Human-In-The-Loop Person Re-Id for Sensitive Datasets

by

Vlad G. Stelea

A Thesis

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

In partial fulfillment of the requirements for the

Degree of Master of Science

in

Computer Science

by

May 2022

APPROVED:

Professor Jake Whitehill, Thesis Advisor

Professor Lane Harrison, First Reader

Professor Craig Wills, Head of Department

Abstract

Tracking who appears within videos and when is an important pre-processing step for understanding how students respond to lessons and analyzing their performance within the classroom. However, existing models are often trained on datasets that are not representative of the videos they are applied to leading to them under-performing. For applications where final label accuracy is critical (such as classrooms), we explore how to combine facial recognition predictions with human annotations to reduce annotation actions by a factor of 10. In this thesis we (1) develop an annotation tool and workflow for labeling facial recognition datasets and use it to (2) annotate a custom classroom dataset. We also (3) develop an extensible simulation where we use the annotated dataset to explore the effect of using a (4) pre-trained FaceNet model to generate annotations and (5) retraining this model on newly labeled data. Next we (6) evaluate how the number of exemplars provided for each video affects the overall dataset annotation accuracy. Finally we (7) introduce a new method of quantifying the uncertainty of a model's predictions and explore whether labeling only uncertain examples can be used to reduce annotation actions.

Acknowledgements

I am extremely grateful to Professor Jacob Whitehill for his guidance and valuable insights making this work possible. His office was always open for me when I ran into challenges.

I would also like to thank Professor Lane Harrision for his valuable comments on my thesis. Thank you to my colleagues at the Whitehill research group for their feedback and support throughout this thesis.

A big thank you to my family and friends who throughout my years of study have provided me with their support, patience and advice.

This research was performed using computational resources supported by the Academic & Research Computing group at Worcester Polytechnic Institute.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Human activity recognition is a field that studies the automatic classification of human action from video [12]. Recently, human activity recognition has been used to estimate the positive climate of classrooms [10]. Positive climate is a useful measure that informs teachers about how students in their classroom are feeling. It helps teachers understand their students and adjust to their needs. Because each student reaction contains information about their feelings, it is important to identify and track students within each frame throughout the video, aggregating their reactions before classifying how they feel. Since this task is done within the educational sector, it is critical that the identities be accurate as the decisions made from this data will affect students. While this step can be done entirely by a human annotator, this can be time consuming and expensive to do [18].

One way to track students through the course of a video is through the use of facial recognition technology. Because of the high performance of existing face detection methods within other domains, we will only identify students by their faces. This allows us to use off-the-shelf models such as FaceNet [16] to reduce the computational complexity associated with training new models.

However, research shows that off-the-shelf FaceNet performance drops when applied to classroom environments [10]. This can be attributed to the data that these models are trained on. In these datasets, children are not as common as they are in the classroom. Because of the sensitivity of classrooms, this is not acceptable.

In order to overcome the limitations of off-the shelf facenet models, we propose having a human correct the predictions of a facenet model. In this thesis we explore various strategies to reduce the load associated with correcting these predictions.

1.1 Background

Often when a model is trained on one dataset (a source dataset) and evaluated on another dataset (a target dataset), the model's performance will drop on the target dataset. This phenomenon is caused by domain shift where the distribution of classes or features in a training dataset is not the same as that of the target dataset. One solution to domain shift is to have annotators incrementally correct predictions of a pre-trained model and use these new labels to improve the model. This strategy will be explored in section 1.1.1. Another way to deal with domain shift is to adapt the model on the target dataset in an unsupervised way. This is called Unsupervised Domain Adaptation and will be explored in section 1.1.2. While this strategy uses less human resources, it often performs worse than Active Learning. As such it can be used as a pre-step to doing active learning. Facial recognition can be done in many different ways. An overview of its strategies and their trade offs will be discussed in section 1.1.3.

1.1.1 Active Learning

Active learning is a field in machine learning where the learning algorithm will query a knowledge base to label new data points. When this knowledge base is a person, this type of active learning can be classified as having a human-in-the-loop (HIL).

One common use case for active learning is the reduction of labeling overhead for images. For example, one solution looked at the use of natural language to adjust semantic segmentation predictions for labeling. They achieved this through the use of a "guiding block" that adjusts neural network activations based on the text encoding of the hint [14].

Recent work has also looked at how to reduce annotator cognitive load in person re-id labeling by having annotators select only obviously negative examples from the ranking list. This strategy tagged "Human Verification Incremental Learning" is an alternative to hard-negative mining that isolates identities classified incorrectly with a high confidence for retraining [18]. While this has been shown to reduce the labeling efforts associated with annotating re-id datasets, it is important to note that because of our need for highly accurately labeled datasets, this strategy on its own would not be enough to generate the final labels we need.

1.1.2 Domain Adaptation

Domain shift occurs when a model that is trained on a source dataset is deployed for use on a target dataset. It is caused by the difference in domain between the source and target datasets which in the case of images can be illumination, backgrounds and in our case age distribution of subjects. The practice of reducing the effect of domain shift is called *domain adaptation* and can be done in a supervised and unsupervised manner. The goal of domain adaptation is to take the knowledge learned from the

source dataset and apply it to the target dataset to align the source and target feature distributions.

Unsupervised domain adaptation refers to a family of strategies that attempt to do domain adaptation without human input. There are two general strategies for unsupervised domain adaptation: *Self-training* or *Adversarial Training*. Many of these strategies have been applied to the semantic segmentation problem.

With the *self-training* method of domain adaptation, the model trained on the source dataset is used to generate labels for the target dataset. The assumption of self-training is that an image with a high predicted score will likely have an accurate pseudo-label. Afterwards the model is trained using standard supervised training techniques and loss functions [13].

Equation 1.1 shows the loss function which can be used for self-training UDA. It is essentially the sum of cross entropies of the predictions on the source domain and the predicted labels on the generated labels of the target domain. In this equation, $\hat{y}_t^{(c)}$ indicates a psuedo label of class c in the target domain.

$$\min_w \mathcal{L}_{CE} = -\frac{1}{|X_S|} \sum_{x_s \in X_S} \sum_{c=1}^{C} y_s^{(c)} \log p(c|x_s, w) - \frac{1}{|X_T|} \sum_{x_t \in X_T} \sum_{c=1}^{C} \hat{y}_t^{(c)} \log p(c|x_t, w) \tag{1.1}$$

Often with self-training processes, only highly confident pseudo-labels will be used at earlier stages and as the classifier becomes more confident on the target domain more target examples will be used for training. For example recent work has used a threshold of confidence in order to select which images to use for their UDA process [8]. This work also used triplet loss in order to do domain adaptation instead of the standard cross-entropy loss.

Another method to do UDA is through the use of adversarial training. This

method uses a segmentation model to produce a segmentation mask and a discriminator model to determine whether the segmentation mask came from the target or source domain. The goal of the segmentation model is to confuse the discriminator model about which domain the segmentation mask came from. Both the discriminator and segmentation models are optimized alternately [13].

Equation 1.2 shows the loss function which can be used for adversarial unsupervised domain adaptation. In this equation X_s is the raw images from the source dataset and X_T is the set of raw images from the target dataset. The first term is the cross entropy of predictions on the source dataset while the second term is the Mean Squared Error of the adversarial process. With this equation $\mathbf{M}$ is the segmentation model that attempts to fool the discriminator $\mathbf{D}$.

$$\min_{\mathbf{w}} \max_{\mathbf{D}} \mathcal{L}X_{AT} = -\frac{1}{|X_S|} \sum_{x_S \in X_s} \sum_{c=1}^{C} y_s^{(c)} \log p(c|x_s, w) + \frac{\lambda_{adv}}{|X_T|} \sum_{x_t \in X_T} [\mathbf{D}(\mathbf{M}(x_t, w)) - 1]^2 \tag{1.2}$$

The form of UDA seen in 1.2 does not consider the task-specific decision boundaries. This is a problem because adapted target dataset points can be shifted to the wrong side of the decision boundary reducing model performance.

In order to solve this problem, another form of UDA adds two network heads to the pre-trained model and adversarialy trains using the discrepancy between the two heads predictions [15]. An overview of the process can be seen in figure 1.1. This process has three steps that get repeated over the course of training.

1. Train both classifier and generator to classify the source samples correctly

2. Fix the generator and train the classifiers in order to maximize discrepancy. This step is important because without it the discrepancy would converge making it hard to detect source and target domain.

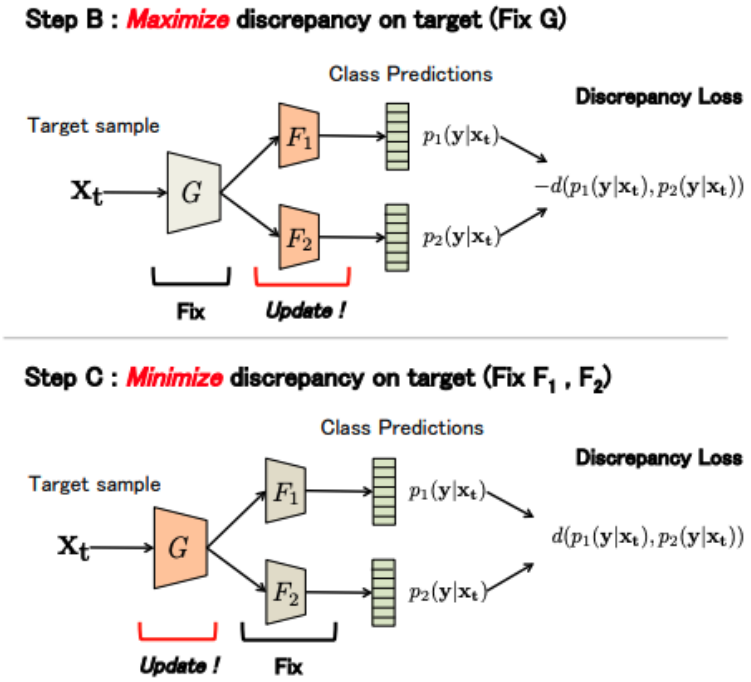

Figure 1.1: Adversarial training steps of MCDDA method [15]

3. Fix the classifiers and minimize the discrepancy between the classifiers. The generator is not fixed anymore so back-prop changes the generator's weights.

Steps 2 and 3 in this process are done using discrepancy loss. This loss can be seen in equation 1.3. This loss function works for K classes where $p1_k$ is the output from network head 1 and $p2_k$ is the output from network head 2.

$$d(p1, p2) = \frac{1}{K} \sum_{k=1}^{K} |p1_k - p2_k| \tag{1.3}$$

Recent work has also looked at taking an active learning approach to Domain Adaptation where annotators are only asked to label datapoints where predicted labels are uncertain. For example, one paper applied this strategy to semantic segmentation labeling tasks by only querying user annotator input for pixels that a separate uncertainty model flagged [17].

In the above mentioned paper, the pixel selector model was created by taking an existing feature extractor and classifier model and copying them over to the pixel selector model (with the classifier being copied twice). Afterwards, the classifier models are trained in an adversarial fashion in order to maximize their prediction discrepancy for pixel level class labels. Finally, a pixel mask is created where pixels that the two classifiers predicted different labels for are selected for relabeling [17].

1.1.3 Face Re-ID models

Face re-identification is the problem of matching face images with their corresponding identities. It often also goes by the name of facial recognition. Face Re-id can be done through the use of standard classification models or through the use of metric encoding models.

The simplest type of face re-id model treats each face's identity as a distinct class where the classes are pre-determined at training time [19]. Often this can be done using CNN's with a softmax layer at the end. Classification models trained in this way have the downside that once a model is trained, it is not able to adapt to new identities without retraining.

Metric encoding models avoid this constraint by encoding face images into latent space where faces from the same identity would have a small L_2 distance from each other. This output can then be fed into a clustering algorithm such as K-Nearest Neighbors algorithm (KNN) which has access to existing encodings and their identities to determine who the identity of the query image [16].

One strong metric encoding model for faces is the FaceNet model. FaceNet models are a class of models which are able to encode faces into a 128 dimension latent space by minimizing a triplet loss function during training. This loss function can be seen in equation 1.4. Triplet loss works by selecting three images from the

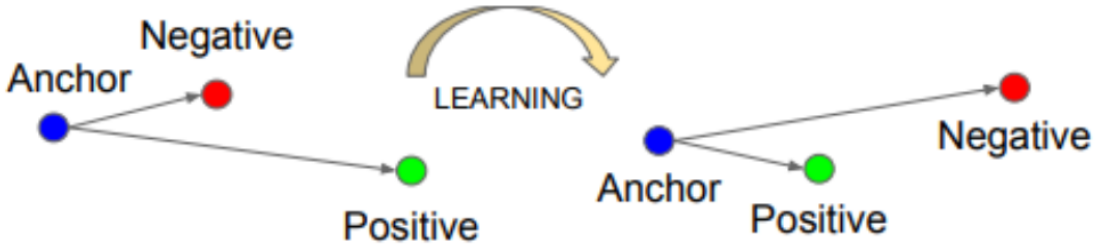

Figure 1.2: Triplet loss moves negative encodings farther apart and positive encodings closer together [16]

training set (an anchor, negative and positive). The anchor and positive are selected to be of the same identity while the negative has a different identity. By minimizing this loss function, the positive encoding is encouraged to move closer to the anchor while the negative encoding is encouraged to move away from the anchor [16].

$$L = \sum_{i}^{N} [||f(x_i^a) - f(x_i^p)||_2^2 - ||f(x_i^a) - f(x_i^n)||_2^2 + \alpha] \tag{1.4}$$

In order for triplet loss to perform well, it is important to select triplets where the positives are far from the anchor and negatives are close to the anchor. However, selecting the best triplets from the entire dataset is not feasible. To fix this issue, different triplet mining strategies have been developed to find the best triplets withing mini-batches. For example, hard-triplet mining is a strategy that works by defining each exemplar within a mini-batch as an anchor, then finding the hardest negative point for that anchor [16].

1.2 Research Questions

We focused on the following research questions to direct our research.

1. How well do existing face re-id models transfer to classroom data sets?

2. How quickly can models trained with a human in the loop improve compared to models trained with data collected in an offline manner?

3. How much "wall-time" can we save by having a human correct model predictions as opposed to annotating the entire dataset manually?

4. What is a useful human in the loop methodology to make face re-id labeling of videos more efficient in new domains

5. How does labeling only uncertain encodings affect performance on the remaining examples

Chapter 2

FaceSim Architecture

Understanding how different factors affect labeling performance is critical before selecting what strategies to include in an annotation tool. One way to do this is to build an annotation tool and evaluate each strategy through user studies. However, this process would be too time consuming as we would need to conduct these user studies with actual people.

In order to avoid these delays when evaluating different strategies and parameters, we wrote a simulator where we were able to swap out different strategies and understand how they perform relative to each other. For example, certain model adaptation policies would perform better than others, allowing us to converge on optimal performance more quickly. Furthermore, these simulations allowed us to estimate important attributes about this application domain such as how quickly our auto-labeler model adapted to classroom datasets, it's best-case performance, and how many relabeling actions annotating agents needed to perform upon convergence.

2.1 Design Goals

When designing the architecture for FaceSim, we had two major design goals: make the code easily extensible (so that we wouldn't spend a lot of time figuring out where certain logic belongs) and make sure that code could be hot-swappable in order to maximize code reuse.

In order to achieve these goals, we designed a layered architecture where layers were only able to directly interact with the layers directly below them. We also utilized a dependency injection strategy to construct layers within each experiment allowing us to choose different strategies within each experiment without rewriting large amounts of code. A high level diagram of the layered architecture can be seen in figure 2.1.

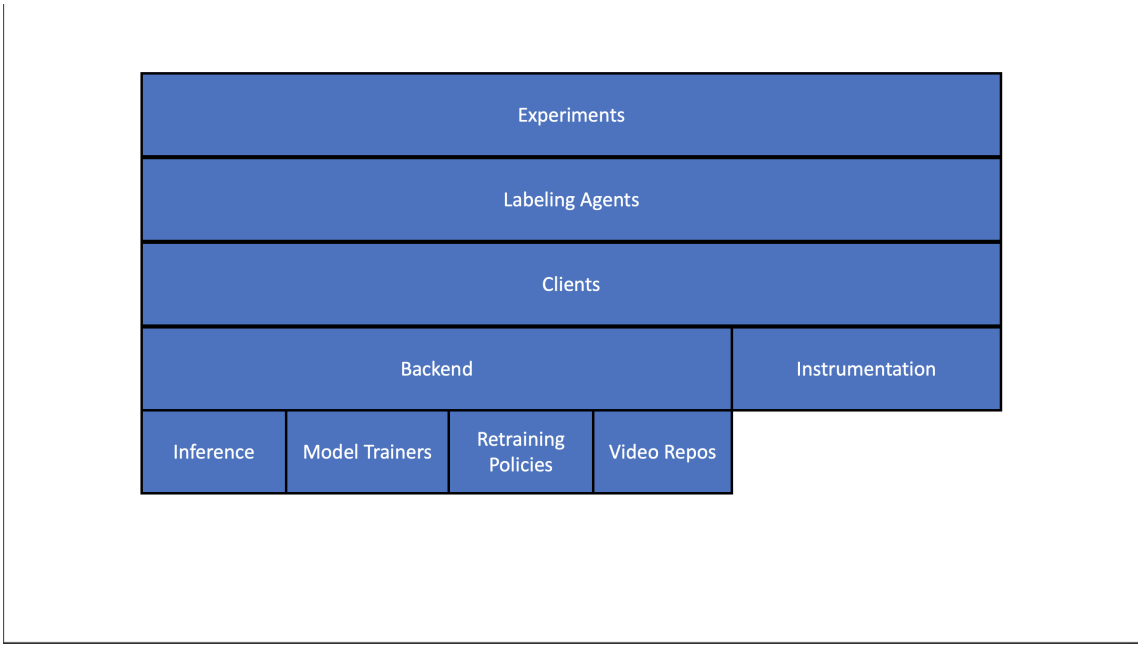

Figure 2.1: FaceSim Layers

2.2 Layers

2.2.1 Experiments

The experiments layer is the entry point of every experiment we write. In order to re-use code, we instantiate the classes of each layer that we will use and combine them here. This is where we do all the dependency injection. This is the only layer in the simulation that has access to all the layers because this allows us to easily select the parameters of the simulation. More information about individual experiments can be found in chapter 4. Each experiment also supports different command line parameters in order to make them more reusable within jobs.

2.2.2 Labeling Agents

Labeling agents are analogous to annotators in an actual experiment. We defined two labeling agents for our experiments: *editing agent* and *relabeling agent*.

The relabeling agent is the simplest type of agent possible and simply relabels each annotation with the ground truth no matter if the machine generated annotation is correct or not. Because the relabeling agent does not look at the machine generated annotation to decide whether a relabel is needed, this agent will perform more actions.

In order to understand how many actions each agent actually needs to take based on machine generated annotations, we also created an editing agent. Like the relabeling agent, this agent is an oracle (knows the ground truth and annotates perfectly), however it will only edit machine generated labels if the annotation is not correct. This agent performs fewer actions than the relabeling agent.

2.2.3 Clients

The client layer is analogous to the web-page that an annotator would interact with in order to correct machine generated annotations. This layer is responsible for talking to the backend layer and presenting the pre-annotated videos to the labeling agents in a format readable by the agents. It is also responsible for talking to the instrumentation layers which keep track of user actions.

2.2.4 Backend

The backend layer acts as the glue for the different machine generated annotation functions. It uses the provided video-repo to load videos and pass them to the inference layers for auto annotation. It will also check with the retraining policies in order to figure out when it is necessary to retrain a model. If necessary, it will tell the model trainer to retrain the inference model. It also passes all the annotated videos to the client.

2.2.5 Video Repos

The video repos layer is responsible for maintaining labeled datasets and loading unlabeled videos for the backend using the specified *VideoLoader* implementation. It is also responsible for creating PyTorch Datasets for use in the retraining step. We have two implementations of Video Repos in the FaceSim project: *DiskVideoRepo* and *VideoRepoImpl*.

The *VideoRepoImpl* was our first implementation of the video repo. It worked by loading videos from the specified location and returning them to the backend when asked. When the backend would update that video's annotations, this implementation would store the loaded video and annotations in memory. While this allowed for

quick retrieval, upon experimentation with our full dataset on the Turing cluster, we realized that this strategy takes up too much memory and was causing our jobs to crash.

In order to fix this issue, we implemented the *DiskVideoRepo*. This implementation worked very similarly to the *VideoRepoImpl* described above, however when annotations were updated, these annotations would get stored to a specified directory in the format shown below. This allowed us to chunk image loading and run jobs without them running out of memory.

2.2.6 Inference

The inference layer is responsible for annotating video frames and passing annotations back to the backend. This layer has no access to ground truth labels, however it is responsible for maintaining the identities associated with different encodings. In order to use the inference layer, the backend maintains a reference to a *ClassifierPipeline* class which is constructed with an encoding model and clustering classifier as parameters.

2.2.7 Model Trainers

Model trainers are responsible for retraining our inference layers with already annotated data. This layer has access to *LabeledVideoDatasets* which contain the labels given by the annotating agents. While it would have been possible to just provide the ground truth values directly to the model trainers, we chose to avoid this in case we wanted to implement some non-perfect annotators to better simulate human annotators.

The *ModelTrainer* we use the most is the *TripletLossModelTrainer*. This model trainer is an implementation of the Triplet Loss training algorithm specified in [16].

It supports Batch Hard Triplet selection. We also have an *UncertaintyModelTrainer* which trains a model based on the process detailed in section 4.7.2.

2.2.8 Retraining Policies

Retraining policies are responsible for determining when a model needs to be retrained. They do this by maintaining different state variables such as the number of videos that are labeled and then returning whether it is time to retrain a model from the *should_retrain* method. It is up to the backend to call the methods associated with each state update when the state would have changed, and check whether it is time to retrain right after.

2.2.9 Instrumentation

Our instrumentation is responsible for logging all user actions. It does this by logging every time an action is taken on the client such as adding a label or deleting a label. These logs are then stored in a pandas [1] *DataFrame* which is exported to a CSV for further analysis. We chose to store all "Agent Actions" because this allows us to aggregate our information in different ways after the simulation has already run. This strategy is analogous to a data lake and makes it possible for us to not have to rerun an entire experiment in the case that we realize that we wanted more data.

Chapter 3

Datasets

In this chapter we will discuss the various datasets we used for our experiments. Why we need to collect and annotated our own dataset as well as how we did it can be found in section 3.1. The off-the shelf model we used was trained on the VggFace2 dataset. A discussion on this dataset can be seen in section 3.2. Because this dataset is no longer available, for our uncertainty model experiments we utilized the labeled faces in the wild dataset. A discussion on this dataset can be found in section 3.3.

3.1 Facesim Dataset

Because to our knowledge, there are not any pre-labeled face datasets of classroom environments, we needed to manually label a dataset for the FaceSim simulation to get ground truth values. However, because of the limitations to manual labeling previously mentioned, we developed a partially automated strategy to collect and label classroom videos.

In order to reduce the effort associated with labeling the collected videos, we developed a two-step process. The first step of the process was to run an entire video

through an auto-labeler which grouped detected faces together. This auto-labeler used a PyTorch [2] implementation of FaceNet [9] and clustered faces with similar encodings together. We then took the auto-detected labels and ran them through an annotation tool we wrote which uses a Split-Merge strategy to reduce the number of actions need to take.

3.1.1 Collecting The Raw Videos

All the videos used for the simulation were collected from a YouTube playlist of classrooms using the PyTube library [3]. This playlist can be found here **https://www.youtube.com/playlist?list=PLG5a7din-gkIBqxwqfaq1r4mMet2Hx4Tz**. After collecting the raw videos, we manually labeled them using the strategy described in the following section.

3.1.2 Auto-labeler

The auto-labeling pipeline is comprised of 5 steps which can be seen in figure 3.1. The first step is the Video Splitting step which splits a video into 60 second sub-video segments. By splitting a video into sub-segments, we reduce annotation load into smaller tasks.

Afterwards, we do a Frame Rate Reduction step which reduces the frame-rate for each video to 2fps. This step achieves two functions. It reduces the amount of faces we need to check after auto-labeling as well as reducing the computational load of auto-labeling. Because of the reduced labeling load, we were able to annotate multiple videos in a much shorter time which is important as each video has faces in different contexts.

After reducing the frame-rate, we used the MTCNN model packaged with [9] to detect all the faces in each frame. This is an important pre-processing step because

the face-net encoder model we use expects a 160x160 image of each individual face. The encoder model takes the detected face images and encodes them into a 512 dimensional vector where faces that belong to the same identity have a small euclidean distance from each other.

In order to cluster faces with the same identity together, we use the agglomerative clustering implementation associated with scikit-learn [4]. While sklearn does not do any operations on the GPU, we found that with the frame-rate reduction step, the auto-labeler performed quickly enough that it did not warrant us writing our own implementation. We chose agglomerative clustering because it allowed us to do clustering based on a maximum distance threshold between a new encoding and existing groups. We initially chose a distance metric of *1.1* because this value worked well for other researchers as a threshold [16]. Through some experimentation we found that this value created many small groups of the same identities so we increased the distance threshold to *1.5*. We defined each identity as a unique id generated from the UUID library included with Python.

After the identities are assigned to each detected face, we save the annotations to the file system in a MSCOCO video format that CVAT is able to read [6]. This format has three major array entries that we can use within our simulation: *categories*, *images*, *annotations*. Each category entry contains an *id* value and *name* value. Each image within an entry contains a *file_name* and *id* entry as well as its size in pixels. Each annotation references the *id* of the image and contain information such as the identity string we generated as well as its location within the referenced image.

3.1.3 Split-Merge Annotation Tool

After the auto-annotation step, it is important that we correct any wrong predictions because as mentioned before, pre-trained FaceNets don't perform perfectly on

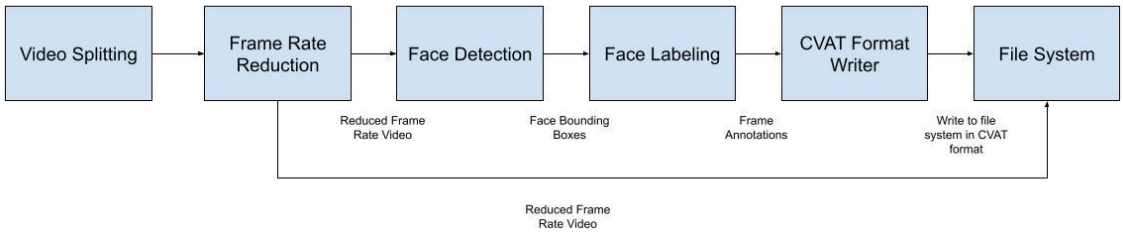

Figure 3.1: Auto-Annotating Pipeline for Simulation

classroom datasets. This can be seen in figure 3.2 where there are three different identities that the auto-annotator assumed was the same person. In order to reduce this editing effort, we wrote an annotation application in Python using the Tkinter GUI library [5]. We chose this combination of language and tools because a major goal of this application was to quickly start labeling so we could run our simulations as early as possible.

This tool reads in directories with the MSCOCO video format described previously and displays representative images of all the identities within the *Group Overview Screen* 3.3. From this screen, annotators can select existing groupings then merge them, save the revised annotations using the confirm button, cancel their annotation and go back to the loading dataset screen or click on a representative image to edit a grouping. Merges can only be done from this screen.

When the annotator clicks on an image they are taken to the group screen which can be seen in figure 3.2. From this screen, annotators are able to select images and label them as False Positives (FP), Back of Heads (BH), or split the selected images into a new identity. In the case that all the images in that group are False Positives or Back of Heads, the annotator can click the *Select All* button and click the corresponding button. After the annotator is pleased with the grouping, they can click the *Back* button to go back to the *Group Overview Screen*.

Overall, this tool made annotating the dataset used in the FaceSim much easier.

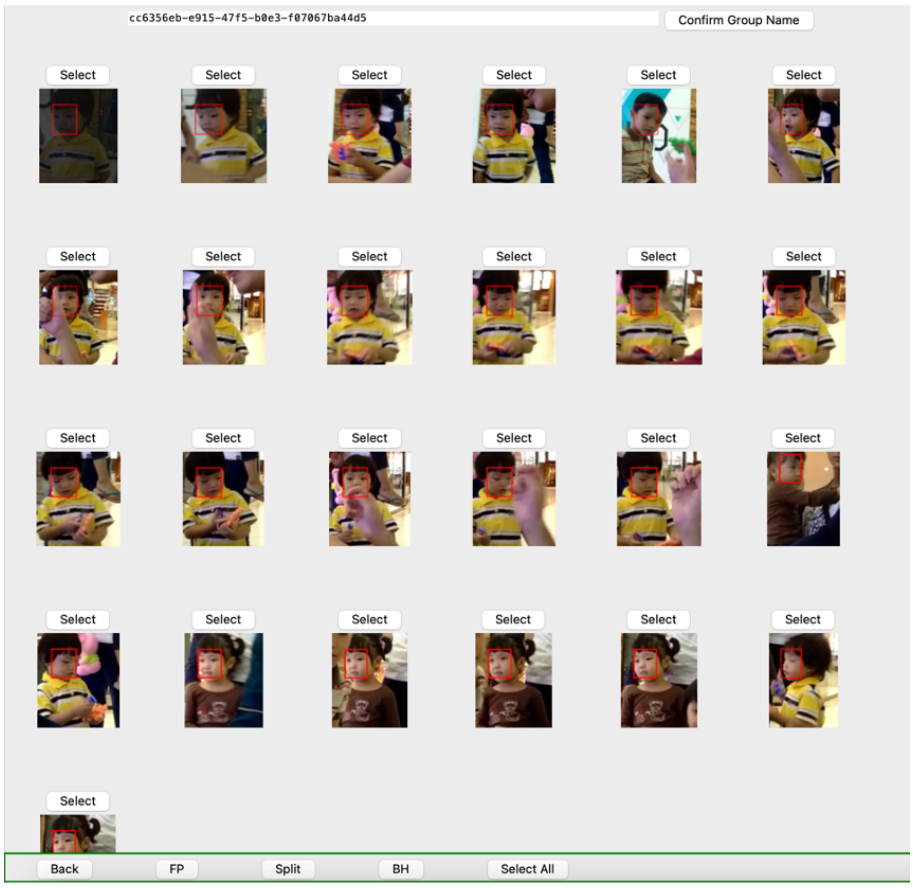

Figure 3.2: Group Screen

However, it did have a few weaknesses. Because this tool only showed the face and torso's of the people we wanted to label, we lost important information such as what the person was doing. For higher resolution images, this was not that big of a problem, however when we had lower resolution images (caused by smaller detected faces which were scaled up), it was sometimes difficult to identify whether two images were of the same student. This is particularly true because many times, the tie-breaker for whether two students were the same person was the color of their shirt in conjunction with their facial features.

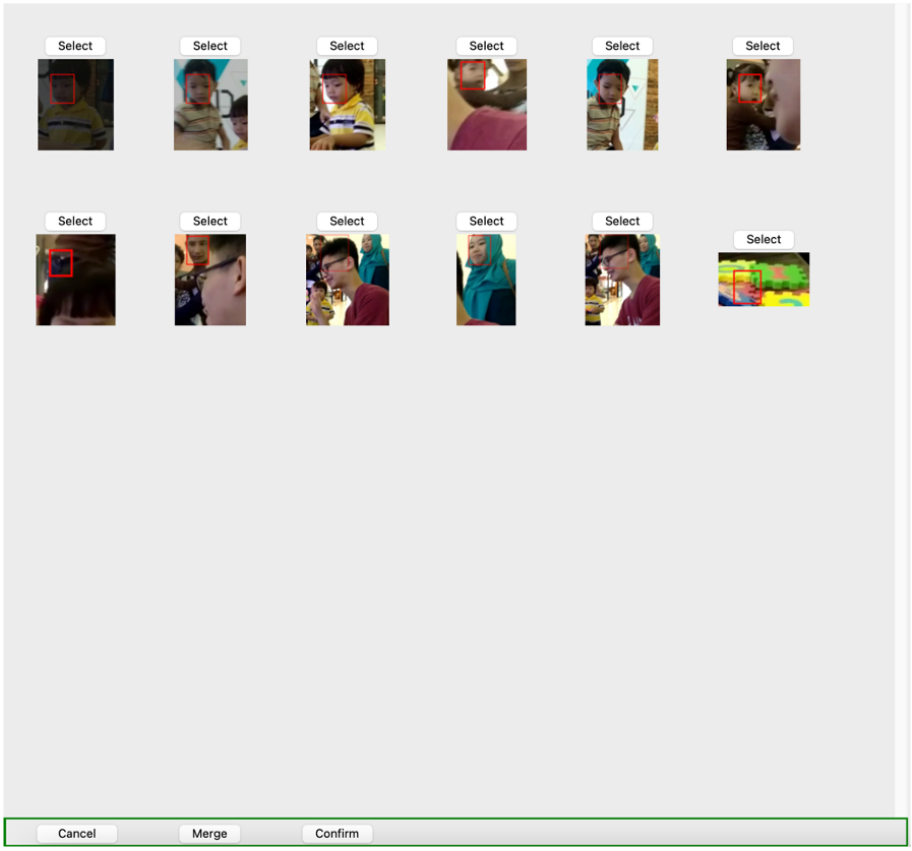

Figure 3.3: Group Overview Screen

3.1.4 Dataset Summary

A summary of our dataset's properties can be seen in table 3.1. From this summary we can see that with the auto-annotation method we were able to label 30,622 frames with 37,792 total annotations. Because the frame rate of the videos was reduced to 2fps, this means that we annotated 255 minutes of classroom footage using the described process. These annotations are attributed to 267 different sub-videos from 25 full length videos.

In order to speed up development without testing experiments on the full dataset, we took a two video subset of the full dataset to create a *Toy Dataset*. This Toy dataset contains 1,293 annotations within 1,363 different frames.

Field	Mean Count	Count STD	Dataset Total
# identities	6.38	4.38	1,704
# False Positives	31.72	30.03	8,470
# Back Of Heads	.49	1.65	132
# Annotations	141.54	112.88	37,792
# Frames	114.69	19.41	30,622

Table 3.1: Full Dataset Per Sub-Video Summary Statistics

3.2 VggFace2

The VGGFace2 dataset is a dataset of 3.31 million face images of 9131 different subjects downloaded from google image search collected by the University of Oxford. There are on average 362.6 different images per each identity [7]. This dataset was used to train the off the shelf facenet model we used for our experiments. However, we were not able to acquire this dataset as the link to download it is no longer active.

3.3 Labeled Faces in the Wild

Because our uncertainty model training required the use of a source dataset and we were not able to acquire the VGGFace2 dataset, we decided to treat the older Labeled Faces in the Wild dataset as our source dataset instead. This dataset contains 13,233 images scraped from the web of 5,749 people. 1,680 of these people have more than one image associated with them [11]. We decided to choose this dataset both Labeled Faces in the Wild and VGGFace2 suffer from the same class imbalances.

Chapter 4

Experiments

In this section we will detail the various experiments we ran to evaluate which strategies reduce operator load. We employed different strategies including using off the shelf models and retraining these models. We also explore how the number of exemplar data points affects final classification performance as well as how labeling only uncertain encodings affects model accuracy.

4.1 Data Split

In order to evaluate our model training over the course of the experiment, we took a small holdout set of 3 videos from our classroom dataset. This test set was used to track the loss during our retraining experiments seen in section 4.4.3 and select our best uncertainty model in section 4.7.3. We decided to remove this set from labeling for all of our experiments because we wanted to keep the dataset consistent for all experiments.

4.2 Evaluation Metrics

Over the course of our experiments, we used two different metrics to evaluate the effectiveness of different strategies: accuracy and number of relabeling actions. Throughout the experiments, accuracy was meant to measure the correctness of model predictions and having a higher accuracy was better. We will discuss two different definitions of accuracy which we used based on experiment in this section. Unlike accuracy, the number of relabeling actions was instead used to understand how using models affects the time annotators take to label datasets.

The first definition of accuracy is what we used for the experiments in sections 4.3 and 4.4. This definition of accuracy can be calculated using $acc = \frac{n_{correct\_pred}}{n_{pred}}$ where $n_{correct\_pred}$ is the total number of correct predictions and n_{pred} is the total number of predictions. Having a higher accuracy meant that our pipeline correctly predicted more faces.

Our next definition of accuracy was used for the experiments in sections 4.6 and 4.7. It is meant to measure how correct a final dataset's labels are. We measure accuracy here because within these experiments we were evaluating whether it is necessary for an annotator to label every face or if it is sufficient to label a subset of faces. To calculate the accuracy here, we look at the labels of the entire dataset (both machine generated and annotator-labeled) and divide them by the total number of labels. The formula for it can be seen in equation 4.1. Here $n_{annotator\_labels}$ is the number of faces the annotator labeled, $n_{correct\_machine}$ is the number of faces annotated correctly by the pipeline and $n_{machine\_generated}$ is the total number of predictions.

$$acc = \frac{n_{annotator\_labels} + n_{correct\_machine}}{n_{annotator\_labels} + n_{machine\_generated}} \tag{4.1}$$

To make this version of accuracy more clear, we give the following example. If

a video has 100 faces within it and an annotator annotated 50 exemplars, and our classification pipeline correctly labeled 25 other faces, the accuracy on that video would be 75%. Because the 50 labeled faces are automatically correct. This means that 25% of the faces for that video were attributed to the wrong person.

The final metric we use is the count of relabeling actions an annotator needs to do. This metric is used in sections 4.3, 4.4 and 4.5. We use this metric because it allows us to estimate how different models affect the user experience. In section 4.5 we also use this metric to estimate how much time can be saved by using a model as opposed to having an annotator label every example.

4.3 Baseline Experiments

In this section, we will detail the experiments we used to get baseline performance for our models. We use the results from these experiments to evaluate whether the strategies employed positively impacted model accuracy and operator load.

4.3.1 Relabeling Experiment

The simplest experiment we wrote is the *RelabelingExperiment* which had the *RelabelingAgent* change all machine generated annotations. The goal of this experiment was two-fold: to validate that our architecture integrated together correctly and to get a baseline of what would happen if the annotators needed to manually annotate each face image. We expected this experiment to have the most annotator actions of any other experiment we ran. Because there is no model backing the Relabeling Experiment, we don't include model accuracy in our analysis. For analysis on how this experiment's labeling actions compared to other strategies, see section 4.5.

4.3.2 Random Guessing Experiment

The next experiment we wrote was the random guessing experiment. This experiment utilized the *RandomPipeline* class in order to randomly select a valid identity for each face. To keep the number of identities consistent between this pipeline and other pipelines, we followed the same initialization process. The goal of this experiment was to get a baseline accuracy for facial recognition models. The overall accuracy for this model was about 20%. For analysis on how this experiment's labeling actions compared to other strategies, see section 4.5.

4.4 FaceNet Based Experiments

In this section we will detail the experiments we ran to evaluate how incrementally retraining an off the shelf FaceNet model on our classroom dataset affected its performance. We will also discuss the pipeline we designed.

4.4.1 Pipeline

Our pipeline consists of the following three components:

1. Image Processing Step
2. Metric Encoding Step
3. KNN Classifier

For our Image Processing step, we first crop out the faces based on the labels file for each sub-video. We do this because we want to avoid the overhead of re-using the MTCNN model to detect faces that we already detected. Using the labels file also allows us to parse out false positives and back of heads which would not be encoded

correctly anyways. After we crop the faces out, we resize them to be 160x160 because this is the dimension our encoder model expects as an input.

After our Image Processing step, we run the processed faces through our selected FaceNet model [9]. This step takes in the 160x160 faces and outputs a 512 dimensional vector for each face.

After running our face images through our FaceNet encoder model, we take the encodings and process them using a KNN model where K=1. We chose K=1 because it was a simple value to use and we wanted to evaluate the encoder performance more than KNN performance for these experiments. This KNN predicted the identity of each face based on already seen faces so if a face was not already seen, then it would automatically predict the wrong identity. To solve this issue, at the beginning of each sub-video, we would find 1 exemplar for each identity and present it as the first frame to be labeled. This allowed us to warm up our KNN.

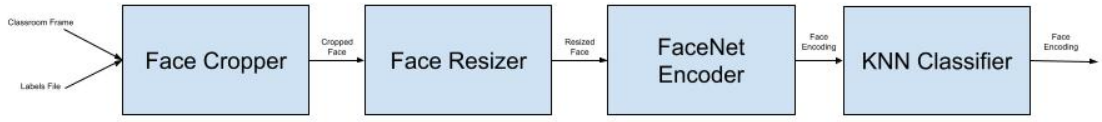

Figure 4.1: Pipeline we used for our FaceNet based experiments

4.4.2 No Retraining Editing Experiment

The next experiment we tried was to utilize an off the shelf FaceNet model and not retrain it at all. The goal of this experiment was to get a baseline accuracy of a model trained on non-classroom datasets and to understand how well it transferred to classrooms and how that affected the number of labeling actions. We found that this experiment performed the best with an **88%** accuracy on faces and about **10X** reduction on the number of labeling actions compared to the random guess

experiment.

4.4.3 Triplet N-Videos Retraining Experiment

The *Triplet N-Videos Retraining Experiment* involves taking an off the shelf FaceNet model and retraining it after n sub-videos using triplet loss. The goal of this experiment was to understand how batch training methods using standard triplet loss can improve our encoding model. Our results did not show any improvement compared to using just an off-the-shelf model.

To try to improve our retraining performance, we tried to find better hyper-parameters for the retraining. To find better hyper-parameters, we implemented a grid-search for the following hyper-parameters: *batch size*, *triplet margin*, *number of epochs*, and *learning rate*. We've found that by increasing the batch size and reducing the triplet margin we were able to get slightly better performance. However, these models still perform worse than the off-the-shelf model.

Because every retraining job takes between three to six hours to complete, and there were left over computational resources during this time, we decided to take a multi-threaded approach to complete the grid search quicker. In order to do this, we spun up 4 threads concurrently and every time a thread finished a job, it would take a new one from the job queue.

After running our grid search, we found that our highest performing model achieved an 84% accuracy over the course of the simulation. This model was retrained every 5 subvideos for 10 epochs with a triplet margin of .5.

We believe that the retraining might have reduced overall performance because the off-the-shelf model had reached a local loss minima which the retrained model had jumped out of. In order to find a better minima, it is possible that we would need a more aggressive retraining strategy where we trained for a longer period.

However, because of the time constraints associated with running the retraining and presenting this new model for use by the annotator, this would not be feasible in a real time environment. This result also shows that adapting an existing model to new data is not as simple as just retraining it on newly labeled data.

4.5 Action Reduction

Having a model predict identities before presenting it to an annotator drastically reduces the number of relabeling actions the annotator needs to do. For example, we can see in 4.2 that by simply using an off the shelf model we are able to reduce the number of annotator actions by almost a factor of 10. We can also see that the model accuracy reduction caused by retraining results in more annotator actions.

If we assume that it takes the annotator about ten seconds to annotate each example (as they would need to search through a gallery of face images), it would take the annotator about 65 hours to complete annotation of our entire dataset. Using the same time estimation technique, we can estimate that it would take about 7 hours for the annotator to correct annotations from an off-the-shelf FaceNet model. This estimation also does not take into account that using the off-the-shelf FaceNet model, we would be able to also provide a ranking list of faces for the annotator, reducing the amount of time they spend searching for the correct identity. Because of this, we expect that the relabeling strategy we propose in this work would actually reduce the wall clock time even more than our estimate.

4.6 Effect of Exemplars On Model Performance

In this section we will analyze how the number of exemplars maintained by the KNN affects the performance of our pipeline. In order to do this, we ran both the *Random*

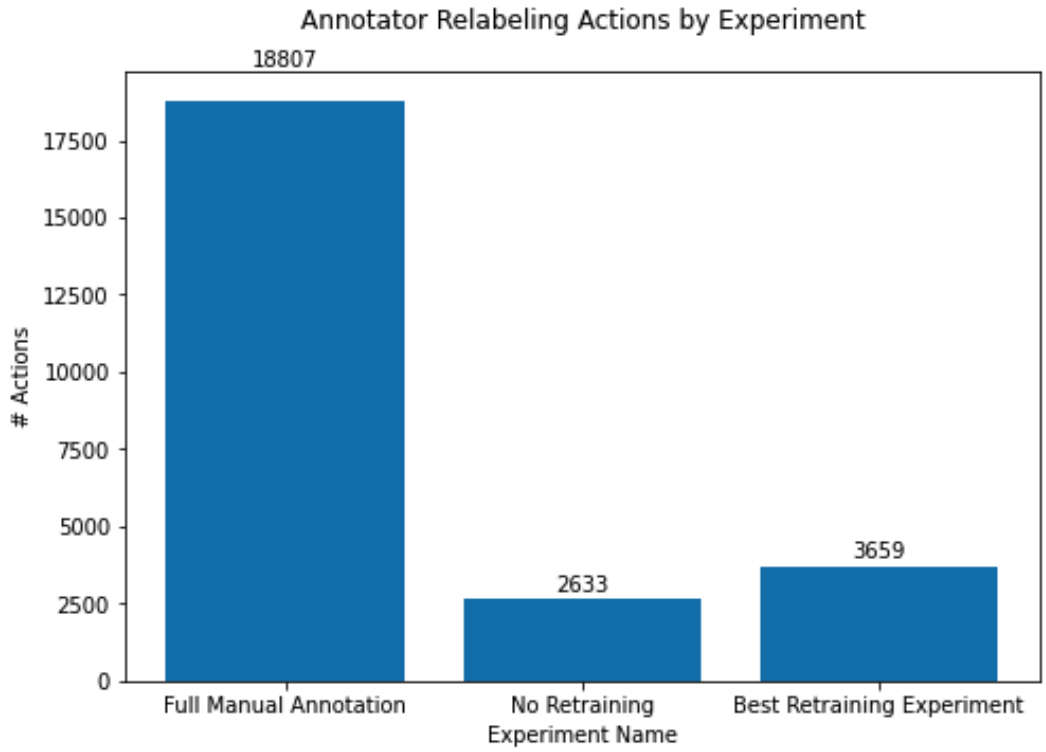

Figure 4.2: Annotator Relabeling Actions by Experiment

Guessing Experiment and the *No Retraining Editing Experiment* to produce a csv file which contained the predicted encoding as well as the ground truth label. We used the *Random Guessing Experiment* as a baseline to validate that the performance gain from using a KNN is significant.

After getting these csv files, we run the encodings for each video through a KNN implementation, incrementally adding 1 more exemplar and evaluating its within video accuracy for that many exemplars on all the examples from that video. Because we evaluate on on the examples in the video, as we get a larger number of exemplars we expected to approach 100% accuracy for the *No Retraining Editing Experiment* output.

In figures 4.3 and 4.4 we plot the average accuracy of each sub video with n exemplars. The grey area above and below the line represents the standard

deviation of the prediction accuracy for all the videos. Because each video has a different number of ground truth labels, videos that had fewer than n labels were not considered for n ground truths.

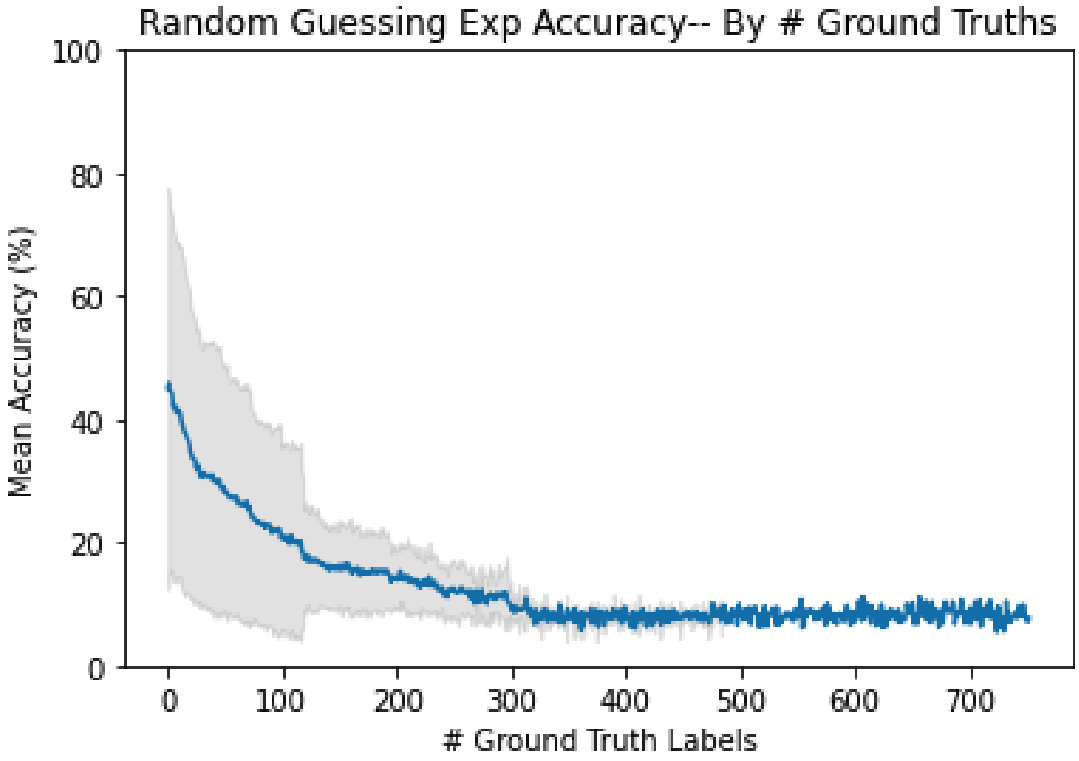

Figure 4.3: With random guessing, the number of existing ground truth labels does not affect the accuracy of the classifier

As can be seen from figure 4.4 it does not take that many exemplars to see a sharp rise in performance. For example, with 1 exemplar, we can see about 50% accuracy. However, when we reach about 50 exemplars, we end up with an average accuracy in high 80 percent range. This shows that KNN is well fitted for the task of classifying FaceNet encodings.

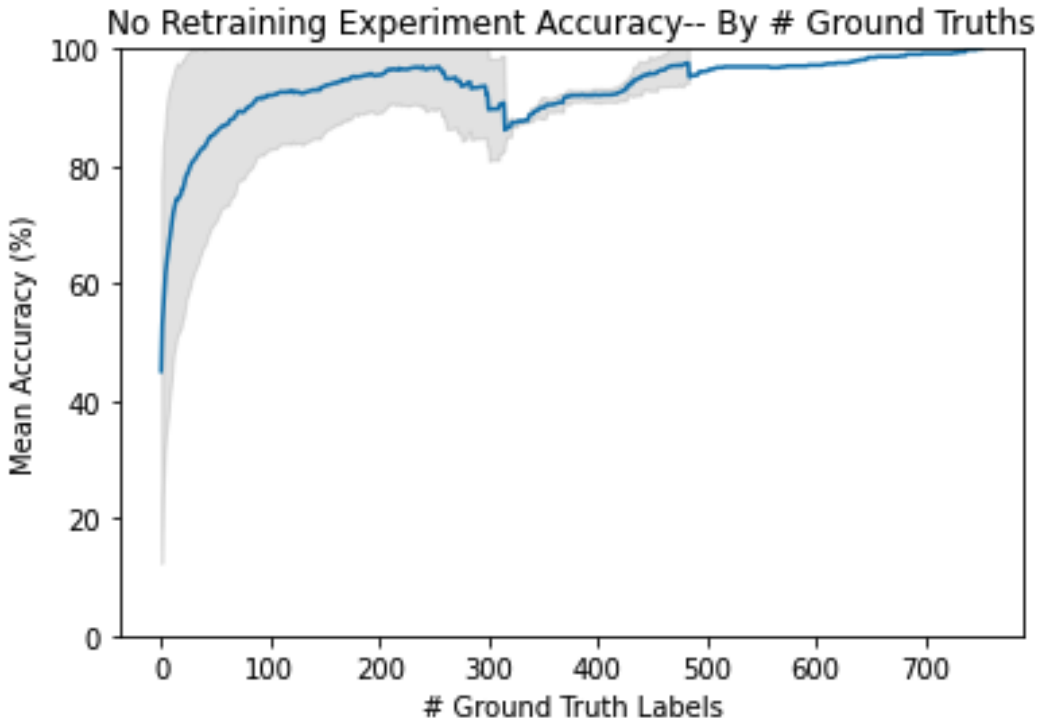

Figure 4.4: When increasing the number of exemplars, the accuracy of the KNN begins to improve

4.7 Uncertainty Experiment

Our final experiment attempts to answer whether labeling only the most uncertain examples improves our model accuracy. To do this we train an uncertainty model detailed in section 4.7.1 and select the best ones for use in the labeling simulation.

4.7.1 Uncertainty Model

Our uncertainty model consists of 2 components: an encoder model which is the model we want to quantify the uncertainty of and two heads that take in the encoder's output and output encodings of the same dimensionality. The two heads each consist of 2 linear layers seperated by a ReLU. For our purposes, the encoder model is the off-the-shelf FaceNet model described in the section 4.4.2. The two heads are

initialised with separate random values and are trained using the methods described in section 4.7.2. Similarly to [17], we theorize that if the two head's encodings are drastically different, the encoder model is uncertain about that prediction. The uncertainty model can be visualized in figure 4.6.

A naive way to quantify the uncertainty of a prediction is to take the L_2 distance between the encodings produced by the two heads. However this leads to the potential that the outputted encodings from both heads can just be rotations of one another. An example of this can be seen in figure 4.5. Here we can see that even though the two encodings are rotations of each other, the distance between the corresponding points would be large. This is an important failure case because if the encodings are rotations of one another then this actually means that the encodings would be classified the same and it would not actually indicate uncertainty.

To solve the above problem, we take the output of the two heads and input them into $CE(\sigma(y_a), \sigma(y_b))$, where y_a refers to the vector produced by equation 4.2 with encoding model Z_a as an uncertainty measure. In equation 4.2, $p(i)$ refers to the encoding of a known identity i. σ refers to the softmax function. By using this function, we get a normalized vector that takes the distance of each identity from the encoded image into account. We can then use the cross-entropy (CE) between two softmax outputs in order to measure the agreement between the two encoding function. A high cross entropy would indicate that the two models disagree while a low cross entropy would indicate agreement.

$$y_j = \text{dist}(Z_j(f), [Z_j(p(1)), ..., Z_j(p(n))] \tag{4.2}$$

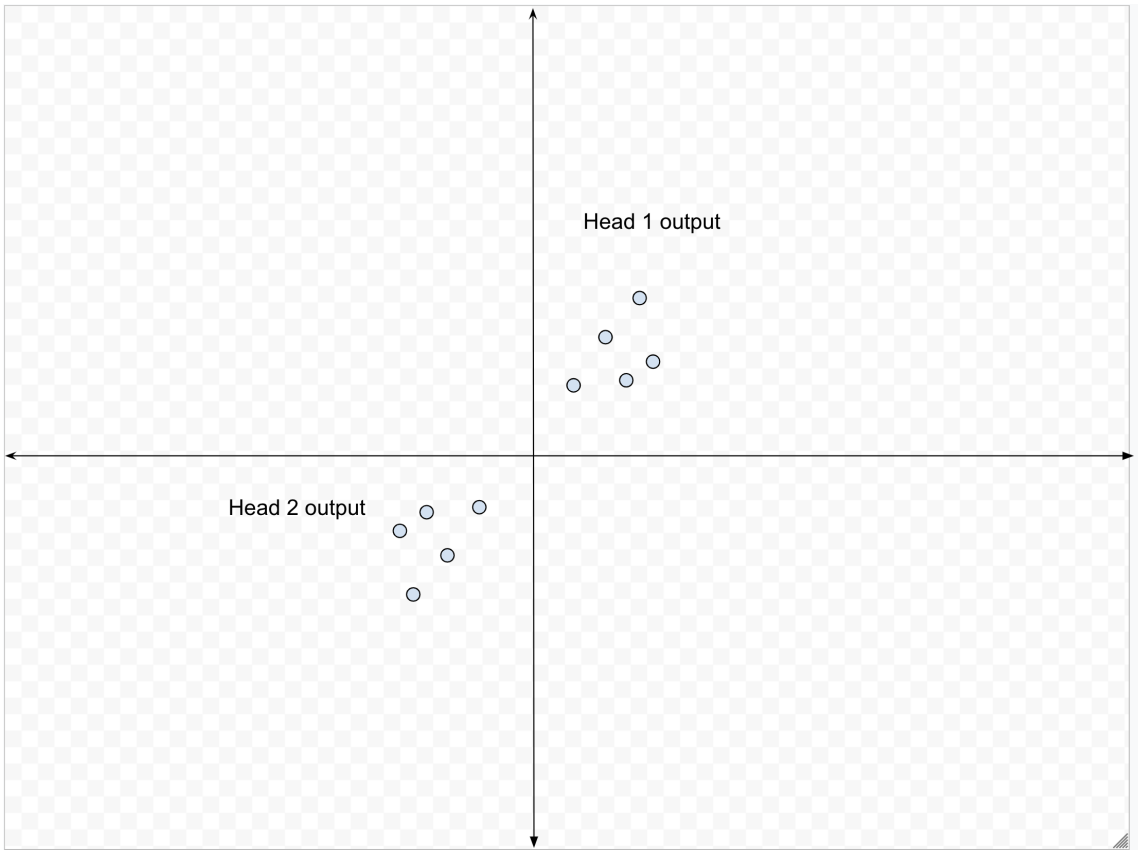

Figure 4.5: Here Head 2's output is a rotation of Head 1's output

4.7.2 Uncertainty Model Training Regimes

In order to train our uncertainty model, we first initialize the two heads separately using random weights. We also freeze our encoder model because we want to quantify uncertainty for the off-the shelf model and tuning the encoder would lead to a different model that we're quantifying uncertainty on. We have three training procedures which we will detail in this section. For all of our training procedures, we treat each face in a batch as an individual identity and compare the uncertainty of each face to the others. This allows us to also train on faces that are not labeled.

The first training regime we used was to simply tune the two output heads to produce encodings with a small triplet loss. For this regime, we treated the Labeled Faces in the Wild dataset as our source dataset for the reasons mentioned in section

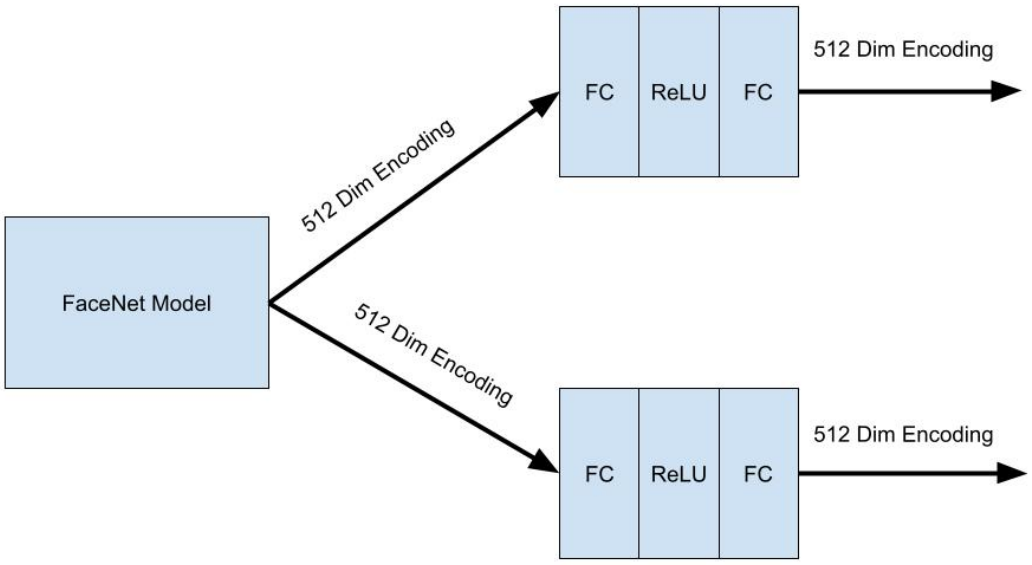

Figure 4.6: Architecture of our uncertainty model

3.3. Because we did not optimize for discrepancy in this regime, we expected the encodings of the two heads to not have high discrepancies.

The second training regime we employed was to combine both triplet loss and treat our discrepancy formula as a loss function modified by a parameter α that represented the weight given to the discrepancy loss. For this regime, the triplet and discrepancy loss work against each other since they are trying to achieve different objectives. We hoped that by adding them together with a well chosen α value we would be able to strike a balance between the two. This loss function can be seen in equation 4.3. In this equation, X_i refers to the the face at index i of the batch (e.g i=0 means the first face within the batch). X refers to the entire batch.

$$\mathcal{L}_{trip\_disc} = triplet\_loss(X_i) - \alpha * discrepancy(X_i, X) \tag{4.3}$$

For our final training regime, we took the three best models trained using our other training regimes and optimized using only discrepancy loss as a loss function on our full classroom dataset. Our strategy for choosing the best model is detailed in section 4.7.3. This strategy is akin to UDA methods which are run on the entire target dataset prior to predicting classes for each image.

4.7.3 Uncertainty Model Selection

All of our training regimes attempted to achieve a balance between meaningful triplet encodings and uncertainty of the two heads. In order to quantify this balance, we utilize a t-score to compare the uncertainty of the two heads between examples classified correctly by the encoder model and those classified incorrectly.

To classify our data points, we used a hold-out test set of 3 full videos from our classroom dataset which we had removed from all of our other experiments. We then took the encodings from the encoder and ran them through the same KNN configuration we used in section 4.4. We also use the same strategy for selecting representative items in the batch as we did in the experiments in section 4.4.

After getting the two sets of uncertainty scores (correct and incorrect), we use equation 4.4 to calculate the t-score between the correct and incorrect discrepancies. Here $\overline{disc_{correct}}$ corresponds to the mean of the correct discrepancies and $\overline{disc_{incorrect}}$ corresponds to the mean discrepancy of the incorrect prediction. Similarly, $var_{correct}$ corresponds to the variance of the correct discrepancy predictions and $var_{incorrect}$ corresponds to the variance of incorrect predictions. $n_{correct}$ and $n_{incorrect}$ is the number of examples in the correct and incorrect uncertainty vectors.

$$t = \frac{\overline{disc_{correct}} - \overline{disc_{incorrect}}}{\sqrt{\frac{var_{correct}}{n_{correct}} + \frac{var_{incorrect}}{n_{incorrect}}}} \tag{4.4}$$

After getting the t-score for each model, we select the 3 best models from each training regime based on the largest t-score. This results in 9 models that we are able to feed into the experiment for our next section.

4.7.4 Uncertainty Experiment

After selecting our best 9 models, we wanted to evaluate whether labeling only the most uncertain examples improves model accuracy. To do this, we created a new *VideoLoader* class which looks at all the faces from a sub-video and uses the uncertainty model to detect the top 100 most uncertain face predictions. These 100 examples then get bubbled up to the agent for labeling so that we can get ground truth labels. Afterwards, we used these 100 examples to predict the classes for the remaining examples. We analyze the accuracy using the same method as in section 4.6. Being able to just change the video loader for this experiment is an example of how our modular architecture helped us develop experiments quickly without needing to rewrite large chunks of code.

4.7.5 Uncertainty Experiment Results

In this subsection we will analyze whether presenting only the most uncertain examples for relabeling presents a benefit compared to labeling random faces. First we will analyze whether this strategy improved overall accuracy by comparing the results to that of section 4.6. Afterwards we will analyze the effect of t-score on the final test time accuracy to understand whether this metric is useful in selecting the best model. Finally we will analyze the results of individual training regimes to

select the best regimes for an uncertainty Model.

In figure 4.7 we show how choosing only the most uncertain exemplars affects the model accuracy on all examples. As shown in the graph, on average with 100 exemplars, using the uncertainty model performs slightly worse than if we had randomly chosen our 100 exemplars. However it is important to note that the discrepancy tuning regime reduced this average while the other two training regimes on average approached the average accuracy of randomly selecting the exemplars.

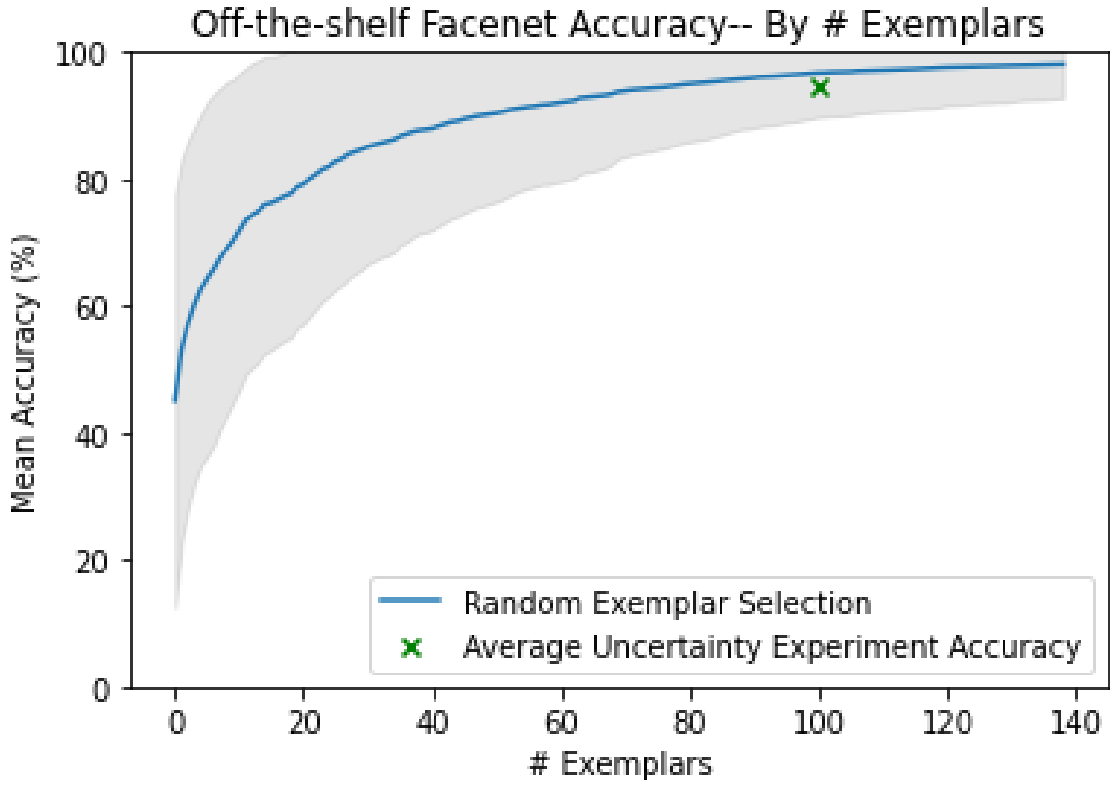

Figure 4.7: Effect of Choosing Most Uncertain Exemplars

In figure 4.8 we show that using a t-score is in fact an effective way to measure whether an uncertainty model is effective. We conclude this because as the t-score increases, so does the average accuracy on 100 exemplars.

Table 4.1 shows the average t-score and accuracy associated with each training

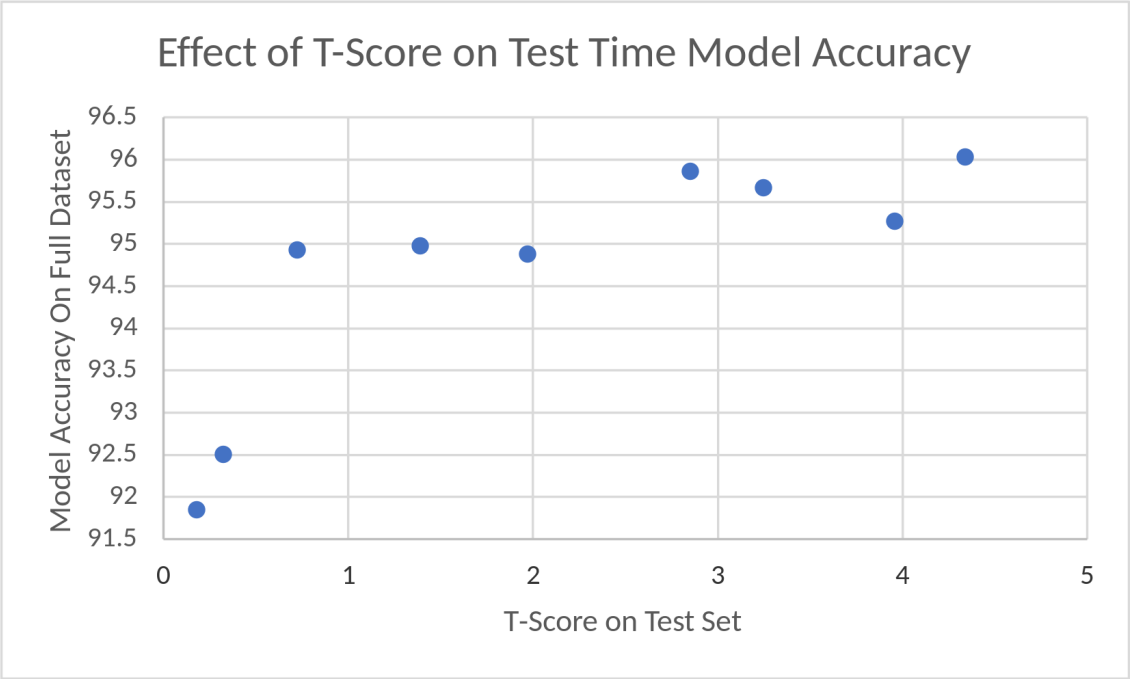

Figure 4.8: T-Score is an effective way to measure effect of uncertainty model

Training Regime	Average T-score	Average Accuracy
Triplet Loss Regime	2.15	95.31%
Combine Triplet and Discrepancy Regime	3.53	95.60%
Discrepancy Tuning Regime	0.82	93.08%

Table 4.1: Average Uncertainty Experiment Performance by Training Regime

regime. From this table, we can see that both the regular triplet loss regime and the combined triplet and discrepancy loss regimes performed relatively similar. Both of these average accuracies approach the value of 96% shown by selecting 100 examples at random. This shows that with a better training regime or potentially better hyper-parameters, labeling only the uncertain examples might lead to better performance. However, when introducing a pure discrepancy tuning step after these steps, our performance dropped drastically. One potential cause of this is that this tuning might have increased the discrepancy of the two heads at the expense of them producing meaningful encodings.

Chapter 5

Conclusion

By developing a custom annotation tool and workflow, we were able to quickly annotate a dataset for facial recognition tasks in the classroom. We were able to take this custom dataset and use it within our extensible simulator and explore various strategies for reducing the work annotators need to do to label datasets for facial recognition tasks.

We demonstrated that introducing an off-the-shelf model to annotate faces and having an annotator label these faces can significantly reduce the time it takes to annotate datasets for facial recognition. By using the classroom domain, we are able to challenge these off the shelf models to understand how well they are able transfer to unseen data using the simulation we wrote. We were also able to show that this problem is more complex where simply retraining on new data from the target domain is not a simple solution.

By using a combination of an off-shelf model and our custom annotation tool we were able to quickly annotate a dataset to evaluate our proposed strategies without the need for costly user studies. We were able to use this dataset to show that later examples within a video will be quicker to be annotated than early examples. This

shows that for each labeled video an annotator will spend more time at the beginning than at the end.

We were also able to develop a custom uncertainty quantification process for encoding models and show that uncertainty can be used to potentially improve labels. We were also able to show that this method is resilient to discrepancy heads which produce encodings that are rotations of each other. Additionally, we were able to show that our method for selecting the best uncertainty model is directly correlated with better labeling performance.

5.1 Future Work

One promising next step is using our results to develop annotation software and evaluate how well the strategies developed in this thesis work in the real world. For example, it is possible that an annotator would only need to annotated 100 exemplars from a video and the other examples would be annotated automatically. This would allow more factors such as annotator fatigue to be analyzed. Understanding how strategies such as using an off-the-shelf model affect the wall-clock time to label a dataset is another important metric because while we estimated the time it takes to label a video, external factors might change this. It can also be interesting to quantify whether the merge/split strategy for annotation provides a benefit compared to traditional methods.

It is also important to develop better training regimes for uncertainty models. As shown in section 4.7.5, the accuracy of the datasets increased as the t-score of the uncertainty model increased. However, we were not able to find a reliable training regime for this model. It is also possible that training the uncertainty model with the newly labeled data after n labeled videos would improve its performance so this

option could be explored.

Bibliography

[1] pandas - Python Data Analysis Library.

[2] PyTorch.

[3] pytube — pytube 11.0.2 documentation.

[4] scikit-learn: machine learning in Python — scikit-learn 1.0.2 documentation.

[5] tkinter — Python interface to Tcl/Tk — Python 3.10.2 documentation.

[6] Computer Vision Annotation Tool (CVAT), Nov. 2021. original-date: 2018-06-29T14:02:45Z.

[7] Q. Cao, L. Shen, W. Xie, O. M. Parkhi, and A. Zisserman. VGGFace2: A dataset for recognising faces across pose and age. *arXiv:1710.08092 [cs]*, May 2018. arXiv: 1710.08092.

[8] W. Deng, L. Zheng, Y. Sun, and J. Jiao. Rethinking Triplet Loss for Domain Adaptation. *IEEE Transactions on Circuits and Systems for Video Technology*, 31(1):29–37, Jan. 2021. Conference Name: IEEE Transactions on Circuits and Systems for Video Technology.

[9] T. Esler. Face Recognition Using Pytorch, Feb. 2022. original-date: 2019-05-25T01:29:24Z.

[10] P. Gardias. Learning Deep Social Interactions to Identify Positive Classroom Climate. page 52.

[11] G. B. Huang, M. Ramesh, T. Berg, and E. Learned-Miller. Labeled Faces in the Wild: A Database for Studying Face Recognition in Unconstrained Environments. Technical Report 07-49, University of Massachusetts, Amherst, Oct. 2007.

[12] R. Khurana and A. K. S. Kushwaha. Deep Learning Approaches for Human Activity Recognition in Video Surveillance - A Survey. In *2018 First International Conference on Secure Cyber Computing and Communication (ICSCCC)*, pages 542–544, Dec. 2018.

[13] K. Mei, C. Zhu, J. Zou, and S. Zhang. Instance Adaptive Self-Training for Unsupervised Domain Adaptation. *arXiv:2008.12197 [cs]*, Aug. 2020. arXiv: 2008.12197.

[14] C. Rupprecht, I. Laina, N. Navab, G. D. Hager, and F. Tombari. Guide Me: Interacting with Deep Networks. *arXiv:1803.11544 [cs]*, Mar. 2018. arXiv: 1803.11544.

[15] K. Saito, K. Watanabe, Y. Ushiku, and T. Harada. Maximum Classifier Discrepancy for Unsupervised Domain Adaptation. *arXiv:1712.02560 [cs]*, Apr. 2018. arXiv: 1712.02560.

[16] F. Schroff, D. Kalenichenko, and J. Philbin. FaceNet: A Unified Embedding for Face Recognition and Clustering. *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 815–823, June 2015. arXiv: 1503.03832.

[17] I. Shin, D.-j. Kim, J. W. Cho, S. Woo, K. Park, and I. S. Kweon. LabOR: Labeling Only if Required for Domain Adaptive Semantic Segmentation. *arXiv:2108.05570 [cs]*, Aug. 2021. arXiv: 2108.05570.

[18] H. Wang, S. Gong, X. Zhu, and T. Xiang. Human-In-The-Loop Person Re-Identification. *arXiv:1612.01345 [cs]*, May 2018. arXiv: 1612.01345.

[19] M. Ye, J. Shen, G. Lin, T. Xiang, L. Shao, and S. C. Hoi. Deep Learning for Person Re-identification: A Survey and Outlook. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, pages 1–1, 2021. Conference Name: IEEE Transactions on Pattern Analysis and Machine Intelligence.