

Technical Manual

Table of Contents

1. Overview	4
2. Glossary	6
3. Research	7
3.1 Keypoint detection	7
3.2 Dataset collection	7
3.3 Dataset preprocessing	9
3.4 Classification algorithms	9
3.5 Classifying unrecognised hand gestures	11
4. Design	12
4.1 Context	12
4.2 Key design decisions	13
4.3 User interface Design	15
5. Implementation	17
5.1 Programming languages	17
5.2 Dependencies	17
5.3 Dataset collection	18
5.4 Dataset preprocessing	19
5.4.1 Dataset cleaning	19
5.4.2 Data transformation algorithms	20
5.5 Classification system	22
5.5.1 Classifier pipeline overview	22
5.5.2 Classifier development	22
5.5.3 K-nearest neighbours implementation	23
5.5.4 Local outlier factor implementation	23
5.5.5 Optimising KNN and LOF	25
5.6 Network API	26
5.7 System front end	27
5.8 Installer	30
5.9 Testing	30
5.10 Continuous integration	30
6. Results	31
7. Known limitations	32
7.1 Left handed gesture detection	32
7.2 Unreliable hand detection from OpenPose	32
7.3 Performance	32
8. Future work	33
9. Appendix	34
References	34

1. Overview

1.1 Description

Jester is a web app which allows users to control their device's camera using hand gestures. User's can take photos by making a peace sign hand gesture, start video recording using a thumbs up hand gesture, and stop video recording using a palm hand gesture.

The Jester system has three main components: a classifier, an API, and a web app. The system has been designed to be easy for app users to adopt and for system administrators to install and host.

The core areas of computer science and software engineering that have been covered in the development of Jester are data mining, distributed systems, and front end development. In addition to these areas, the authors have done extensive work on testing, automation, and algorithm development.

1.2 Motivation

The Jester web app allows users to control their device without touching it. This makes taking group photos easier, it allows athletes to easily record training, and it allows people to seamlessly start and stop recording during presentations.

The potential uses of Jester go beyond the web app, as the system's loosely coupled design allows for the gesture classification API to be easily integrated into other projects. This could allow for applications in home security, controlling industrial machines in noisy environments, and limiting contact with surfaces such as light switches in hospitals to limit the spread of infections.

2. Glossary

Classifier

A system which categorises inputs into known classes.

Anomaly detection

The process of identifying outliers relative to a dataset.

Training set

A dataset used in machine learning to create a model that can predict outcomes (classifications in this case).

Out-of-distribution (OOD)

Out-of-distribution data items are dissimilar to all other data items in the training set.

Keypoints

Keypoints are areas on an image which correlate to specific points on a person's body, for example, the fingertips.

Distributed system

A software system with components that run concurrently on multiple machines.

Graphics processing unit (GPU)

A computer hardware component used for image processing and graphics.

REST API

A type of API which uses HTTP methods to allow devices to communicate across a network.

3. Research

3.1 Keypoint detection

As the authors were primarily interested in gesture recognition as a data mining problem, it was necessary to incorporate an existing 3rd party image processing system into our project to extract keypoint data from images. It is this keypoint data that Jester's algorithms manipulate and classify. There are a range of existing projects that can detect keypoints of people in images, including OpenPose and OpenVino. Several approaches to keypoint detection were tested by the authors, but OpenPose was settled on because of its out of the box support for hand keypoint detection.

3.2 Dataset collection

In order to train a classifier to recognise hand gestures, a training set of labelled hand gesture images was required. The authors decided to build a dataset of images from scratch which required research.

3.2.1 Selecting hand gestures

The hand gestures that were selected to be used in the dataset had to be:

1. Low in variability
2. Highly distinct
3. Detectable by OpenPose

Hand gestures that are low in variability look similar to all other hand gestures of the same class. For example, all peace sign hand gestures look roughly the same, whereas the hand gesture for the number 3 can look very different depending on which three fingers a person is holding up. The advantage of low variability hand gestures are that they require less training data, and the classifier would likely have less false negative classifications of that particular gesture.

The gestures also needed to be distinct from each other in order to maximise the accuracy of the classifier. For example, since the "stop" and "OK" hand gestures look quite similar to each other, they would be a bad choice.

It was also necessary to verify that OpenPose could reliably detect hand keypoints in photos of the hand gestures used.

With these requirements in mind, the peace sign, thumbs up, and stop hand gestures were settled on.

3.2.2 Guidelines for dataset images

A variety of images were run through OpenPose to ensure that the software could accurately detect hand keypoints. It was found that OpenPose struggles with images that don't include a person's face or elbow, images where a person's hands are too small, and dark images.



Figure 1: Images which are too dark, do not include elbows, and are too small

3.2.3 Determining training dataset size

Training datasets should be large in order to limit the impact of variance [1]. Smaller training sets lead to overfitting, which reduces that accuracy of the classifier. A large and diverse dataset would produce a more robust classifier which works on different hand sizes and tolerates more variation in how hand gestures are made.

As the number of attributes in a class grows, the amount of items needed in a dataset to produce an accurate classifier can grow exponentially, this is known as the “curse of dimensionality” [2]. One recommendation that was found was that there should be approximately 5 items for each in the class [3]. The attributes in the system’s training set would be hand keypoints. Since the x and y coordinates of 21 hand keypoints are returned by OpenPose, a dataset of 210 items would be required based on the 5 items per attribute heuristic.

Existing hand gesture datasets were also looked at to estimate how many images would be required. The HGM-4 and Creativ Senz3D datasets have approximately 150 images per class. The tiny Hand Gesture Recognition dataset used augmented images to get 70,000 items per class. Due to the low number of classes and distinctiveness of the hand gestures chosen for this project, it was decided to be unlikely that such a large number of images would be required per class.

3.2.4 Sample bias

Another consideration that was researched was sample bias. Sample bias occurs when a particular class appears in the dataset very often, which can cause classification models to select that class erroneously. To avoid having a sample bias in the Jester dataset we aimed to collect a roughly even amount of images of the three gestures.

3.3 Dataset preprocessing

It was found that dataset cleaning - the process of removing bad values from the dataset, would be needed. It was also found that the items in the dataset would need to be standardised. In the case of the Jester keypoint dataset, this would mean ensuring that the size, location, and orientation of the hand keypoint coordinate distributions is roughly equal.

3.4 Classification algorithms

3.4.1 Selecting a classification algorithm

Convolutional Neural Network (CNN)

Initially a CNN was considered as they are a popular classification algorithm choice in other pose estimation projects. Upon research it was found that CNNs are primarily used for image processing problems, whereas the dataset for this project was in the form of graph coordinates.

Decision trees

The decision tree algorithm was ruled out fairly on. Decision trees work better with discrete data rather than continuous sets of coordinates, which would lead to an extremely complex and slow decision tree model.

Neural network

A multi-layer neural network with back propagation would be a suitable choice and would be considerably faster than the KNN algorithm that was settled on, but there was a few drawbacks to using a neural network:

1. Neural networks can require large datasets.
2. Neural networks require a large amount of hyperparameter tuning to get an accurate classifier, which takes time to implement.

K-nearest neighbours (KNN)

The advantages of implementing KNN in this project would be the wide variety of confidence and anomaly detection methods that can be achieved through modifying a KNN classifier. This was an important consideration for this project because most of the inputted images would be out-of-distribution once the web app is connected to the classifier.

The main drawback of KNN is performance, but there was reason to believe that this would not be an issue as the training set used was relatively small, with under 1800 items.

Due to time required to accurately tune a neural network and the wide range of potential KNN anomaly detection solutions, it was decided that a KNN classifier would be implemented.

3.4.2 Finding the optimum value for K in KNN

When designing the KNN implementation it was important to select an optimum value for K, the number of nearest neighbours examined by the algorithm. If the value is too low the classifier would be strongly influenced by noise and outliers in the data, but higher values are computationally expensive.

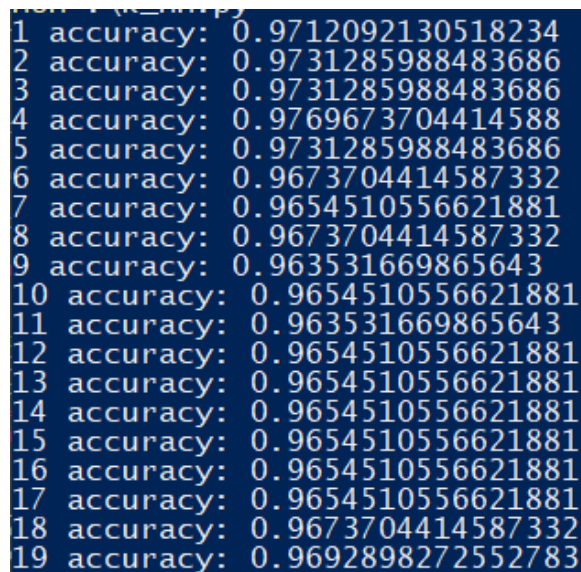
A script called `find_best_knn.py` was written to find the optimum value for K. The script randomly divides the keypoint dataset into an 80/20 split between the training set and a test set. The script then classifies the test set items using a 3rd party library KNN implementation 100 times, iterating through values for K between 1 and 100.

The `find_best_k()` method

```
def find_best_k():
    k = 1
    max_accuracy = 0
    best_k = 0
    while k < 100:
        classifier, accuracy = knn(k)
        if accuracy > max_accuracy:
            max_accuracy = accuracy
            best_k = k
        k += 1
    print(best_k)
```

It was found that the accuracy of the classifier was roughly 97% when K was between 1 and 5, and the accuracy of the classifier gradually dropped off as the values of K increased.

A potential reason why low values were more accurate is because the training and test data all came from the same dataset, which has less variation and noise than real world gesture images. With that in mind, 5 was selected as the value for K as it is on the higher side of the optimum 1-5 range that the script suggested.



```
1 accuracy: 0.9712092130518234
2 accuracy: 0.9731285988483686
3 accuracy: 0.9731285988483686
4 accuracy: 0.9769673704414588
5 accuracy: 0.9731285988483686
6 accuracy: 0.9673704414587332
7 accuracy: 0.9654510556621881
8 accuracy: 0.9673704414587332
9 accuracy: 0.963531669865643
10 accuracy: 0.9654510556621881
11 accuracy: 0.963531669865643
12 accuracy: 0.9654510556621881
13 accuracy: 0.9654510556621881
14 accuracy: 0.9654510556621881
15 accuracy: 0.9654510556621881
16 accuracy: 0.9654510556621881
17 accuracy: 0.9654510556621881
18 accuracy: 0.9673704414587332
19 accuracy: 0.9692898272552783
```

Figure 2: Output of `find_best_k()`

3.5 Classifying unrecognised hand gestures

As well as being able to classify hand gestures, the classification system was also required to be able to assert if an image does not contain a known hand gesture. This is known as out-of-distribution (OOD) detection or anomaly detection.

A range of potential anomaly detection algorithms were considered. Each algorithm was implemented to be tested and can be found in the `conf_functions.py` program.

KthNN and KNN anomaly detection algorithms

The KthNN algorithm looks at the distance of the kth nearest neighbour from the item being classified in order to determine confidence. When the distance value is higher, the confidence value is lower, and if the confidence value is below a threshold the item is considered to be an anomaly [5]. The KNN anomaly detection algorithm works in a similar way to KthNN except it averages the distance of the first K nearest neighbours.

These algorithms were ineffective at anomaly detection for this project, and misclassified a large amount of known hand gestures as anomalies, and out-of-distribution items as known hand gestures.

These algorithms failed to detect anomalies because of the variation of distances between items in the dataset. Peace sign keypoint coordinates are generally have less distance between each other than thumbs up and palm keypoint coordinates (ie. they have less variation in how people make the gesture). This means different max distance thresholds for each gesture are needed for accurate anomaly detection.

Local Density Outlier Factor (LdoF)

LdoF is a recently developed algorithm which compares the distance between the K nearest neighbours of an item to each other, and then to the item [6]. If the K nearest neighbours are all relatively close to each other, then the item needs to be close to the neighbours to not be considered an anomaly. If the K nearest neighbours are more scattered, then the item can have a greater distance from its neighbours and still be classified as being in-distribution.

Using a selection of 10 items, the LdoF algorithm was 90% accurate at classifying the items as in or out of distribution.

Local outlier Factor (LOF)

The local distance outlier factor is a more complex anomaly detection algorithm which compares the local density of an item (the average distance between the item's nearest neighbours), to the local density of each of the items' nearest neighbours. The LOF algorithm can be slow as it must calculate lists of nearest neighbours k^2+1 times, which leads to an enormous number of distance calculations.

The LOF implementation was able to achieve a 100% classification accuracy using a selection of 10 items. For this reason it was selected as the algorithm to be implemented in the final system.

4. Design

4.1 Context

4.1.1 Interfaces

Users interact with the Jester web app, which interfaces with the classifier (typically running on a different machine) via a network API. Users interact with the web app either through hand gestures (via their device's camera), or through their device's touch screen. The classifier and API are hidden from the user.

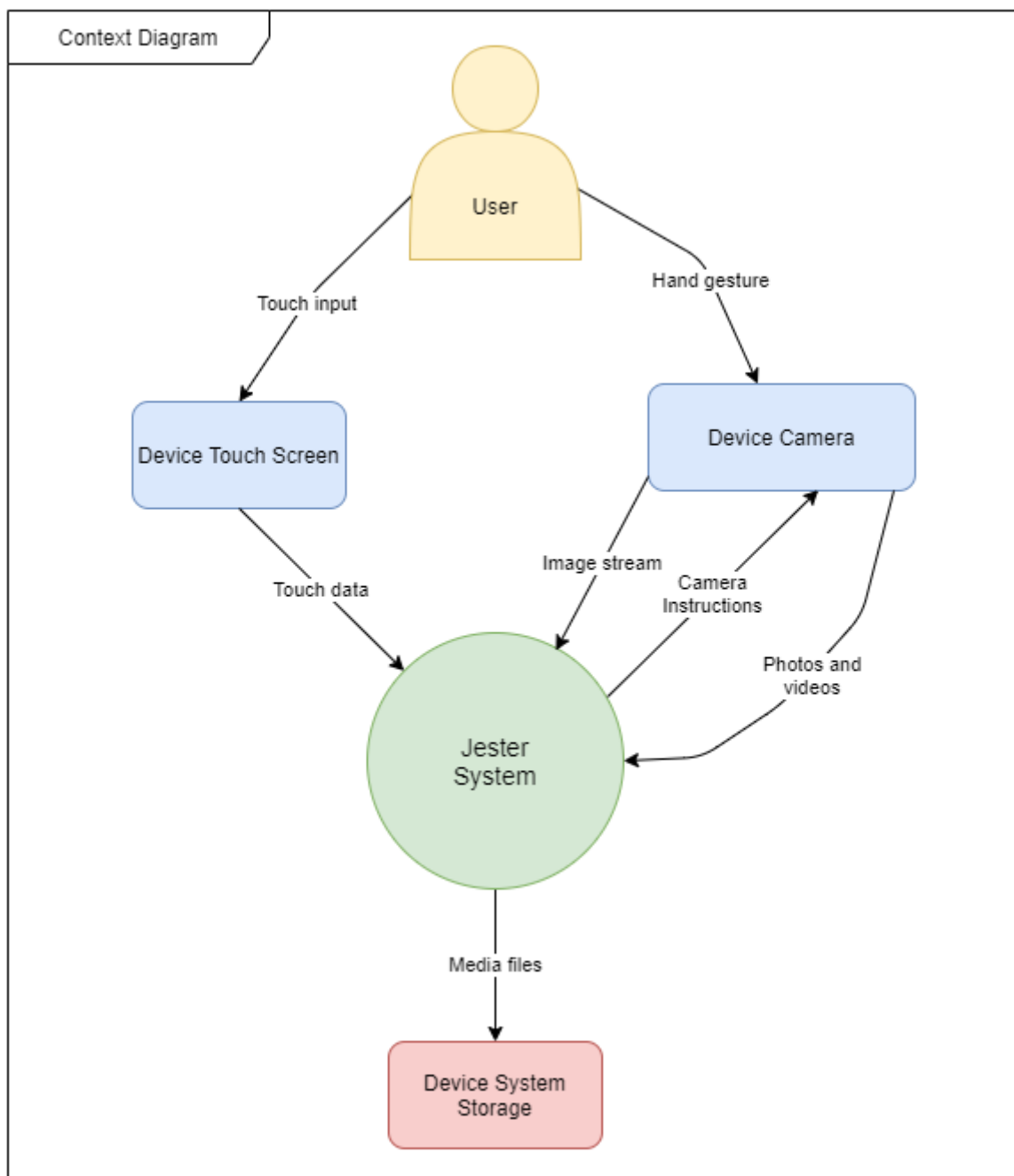


Figure 3: The inputs and outputs of the Jester system

4.2 Key design decisions

4.2.1 Distributed system

Jester is a distributed system, as it has components that run simultaneously on different machines.

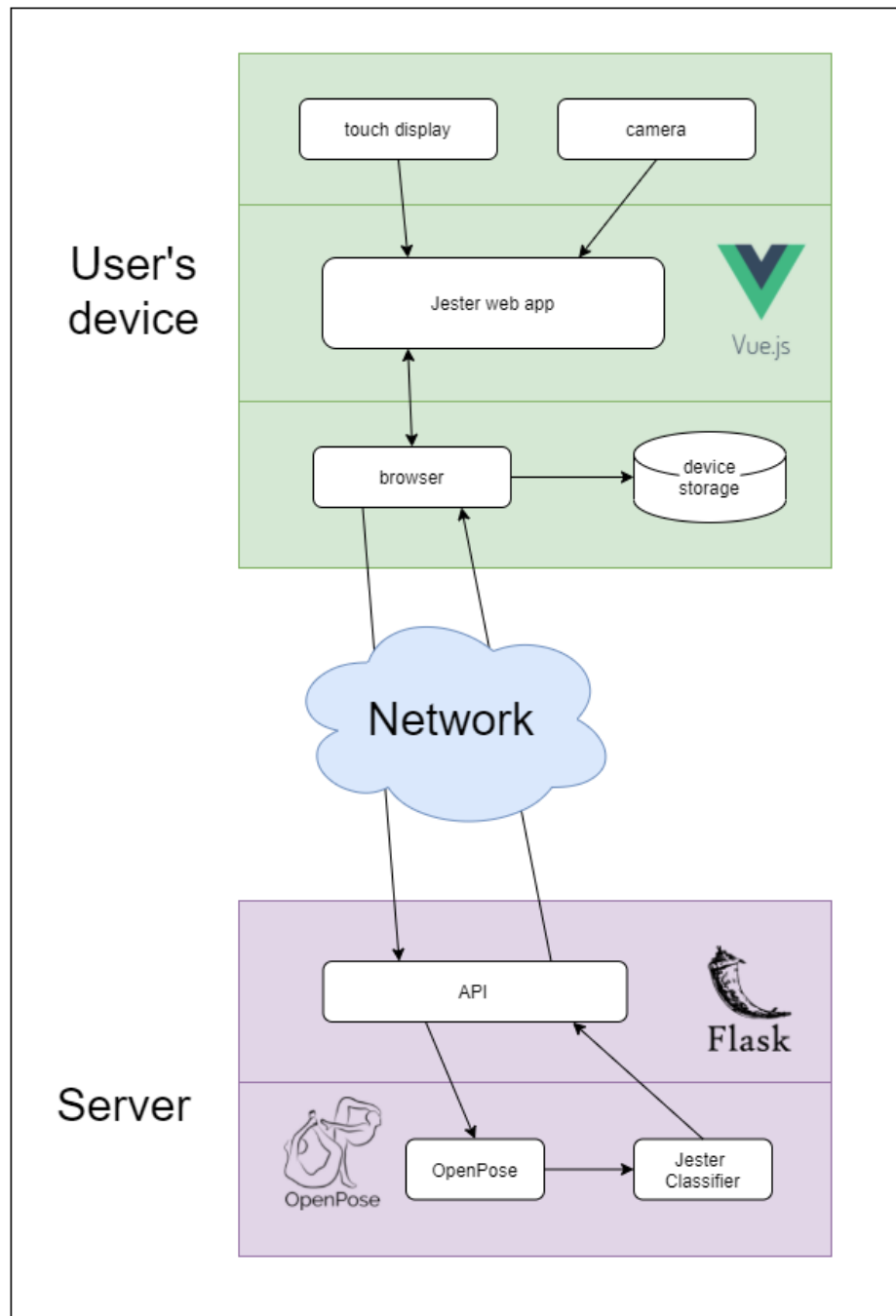


Figure 4: Jester system architecture diagram

The image processing and data mining processes of the system are more resource intensive than the web app component of the system. By implementing Jester as a distributed system, these processes can be run on a separate, more powerful machine to the front end, minimising the impact of this bottleneck.

By abstracting the classifier away from the user's devices, users are also no longer limited to Windows devices that support the Jester classifier. Abstracting the system's backend to a server also means that hardware upgrades are only required for a single machine, but affect all users.

4.2.2 Windows server

The primary reason for developing the classifier for Windows is because OpenPose has out of the box support for Windows 10. AWS and Google Cloud have offered instances of GPU-accelerated Windows servers at competitive prices, so the choice to support Windows servers over Linux should not be a considerable overhead cost.

4.2.3 Web technologies

Developing the front end as a web app has several advantages. Web apps are compatible with any device that has an internet browser. Running the app via a web browser also means no installation is required by users. Web apps can also be updated across all devices without requiring any input from users.

4.2.4 REST API

The web app and the server side classifier interface via a REST API. A REST API was chosen as JavaScript has native compatibility with HTTP requests.

Python Flask was used to implement the API as the developers had previous experience with the framework.

4.2.5 Vue.js

The front end was implemented as a single page application to achieve a better user experience. This is because switching between views does not require loading a new web page from a server as components can be dynamically rendered from the client.

Vue.js is a popular front end JavaScript framework used to create single page applications, and it was selected due to the framework's excellent online documentation resources. The framework also promotes good design practices by encapsulating front end logic into components.

4.3 User interface Design

Several user interface mockups were developed using Figma, and feedback was gotten from users to refine the design.

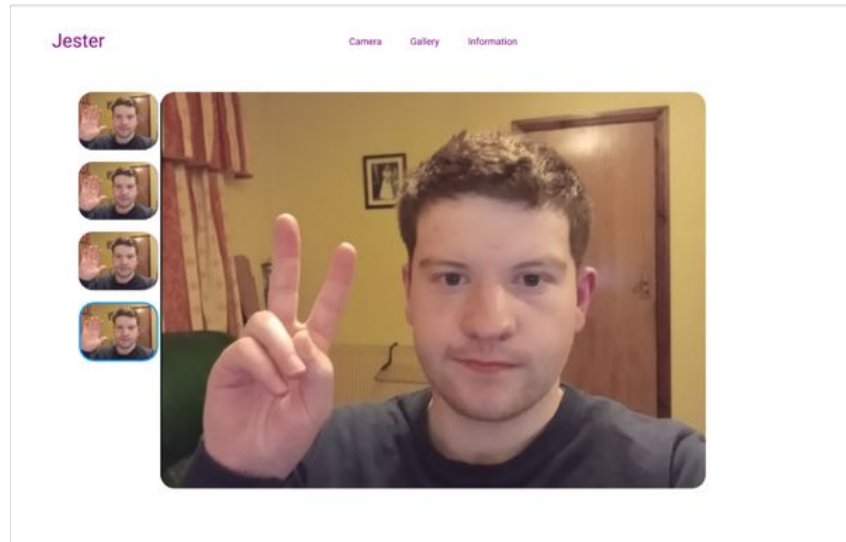


Figure 6: Mockup design for desktop devices

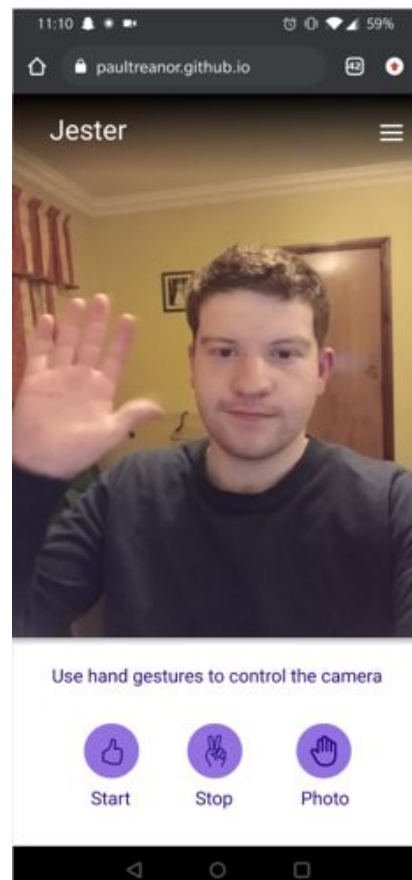


Figure 7: Mockup design for mobile devices

The final design which was settled on consisted of 3 views: a camera view, a gallery, and an info page.

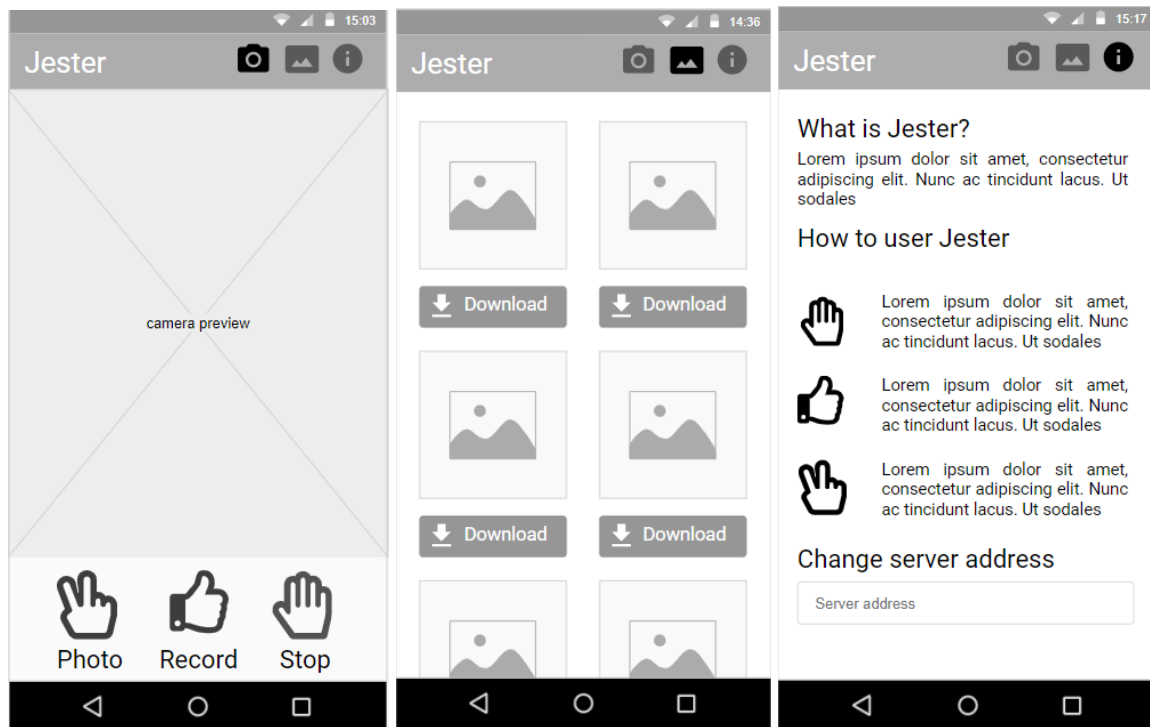


Figure 8: Wireframes of final user interface design

5. Implementation

5.1 Programming languages

Python

- Version 3.9
- Usage: Dataset collection & preprocessing, Jester classifier, REST API

Vue.js

- Version 2.6
- Usage: Jester web app

JavaScript

- Version: ES6
- Usage: Proof of concept web app

5.2 Dependencies

Dependencies are downloaded and installed from the installation script. For more information see the Jester user manual.

OpenPose

- Version: 1.6
- Usage: Jester classifier

Python Flask

- Version: 1.1.2
- Usage: REST API

Python Flask-Cors

- Version: 3.0.10
- Usage: REST API

Python sklearn

- Version: 0.24.1
- Usage: Jester classifier

npm Bootstrap-Vue

- Version: 2.21.2
- Usage: Jester web app

npm Axios

- Version: 0.21.1
- Usage: Jester web app

Python Pandas

- Version: 1.2.3
- Usage: Research

5.3 Dataset collection

Due to Covid-19 restrictions on movement, friends of the developers were relied on to help gather hand gesture images. Ethical approval was granted to collect these images, and a plain language statement was distributed to participants so they could provide informed consent to taking part in the project.

Participants involved in the collection of the hand gesture image dataset were informed of the requirements that the images would need to conform to. It was requested that images were taken in batches of each hand gesture so they could be more easily sorted into folders for each hand gesture.

In total 1800 images were collected, consisting of images from 13 people. According to the research in section X, this should be enough images to produce an accurate classifier model.

These images were sorted into folders according to the hand gesture, and a script was run to rename all the images according to the name of their parent folder. For example, images in the “peace” folder would be named “peace_1”, “peace_2”, “peace_3”...and so on. This would make adding a “class” field to the dataset csv much easier later.

A script was then written to place each image in a unique folder and run through OpenPose to collect the image’s keypoint data. Images were placed in unique folders because OpenPose automatically processes all images in a folder, but runs out of memory when processing more than one image.

Running this script placed a JSON file of body and hand estimation keypoint coordinates in an output folder.

The scripts used in the dataset collection phase can be found in the /src/dataset/dataset-collection directory of the project’s GitLab repository.

5.4 Dataset preprocessing

5.4.1 Dataset cleaning

The script `data_cleaning.py` was created to process the OpenPose JSON output into a more convenient CSV file.

- The JSON files were automatically read and iterated through.
- Right hand keypoints were selected and placed in a CSV file.

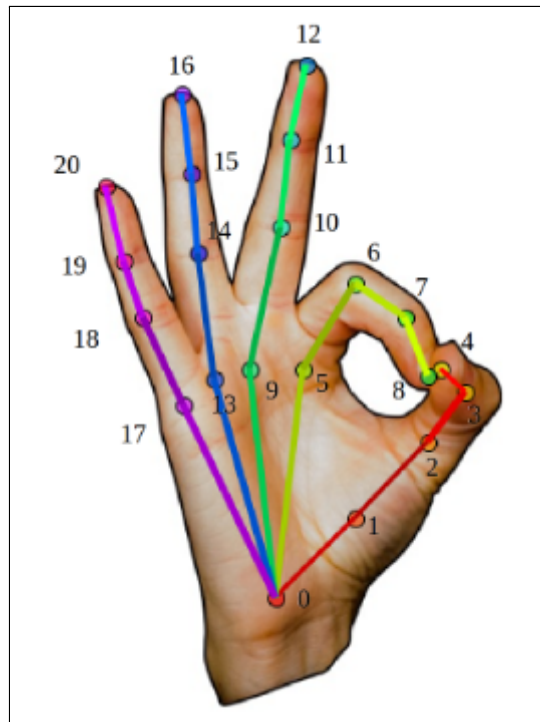


Figure 9: OpenPose hand keypoints (credit: OpenPose)

	A	B	C	D	E	F	G	H	I
1	image_name	point_0_x	point_0_y	point_1_x	point_1_y	point_2_x	point_2_y	point_3_x	point_3_y
2	palm_101_keypoints	0	0	-2.039	2.912	-4.369	6.213	-4.66	9.514
3	palm_102_keypoints	0	0	-2.79	2.827	-5.419	6.347	-5.372	10.191

Figure 10: Snippet of training set CSV file showing coordinate values

The CSV file fields for x and y coordinates for each of the 21 keypoints OpenPose detects in the hand. Each item also has a “class” attribute, which is either “peace”, “palm”, or “thumbs_up” and is assigned based on the file name of the item’s corresponding JSON file.

The script then removed inconsistent data from the CSV file. Items in the dataset with null values for keypoints were removed. These corresponded to blurry images.

5.4.2 Data transformation algorithms

Data transformations were carried to change the data into a homogeneous form that would allow for more accurate machine learning [4]. Transformation algorithms were developed to standardise the size, position, and orientation of the keypoint distribution for each item in the dataset. The code that implements these transformations can be found in `data_transformer.py`.

The `translate()` method moves the 0th keypoint of each item to the origin. The `enlarge()` method standardises the distance between 0th and 9th keypoint (the keypoints corresponding to the wrist and the base of the middle finger) of each item to 10 units. The `rotate()` method rotates all the keypoint coordinates around the origin so that the 0th and 9th point both lie on the y-axis.

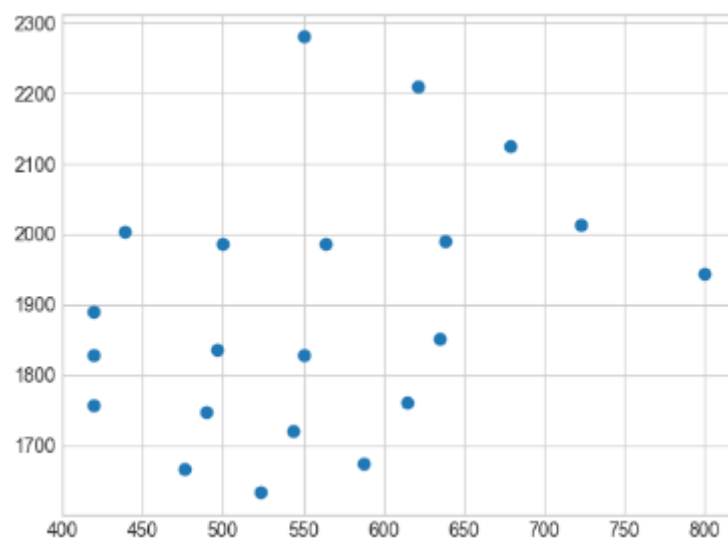


Figure 11: Plotted hand keypoints before transformations

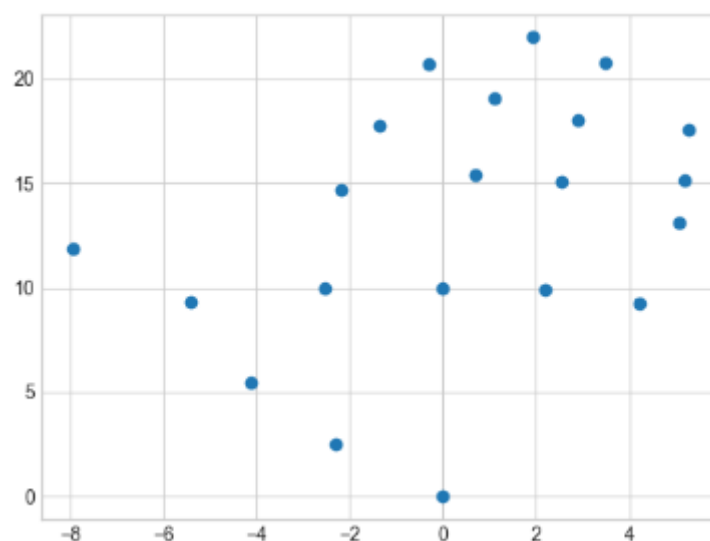


Figure 12: Plotted hand keypoints after transformations

The data transformation methods

```
def translate(hand_gesture):
    x_dif = hand_gesture[1]
    y_dif = hand_gesture[2]
    x = hand_gesture[1::2]
    y = hand_gesture[2::2]

    hand_gesture = [hand_gesture[0]]
    i = 0
    while i < len(x):
        new_x = round(x[i] - x_dif, 3)
        hand_gesture.append(new_x)

        new_y = round(y[i] - y_dif, 3)
        hand_gesture.append(new_y)
        i += 1
    return hand_gesture

def enlarge(hand_gesture):
    x = hand_gesture[1::2]
    y = hand_gesture[2::2]

    # Find the scale factor
    p_0 = (x[0], y[0])
    p_9 = (x[9], y[9])
    scale_factor = 10/dist(p_0, p_9)

    i = 1
    while i < len(hand_gesture):
        hand_gesture[i] = hand_gesture[i]*scale_factor
        hand_gesture[i] = round(hand_gesture[i], 3)
        i +=1
    return hand_gesture

def rotate(hand_gesture):
    x = hand_gesture[1::2]
    y = hand_gesture[2::2]
    hand_gesture = [hand_gesture[0]]

    # Find how much to rotate by
    point_9 = (x[9], y[9])
    degrees = findDegrees(point_9)

    # Rotate points by degrees and add them to list
    i = 0
    while i < len(x):
        tmp = x[i]
        x[i] = round((math.cos(degrees) * x[i] - math.sin(degrees) * y[i]),3)
        y[i] = round((math.sin(degrees) * tmp + math.cos(degrees) * y[i]),3)
        hand_gesture.extend([x[i], y[i]])
        i+=1
    return hand_gesture
```

5.5 Classification system

5.5.1 Classifier pipeline overview

The classification system uses a pipeline design pattern, which accepts an image as an input (from the network API), and outputs a classification.

1. OpenPose detects hand keypoints in image
2. Keypoint coordinates are converted from JSON into a Python list.
3. The coordinates are standardised using the same translate, rotate, and enlarge algorithms as described in section 5.4.2.
4. The coordinates list is classified by the KNN algorithm.
5. A confidence value is determined using Local outlier factor (which uses KNN).

If OpenPose does not detect a complete set of right hand keypoint in an image, an OOD classification is returned by the pipeline. If OpenPose's confidence values for keypoint coordinates is below a threshold, an OOD classification is returned.

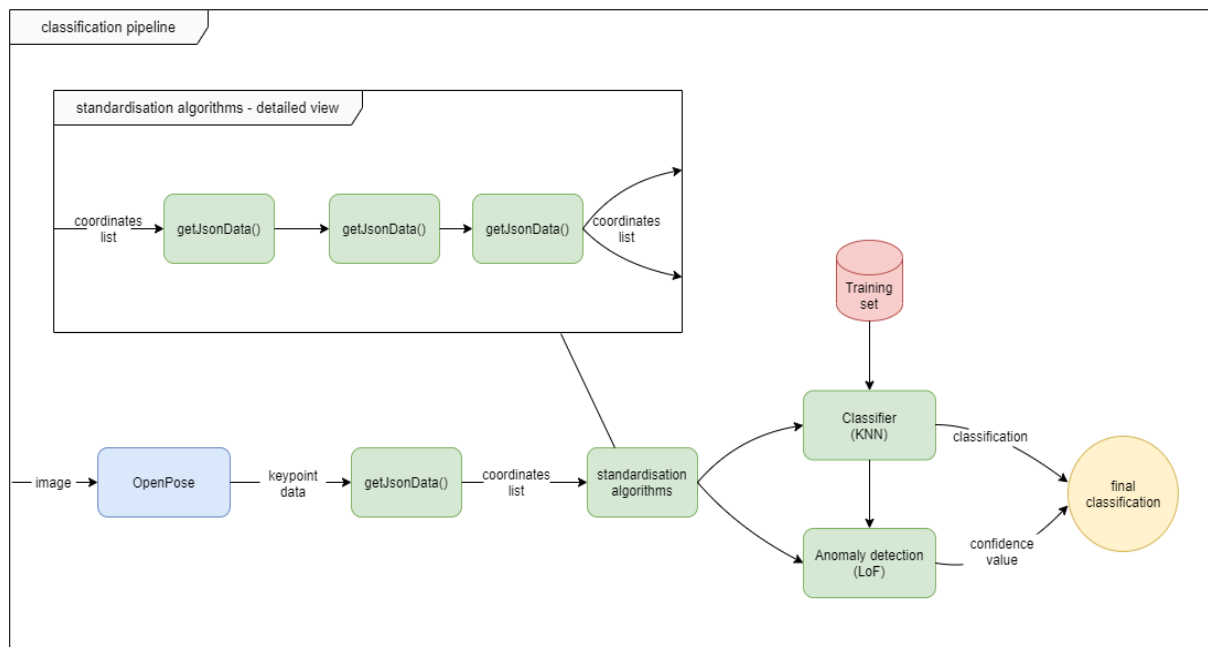


Figure 13: Diagram showing processes that make up the classifier pipeline

5.5.2 Classifier development

The classifier pipeline was developed iteratively. Initially the pipeline accepted a list of transformed coordinates and returned a classification using a 3rd party library implementation of KNN. Next the pipeline integrated OpenPose and the data transformation algorithms, which allows the pipeline to accept an image as input. Next the KNN algorithm was implemented from scratch. Finally the anomaly detection algorithm (local outlier factor) was implemented which allowed images to be classified as being out-of-distribution.

5.5.3 K-nearest neighbours implementation

The KNN algorithm determines the classification of an item by examining the classifications of the item's K nearest neighbours in the training set. The nearest neighbours are determined through a distance function, and the most common classification of the item's k-nearest neighbours is returned as the classification of the item.

KNN pseudocode

```
knn_classifier(inputItem) {
    nearest_neighbours = []           // list of tuples (item, distance)
    for (item in trainingSet) {
        distance = getDistance(inputItem, item)
        if (length(nearest_neighbours) < k) {
            nearest_neighbours.add(item)
        }
        if (distance < max [distance for neighbour in nearest_neighbours]){
            remove most distant neighbour in nearest_neighbours
            nearest_neighbours.add(item, distance)
        }
    }
    neighbour_classes = []
    for (item in neighbours) {
        neighbour_classes.add(item[class])
    }
    classification = most frequent class in neighbour_classes
    return classification
}
```

The Jester implementation of KNN can be found in knn.py. KNN is implemented as a class with the attributes k, and dataframe (the training set). The implementation uses the euclidean distance as the distance function.

5.5.4 Local outlier factor implementation

Local outlier factor is an anomaly detection algorithm which compares the local density of the item being classified with the local density of each of the item's k nearest neighbours. Density is the average distance between items in an area (in this case, the area is the radius around an item which includes the k nearest neighbours). If the local density of an item is considerably less than the local density of an item's neighbours, then the item is considered to be an outlier. If an item is an outlier then there is less confidence that it has the same classification as its k nearest neighbours, and so it is classified as being out-of-distribution.

The implementation of the LOF algorithm was the most complex component in the project. Other implementations of the LOF algorithm that were found online seemed needlessly complicated, so pseudo code was developed based on the mathematical notation representation from the Breunig et al. 2000 study which proposed the algorithm [7].

The implementation of local outlier factor can be found in conf_functions.py.

Local outlier factor is based on the concepts of reachability distance and local reachability-density.

Reachability distance

$$reach-dist_k(p, o) = \max \{k - distance(o), d(p, o)\}$$

The reachability distance between objects A and B is the maximum distance of B's K nearest neighbours or the distance between A and B (whichever is larger).

Local reachability-density

$$lrd_{MinPts}(p) = 1 / \left(\frac{\sum_{o \in N_{MinPts}(p)} reach-dist_{MinPts}(p, o)}{|N_{MinPts}(p)|} \right)$$

Local reachability-density is the inverse of the average reachability distance of A from each of its K nearest neighbours.

Local Outlier Factor algorithm

$$LOF_{MinPts}(p) = \frac{\sum_{o \in N_{MinPts}(p)} \frac{lrd_{MinPts}(o)}{lrd_{MinPts}(p)}}{|N_{MinPts}(p)|}$$

Local outlier factor of object A can then be defined as the average local reachability-density of each of A's neighbours divided by the local reachability-density of A.

LOF

```
reachabilityDistance(A, B) {
    B_nearest_neighbours = KNN(B)    // knn returns list of nearest neighbours
    BNN_distances = []
    for (neighbour in B_nearest_neighbours) {
        BNN_distances.add(distance(B, neighbour)    //euclidean distance
    }
    Kth_distance = max(BNN_distances)
    AB_distance = distance(A, B)
    reachability_distance = max(Kth_distance, AB_distance)
    return reachability_distance
}

localReachabilityDensity(A, nn) {
    r_distances = []
    for (B in NN) {
        B_r_distance = reachabilityDistance(A, B)
        r_distances.add(B_r_distance)
    }
    avg_r_distance = sum(r_distances/k)
    local_r_density = 1/avg_r_distance
    return local_r_density
}

localOutlierFactor(A, NN) {    //parameterised by list of nearest neighbors
    A_IRDensity = localReachabilityDensity(A, nn)
    IRRDensity_list = []
    for (B in NN) {
        B_nearest_neighbours = KNN(B)
        BLRDensity = localReachabilityDensity(B, nn)
        IRRDensity_list.add(BLRDensity)
    }
    IRRDensity_total = sum(Irrd_list)
    local_outlier_factor = IrrDensity_total/(A_IRDensity*K)
    return local_outlier_factor
}
```

5.5.5 Optimising KNN and LOF

Profiling the KNN and LOF implementations allows the bottlenecks to be identified. It was found that reading the CSV file that the training dataset was stored on was causing the KNN algorithm (which the LOF algorithm depends on) to be slow. By implementing KNN as a class with a function to return the K nearest neighbours, the CSV file can be read as a dataframe attribute when an instance of KNN is declared, and when the method to get the K nearest neighbours is called it reads from the dataframe object rather than reading in a CSV file. This change had a large positive impact on performance:

Algorithm (k=5 for all)	Before Optimisations (seconds)	After Optimisations (seconds)
KNN	0.078	0.071
LoF	2.790	1.956

5.6 Network API

5.6.1 API interfaces

The interfaces of the API were outlined based on the system's requirements:

Requirement 1:

The API must be able to accept an image and return a classification.

Get image classification	
Path	"/<imageName>"
Request type	POST
Input	Image File
Output	Classification, 201

Requirement 2:

The API connection should be easy to test.

API connection test	
Path	"/"
Request type	GET
Input	{}
Output	"Connection OK", 200

5.6.2 API development

The API program is contained in the file flask_server.py. Once the required interfaces were implemented, validation was added to ensure that POST requests contained a file, and that the file was of an acceptable file type (jpg or png).

5.7 System front end

5.7.1 Camera implementation

The front end was developed in an iterative. Initially a HTML and JavaScript proof of concept web page was created to ensure that capturing and previewing photos and videos was possible with web technologies. The final web app uses the same methods to take photos and videos using hand gestures.

This web page used the `getUserMedia()` JavaScript method to connect to the device's camera and display the camera's preview on the web page using a HTML video tag.

A HTTP post request called `postData()` which is parameterized by a URL and an image was created to post images to the API. At set intervals of 8 seconds the camera preview is captured as a HTML canvas item. This canvas data is then converted to a jpg and posted to the network API.

If the API returns a "peace" classification, the canvas snapshot is saved to the webapp.

If the API returns a "thumbs up" classification, a "mediaRecorder" item is created, which records the camera preview until the `stopRecording()` method is called, and sets the boolean variable `recording` to true.

If the API returns a "palm" classification and `recording` is true, then `stopRecording()` is called and `recording` is set to false.

5.7.2 Front end data flow diagram

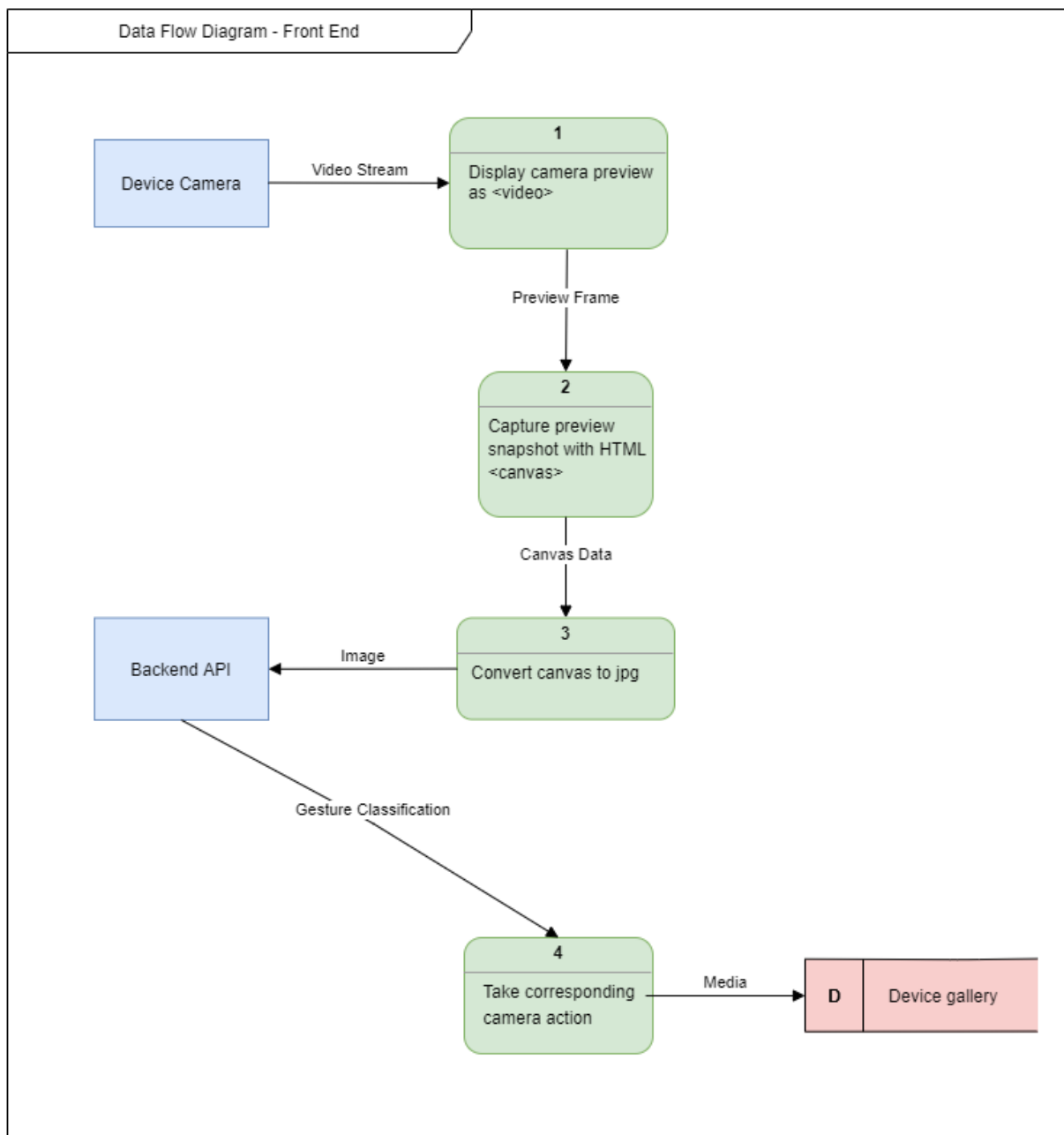


Figure 13: Front end data flow diagram

5.7.3 Vue.js components

Vue apps are made up of reusable components. Each component has its own HTML, JavaScript, and CSS, and Vue.js provides interfaces to allow the components to communicate with each other. The main components in the Jester app are the camera and gallery components, although there are subcomponents within these such as photo and video items in the gallery.

The camera.vue component contains the camera logic outlined in section 5.7.1. When a photo is taken or a video is finished recording, the photo/video data is emitted from the camera component and sent to the gallery component. The gallery has a list of photo and video objects, and renders each object in the list as a photo or video component containing the media and a download button.

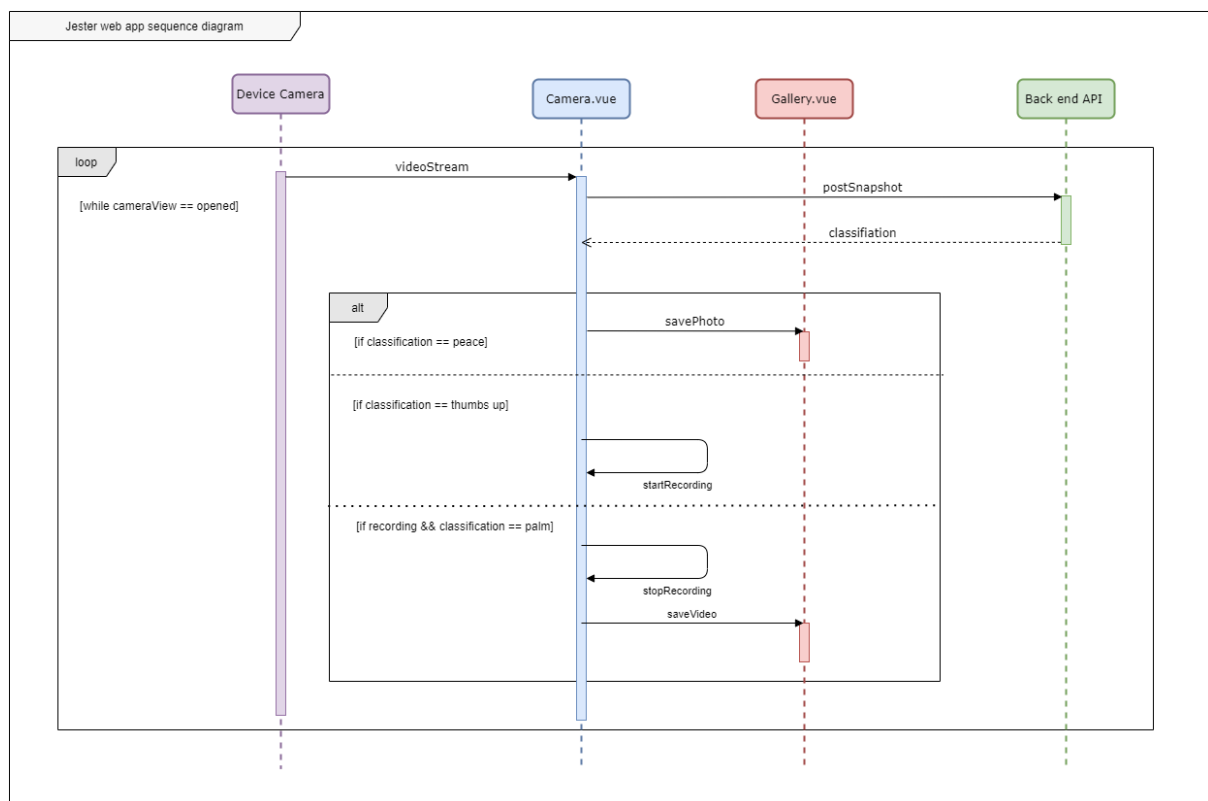


Figure 14: Front end sequence diagram

The info.vue component is a view which contains information to help users use the app. This component contains an input box where users can change the API URL depending on where it is hosted. This resets the post_url variable in camera.vue to the inputted address.

5.8 Installer

An installer was created in order to make setting up the Jester API on a server easier.

The installation script downloads and installs OpenPose into the correct path and installs all required python modules on the server machine.

The installation script is called `server_install.py` and can be found in the `/res` directory of the project.

5.9 Testing

For a full overview of the tests carried out please refer to the testing report document.

5.10 Continuous integration

Regressions tests were run automatically with each git commit in order to ensure that new features did not break existing functionality. This automated testing was carried out with a GitLab CI pipeline and Git Hooks.

GitLab CI Pipeline

A GitLab CI/CD pipeline was configured to run tests on the classifier code with each commit. An environment was set up which installed dependencies onto a GitLab container image, and then runs Python tests found in the repository. The environment configuration details can be found in the repository's `.gitlab-ci.yml` file.

Git Hooks

The GitLab CI pipeline could only run a limited number of tests due to the system's dependency on OpenPose which requires a GPU accelerated machine. Git Hooks were a more useful way of implementing continuous integration as they could run locally on machines which already have OpenPose running correctly. This allowed for more test coverage.

A Git Hook which runs automated python tests before each commit was created, alongside a script to warn the developers if they are committing to the master branch, and a script that ensures that commit messages are of an acceptable length.

The Git Hooks used in the development of Jester can be found in the `/res/git-hooks` directory of the repository.

6. Results

Jester demonstrates that an effective gesture recognition system can be implemented with k-nearest neighbours and local outlier factor. It appears to be the first gesture recognition system to implement these algorithms. This makes Jester a valuable learning resource if it is continued to be maintained as an open source software.

The Jester web app achieves the core requirements that were set out in the project's functional specification, that a mobile device's camera can be controlled with hand gestures.

While there are scalability issues that would need to be addressed before the app could be a viable consumer facing product, the system has the potential to solve real world problems. Swimmers can start and stop recording videos of their technique without needing to get out of a pool, family photos can be taken without the need for awkward timers, and people can record themselves with a tripod without needing to touch their device every time they want to take another take.

7. Known limitations

There are several known limitations in the system:

7.1 Left handed gesture detection

Currently Jester only supports gesture recognition when users use their right hand.

Left handed support could be implemented through the use of a “mirror” method, which negates the y coordinates of left hand keypoints to effectively turn them into right hand keypoints. This would have a negative impact on the system’s responsiveness.

7.2 Unreliable hand detection from OpenPose

OpenPose relies on detecting the position of a person’s elbow and head in order to detect hand keypoints. This results in Jester being unresponsive when a user’s body isn’t in full view.

7.3 Performance

The system's slow performance can cause a bad user experience, especially when the server is not equipped with a powerful GPU. The main performance bottleneck is OpenPose, which takes up to 5 seconds to process an image even on systems with a dedicated GPU.

8. Future work

Jester's KNN/local outlier factor classifier is a novel but very effective approach to gesture recognition, which potentially makes the project valuable as a learning resource. Because of this the authors are keen to maintain Jester as an open source software project.

There are several recommendations for future work:

1. Implementing the data transformation and classification algorithms in C

While Python allowed for the rapid development of the system due to the languages great support for manipulating data, it does leave room for some performance improvements. Implementing the system's classifier in a compiled language such as C could improve the systems performance significantly.

2. Experimenting with other keypoint detection solutions

OpenPose is a performance bottleneck in the classification system as it must carry out full body pose estimation in order to detect hand keypoints. Further research into alternative keypoint detection implementations is necessary for large performance improvements. Creating a custom lightweight hand keypoint estimator using OpenCV is one possible alternative.

3. Recording more metrics

More metrics are required to ensure that the future changes to the gesture classification system have a positive impact. A study on the classifier's accuracy and performance should be carried out with a large selection of images which did *not* contribute to the training set.

9. Appendix

References

1. Brain, D. Webb, G. (2000), On the Effect of Data Set Size on Bias and Variance in Classification Learning. Proceedings of the Fourth Australian Knowledge Acquisition Workshop. Available at:
https://www.researchgate.net/publication/2456576_On_the_Effect_of_Data_Set_Size_on_Bias_and_Variance_in_Classification_Learning
2. Raudys, S. Jain, A. (1991). Small Sample Size Effects in Statistical Pattern Recognition: Recommendations for Practitioners. Transactions on pattern analysis and machine intelligence, volume 13, page 262. Available at:
<https://sci2s.ugr.es/keel/pdf/specific/articulo/raudys91.pdf>
3. Vishwesh, K. (2020). Curse of Dimensionality: An intuitive and practical explanation with examples. Medium. Available at
<https://medium.com/flutter-community/curse-of-dimensionality-an-intuitive-and-practical-explanation-with-examples-399af3e38e70> [Accessed 22 Jan. 2021]
4. Han, J. Kamber, M. Pei, J. (2012). Data Mining Concepts and Techniques. Waltham, USA: Morgan Kaufmann, pages 85 & 111
5. Gu, Xiaoyi & Akoglu, Leman & Rinaldo, Alessandro. (2019). Statistical Analysis of Nearest Neighbor Methods for Anomaly Detection. Available at:
https://www.researchgate.net/publication/334361033_Statistical_Analysis_of_Nearest_Neighbor_Methods_for_Anomaly_Detection
6. Zhang K., Hutter M., Jin H. (2009). A New Local Distance-Based Outlier Detection Approach for Scattered Real-World Data. In: Theeramunkong T., Kijssirikul B., Cercone N., Ho TB. (eds) Advances in Knowledge Discovery and Data Mining. PAKDD 2009. Lecture Notes in Computer Science, vol 5476. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-01307-2_84
7. Breunig, Markus & Kriegel, Hans-Peter & Ng, Raymond & Sander, Joerg. (2000). LOF: Identifying Density-Based Local Outliers.. ACM Sigmod Record. 29. 93-104. <https://doi.org/10.1145/342009.335388>.