx on the EC2 instance, to act as our web server. The next step was to set up a Gateway Interface HTTP server using Gunicorn and to connect our application to an SQL database for user centric functionality. For this, we choose to use an Amazon Relational Database Service (RDS) instance. This allowed for a somewhat seamless interface between our web server and our database, as they are both hosted on the AWS platform.

Furthermore, a domain name was then obtained from the domain registrar GoDaddy i.e. orthodoxmma.com. An SSL certificate was then secured for this domain using Certbot and AWS Route 53.

.

The process of installing requirements on our instance was made easy due to our requirements.txt file.The relationship between the two Amazon Web Services, RDS and EC2 required a Virtual Party Cloud(VCP), which is a virtual network between the two instances. A set of rules, called routes, were used to determine where network traffic is directed. Originally, RDS was set to only have inbound traffic coming from the EC2 instance similar to Fig 4.5.1, however, to allow for local testing and for ease of a local set up, the RDS instance was set to be publicly accessible.

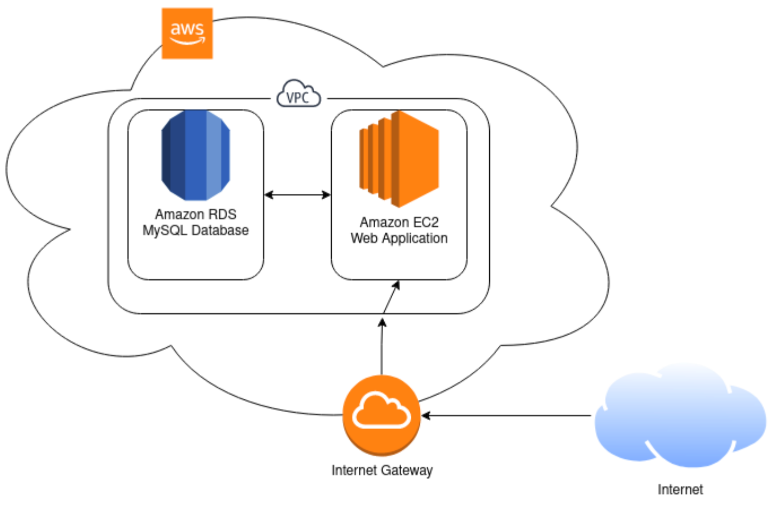


Fig. 4.5.1 High level diagram of implemented system architecture

Modifications were needed to the configuration file of nginx to allow the transfer of sizable data. This was needed for large video files to be sent from the user to our pose estimation models.

Moreover, our API is also running in the background of our EC2 server as we did not feel that there was a need for setting up another instance. Our overall development stack was LNMP(Linux Server, Nginx, MySQL and Python), a hybrid of the LAMP stack. The EC2 instance was an Amazon Linux Operating System, an operating system that we were not familiar with prior to working with it.

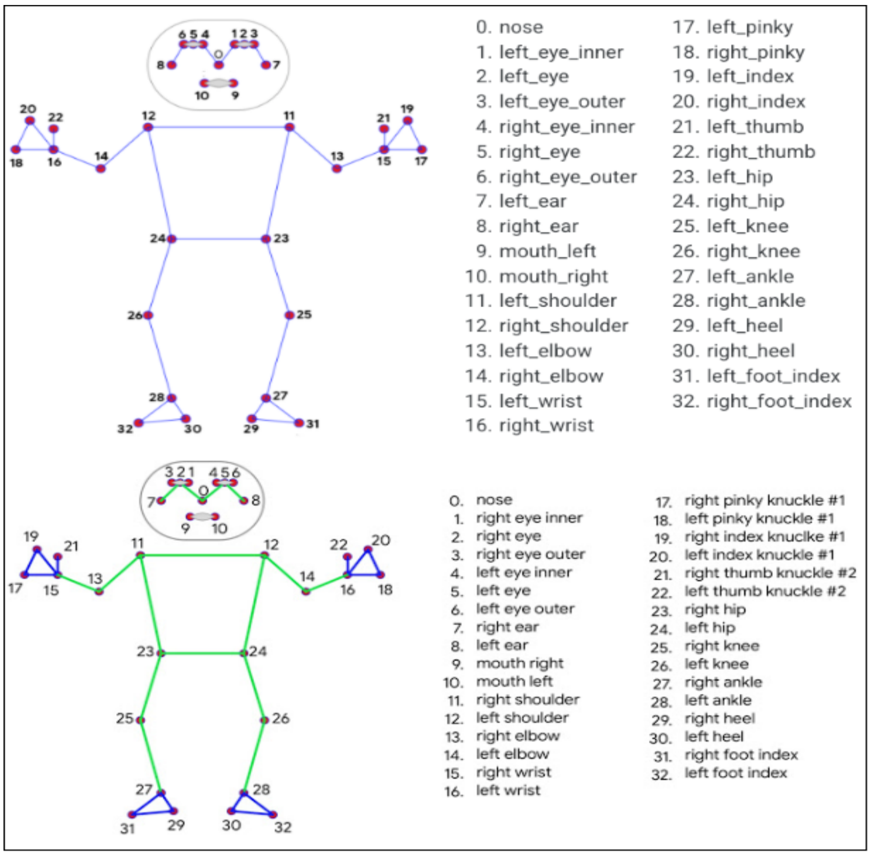
All scenarios were taken into consideration while creating the instances, hence, a “snapshot” of the instance data is taken at regular intervals(once a week) so little data is lost in the hypothetical case that one of the instances goes down. Snapshots are incremental backups.

Secure Socket Layer was required for the access of the users video camera. Privacy is, and will always be, utmost important from a users point of view and to us, hence, every recorded video is immediately deleted post analysis along with every uploaded video. We only keep records of the feedback received from our models.

5. Problems and Solutions

Problem:

When starting the project, MediaPipe's pose estimation machine learning pipeline BlazePose was relatively new. The first version of their BlazePose Pose Estimation technology was only released on the 13th of August 2020. This made integration of the pose estimation pipeline into our project difficult as there was little to no documentation, as BlazePose was still in the development phase. For example, we had difficulty determining which of the landmarks obtained from BlazePose corresponded to the left-side of the participants body and which landmarks corresponded to the right-side of the body. This confusion originated from a discrepancy between two diagrams which both displayed the indices of the landmarks according to BlazePose. One of the diagrams was taken from the MediaPipe Pose docs [1] and another was taken from an article on MediaPipe and BlazePose from the official Google AI blog [11]. The differences between the two are shown below. Notice how the top image labels the left extremities with odd numbers while the bottom image labels them with even numbers, and vice versa for the right side extremities.



0. nose	17. left_pinky
1. left_eye_inner	18. right_pinky
2. left_eye	19. left_index
3. left_eye_outer	20. right_index
4. right_eye_inner	21. left_thumb
5. right_eye	22. right_thumb
6. right_eye_outer	23. left_hip
7. left_ear	24. right_hip
8. right_ear	25. left_knee
9. mouth_left	26. right_knee
10. mouth_right	27. left_ankle
11. left_shoulder	28. right_ankle
12. right_shoulder	29. left_heel
13. left_elbow	30. right_heel
14. right_elbow	31. left_foot_index
15. left_wrist	32. right_foot_index
16. right_wrist	

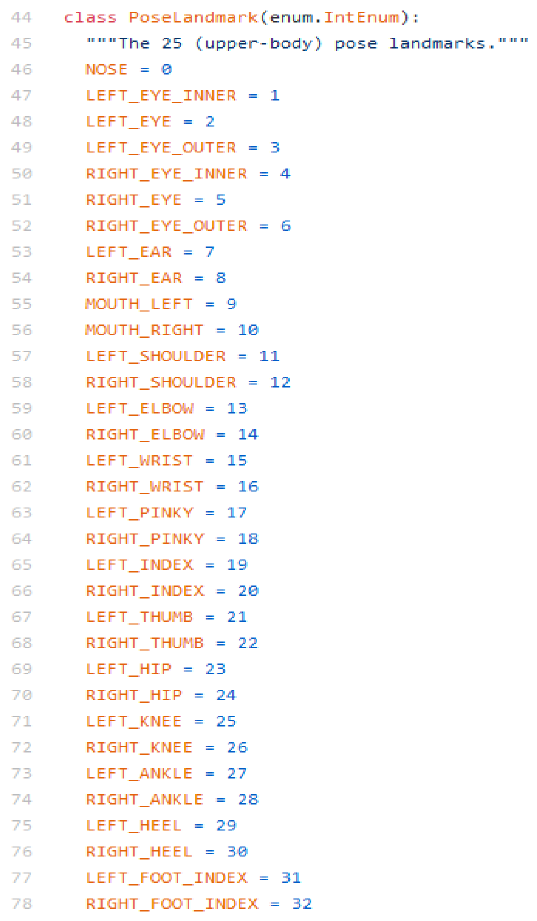
0. nose	17. right pinky knuckle #1
1. right eye inner	18. left pinky knuckle #1
2. right eye	19. right index knuclke #1
3. right eye outer	20. left index knuckle #1
4. left eye inner	21. right thumb knuckle #2
5. left eye	22. left thumb knuckle #2
6. left eye outer	23. right hip
7. right ear	24. left hip
8. left ear	25. right knee
9. mouth right	26. left knee
10. mouth left	27. right ankle
11. right shoulder	28. left ankle
12. left shoulder	29. right heel
13. right elbow	30. left heel
14. left elbow	31. right foot index
15. right wrist	32. left foot index
16. left wrist	

Fig. 5.1 BlazePose 33 keypoint topology taken from MediaPipe docs and BlazePose 33 keypoint topology taken from Google AI Blog

Solution:

In the beginning, in order for ourselves to get a reasonable understanding of the BlazePose and how to the pipeline we referred to their documentation. However, the documentation was sparse and didn't contain examples on how to extract the body landmarks, instead only showing how to display the results with OpenCV. Therefore, we resorted to examining their Github repository for the source code, in order to determine how to manipulate the objects returned from BlazePose. This allowed us to build a program, `python_video_stream_pose_estimation.py`, which could extract the landmarks from a video frame using BlazePose, instantiate our own Body Landmark class with these landmarks, and add this instantiated object to a Frame Collection object. This would be repeated for each frame in a video, with the Frame Collection object being returned after each frame has been analysed.

To fix the issue between the conflicting diagrams, we again looked through the MediaPipe Github repository for answers. We came across a piece of code, shown below, which clarified that the diagram taken from the MediaPipe docs displayed the correct topology.



```
class PoseLandmark(enum.IntEnum):
  """The 25 (upper-body) pose landmarks."""
  NOSE = 0
  LEFT_EYE_INNER = 1
  LEFT_EYE = 2
  LEFT_EYE_OUTER = 3
  RIGHT_EYE_INNER = 4
  RIGHT_EYE = 5
  RIGHT_EYE_OUTER = 6
  LEFT_EAR = 7
  RIGHT_EAR = 8
  MOUTH_LEFT = 9
  MOUTH_RIGHT = 10
  LEFT_SHOULDER = 11
  RIGHT_SHOULDER = 12
  LEFT_ELBOW = 13
  RIGHT_ELBOW = 14
  LEFT_WRIST = 15
  RIGHT_WRIST = 16
  LEFT_PINKY = 17
  RIGHT_PINKY = 18
  LEFT_INDEX = 19
  RIGHT_INDEX = 20
  LEFT_THUMB = 21
  RIGHT_THUMB = 22
  LEFT_HIP = 23
  RIGHT_HIP = 24
  LEFT_KNEE = 25
  RIGHT_KNEE = 26
  LEFT_ANKLE = 27
  RIGHT_ANKLE = 28
  LEFT_HEEL = 29
  RIGHT_HEEL = 30
  LEFT_FOOT_INDEX = 31
  RIGHT_FOOT_INDEX = 32
```

Fig. 5.2 PoseLandmark names and their respective indices, taken from MediaPipe Github repository [12]

Problem:

The process of deploying an application seemed like a simple task to begin with however after commencing the initial set up it was more difficult than it seemed. Mistakenly, we created our Amazon EC2 instance on an Amazon Linux Distribution which is based on Red Hat Enterprise Linux (RHEL). This mistake brought unanticipated difficulties when attempting to install the necessary utilities that were required for our project, such as nginx and gunicorn. This became more arduous when it came time to secure a Secure Socket Layers Certificate with certbot, which provided the application SSL certs, as Certbot and it's packaging manager snap were not supported on our EC2 instance's operating system.

Solution:

This problem was vital as an SSL certificate is necessary to be able to access, stream and transmit user video camera data. After many days of research we stumbled upon a version of the certbot library that supported our operating system, and through trial and error we were able to obtain an SSL certificate for our domain.

Problem:

As previously mentioned, the web application gives the user the option to either upload pre-recorded video or send a live recording from their local camera for technique prediction. When researching ways to implement this functionality, many issues were encountered. Initially, we wanted to implement this feature within Python, for easy interacting with our Flask App and our classifier handlers. Many of the resource's online pointed to using OpenCV VideoCapture to open a video stream within python. However, OpenCV is only able to open VideoCapture streams for cameras local to the device it is being executed on, and so the code would have to be downloaded and executed on the client-side in order for this solution to work. Therefore, another solution which would not require running python scripts on the client-side had to be determined.

Solution:

______The solution to this was to use the MediaStream Recording API [10], as previously mentioned. This required programming in Javascript, which was a programming language we were not accustomed to, but one which was absolutely necessary.

Problem:

When initially training our classifiers, we were receiving unsatisfactory results i.e. low accuracy. As such we had to implement certain techniques to increase the accuracy of our models. Some of these techniques were to be implemented regardless of accuracy, but we have included them anyway for completeness.

Solution:

As mentioned throughout our Technical Specification a number of techniques were used to ultimately obtain the best accuracy for our models. Majority of the methods involve making modifications to the dataset to reduce bias and remove discrepancies.

Dataset techniques:

- Intersession variability.
- Class label balancing.
- Normalisation of datasets.
- Removing frames with low visibility from the datasets.
- Removing landmarks from datasets which are not necessary for predicting a contributor to technique e.g. only considering body landmarks associated with the shoulders and head when predicting if the chin is correctly covered during jabbing.
- Splitting a dataset for predicting a strike into a number of smaller datasets for each contributor to a strike's technique.

Classifying techniques:

- Benchmarking machine learning algorithms.
- Rolling Prediction Average post processing.

6. Future Work

6.1 Future Development

Due to time constraints, a few features that weren’t a priority for the project were placed on the back burner. As the project deadline loomed, these features ended up not being implemented. After college is complete we do hope to continue development on this project, and implement these features.

Firstly, we would like to make our application a fully progressive web application. As mentioned previously, a key feature of progressive web applications is that they are installable, however, this was not something we were able to implement due to time constraints. Implementing this functionality would allow us to provide offline functionality with the installed version. This can be easily achieved as BlazePose is a lightweight pose estimation ML solution, built to run on both mobile devices and desktops.

Moreover, we would also like to increase the security and reliability of the application. For example, implementing an email verification feature, so false email addresses cannot be used to register for our application.

Furthermore, we would like to add more variety of techniques for the user to choose from. This would include basic strikes such as a hook, a straight punch and even some kicks such as a front kick and a push kick. This would involve making more datasets and training classifiers on these datasets. This should be a relatively straightforward task as we have already implemented programs to create these datasets and train models on these datasets, all that needs to be done is to make the raw videos for a technique dataset in the first place. Also introducing different Martial Arts such as Ti Quan Do or Karate as different subsections to our website would increase the variety available to users. These could be facilitated with the use of another ML pipeline provided by MediaPipe, such as MediaPipe Hands. This is a hand and finger tracker, which could be used to produce more detailed datasets for technique prediction of more complex martial arts exercises.

On the subject of datasets, we plan on adding more data and more individuals to our existing datasets. This would increase our models ability to generalise, reducing overfitting and as a result increase our models accuracy in practice.

Finally, as described in our functional specification, we intended on implementing another feature within our application in which a user could practice offensive and defensive techniques against a simulated opponent. Unfortunately, we were unable to implement this feature. The user would position themselves in front of their device’s camera, and be given visual or audio prompts on how to perform a routine, involving both offensive and defensive techniques. After the exercise is complete, the captured video would be returned to the user with a visual overlay of an “opponent”, along with a customised feedback report. The premise of this feature would be that the user can learn to improve upon technique through non physical combat somewhat similar to a video game. We plan on implementing this feature into the application in the future.

6.2 Continuous Development

Since our application is deployed, constant maintenance and continuous development is required. As said, BlazePose is constantly updating and improving hence continual observation is needed if any improvements can be made to the accuracy of our model predictions. Also, wishfully thinking, if our application gains attraction in the future, more resources would be needed to handle the increased traffic. For example, adding a load balancer to our infrastructure would most likely be required. This leads on to the constant backups of the infrastructure if, in the unlikely but not improbable case, any of the instances shutdown due to problems out of our control.

Renewing the resources that have been instantiated would also be a necessity. These include all Amazon Web Services instances since all of the resources are Free Tiered. Not forgetting our Domain Name, othodoxmma.com, in which we currently hold a two year license, would have to be renewed if the project continues for that long of a period of time.

7. References

1. Mediapipe. 2021. *Pose*. [online] Available at: <https://google.github.io/mediapipe/solutions/pose.html>.
2. Sieńko-Awierianów, E., Orłowski, Ł. and Chudecka, M., 2021. *Injuries In Thai Boxing*. [online] Psjd.icm.edu.pl. Available at: <http://psjd.icm.edu.pl/psjd/element/bwmeta1.element.psjd-8b4cd60f-c048-4aae-b07a-66599c986570;jsessionid=1476050D1C005FE34043941D91923F91>.
3. Drury, B., Lehman, T. and Rayan, G., 2021. *Hand and Wrist Injuries in Boxing and the Martial Arts*.
4. Hand and Wrist Institute. 2021. *Sports Injuries of Hand and Upper Extremity*. [online] Available at: <https://www.handandwristinstitute.com/sports-injury-hand-wrist-doctor-dallas-fort-worth/>.
5. M. T. O. Worsey, H. G. Espinosa, J. B. Shepherd, and D. V. Thiel, “An Evaluation of Wearable Inertial Sensor Configuration and Supervised Machine Learning Models for Automatic Punch Classification in Boxing,” *IoT*, vol. 1, no. 2, pp. 360–381, Nov. 2020.
6. Worsey, M., Espinosa, H., Shepherd, J. and Thiel, D., 2021. *An Evaluation of Wearable Inertial Sensor Configuration and Supervised Machine Learning Models for Automatic Punch Classification in Boxing*.
7. SJTU Machine Vision and Intelligence Group. 2021. *CS348 Computer Vision*. [online] Available at: <https://www.mvig.org/research/alphapose.html>.
8. Csc.kth.se. 2021. *Recognition of human actions*. [online] Available at: <https://www.csc.kth.se/cvap/actions/>.
9. Velastin.dynu.com. 2021. *Gaming Datasets*. [online] Available at: <http://velastin.dynu.com/G3D/G3Di.html>.
10. Developer.mozilla.org. 2021. *MediaStream Recording API - Web APIs | MDN*. [online] Available at: <https://developer.mozilla.org/en-US/docs/Web/API/MediaStream_Recording_API>.
11. On-device, R., 2021. *On-device, Real-time Body Pose Tracking with MediaPipe BlazePose*. [online] Google AI Blog. Available at: <https://ai.googleblog.com/2020/08/on-device-real-time-body-pose-tracking.html>.
12. GitHub. 2021. *google/mediapipe*. [online] Available at: <https://github.com/google/mediapipe/blob/master/mediapipe/python/solutions/pose.py> [Accessed 2 May 2021].