

IMPERIAL COLLEGE LONDON

DEPARTMENT OF COMPUTING

AIGVA: AI Generated Video Annotation

MSC INDIVIDUAL PROJECT FINAL REPORT

Author:

Emil Soerensen

Project Supervisor:

Dr. Bernhard Kainz

Second Marker:

Dr. Wenjia Bai

Submitted in partial fulfillment of the requirements for the MSc degree in
Computing Science of Imperial College London

September 2019

Abstract

Most researchers agree that the "best way to make a machine learning model generalize better is to train it on more data" (Goodfellow et al., 2016). However, gathering and labelling training data is a laborious, tedious, and costly task. Although researchers have made significant developments in improving machine learning architectures and developing custom hardware, limited focus has been on designing better ways to acquire and label more training data.

Worse still, the problem of labelling training data becomes more difficult when you transition into the video domain because of the substantial increase in data size. Currently available video annotation software tools require you to label each frame in a video individually which can be prohibitively expensive, often resulting in large amounts of either unused or unlabelled video data. This is unfortunate as video data is crucial for training algorithms in many of the most promising areas of machine learning research such as autonomous vehicle navigation, visual surveillance, and medical diagnostics.

The dissertation aims to address these problems by building a web application that can speed up the labelling process of video datasets. The outcome of the dissertation is the AI-Generate Video Annotation Tool (AIGVA), a web-based application for intelligently labelling video data. A set of novel labelling techniques called Cold Label and Blitz Label were developed to automatically predict frame-level labels for videos. A scalable and distributed software architecture was built to handle the heavy data processing related to the machine learning and video processing tasks. The first in-browser video player capable of frame-by-frame navigation for video labelling was also built as part of the project.

Quantitative and qualitative evaluations of the AIGVA application all show that the tool can significantly increase labelling efficiency. Quantitative testing using the CIFAR-10 dataset suggest that the tool can reduce labelling time by a factor of 14-19x. User interviews with machine learning researchers were positive with interviewees claiming that they had "no doubt it could be used today in our research group". Furthermore, software testing of the web application and Core Data API demonstrate ample run-time performance.

Acknowledgments

I would like to thank the following people:

- My supervisor Dr. Bernhard Kainz and co-supervisor Dr. Wenjia Bai for their valuable support throughout the process
- Giacomo Tomo, Sam Budd, Anselm Au, Jeremy Tan from the Biomedical Image Analysis Group at Imperial College London for providing both technical and non-technical feedback to improve the usability of the application
- Jacqueline Matthew, Clinical/Research Sonographer @ Guy's and St Thomas' NHS Foundation Trust, who volunteered to teach me about the problems of data labelling in a clinical setting
- My family, friends and fellow students, particularly Finn Bauer, who had to suffer through countless hours of discussions about web app development, data labelling and machine learning

Contents

1	Introduction	1
1.1	Motivation	1
1.2	Objectives	3
1.3	Contributions	4
1.4	Outline	8
2	Background	9
2.1	Literature Review	9
2.1.1	The Stagnation of Image Classification Performance	9
2.1.2	Beyond ImageNet: Improving Image Classification Performance	11
2.1.3	The Challenges of New Data Acquisition	13
2.1.4	Methods to Increase Data Labelling Efficiency	14
2.1.5	Sourcing Image Data From Videos	18
2.2	Existing Labelling Tools	20
2.3	The iFIND Project	24
2.4	Ethical & Professional Considerations	26
3	Method	28
3.1	Label Prediction	29
3.1.1	Cold Label	29
3.1.2	Blitz Label	31
3.1.3	Custom Model	32
3.2	User Stories	33

3.2.1	User Story #1: Building a Pedestrian Classifier for an Autonomous Vehicle Startup	33
3.2.2	User Story #2: Improving Standard Plane Detection in Fetal Ultrasound Classifiers	37
3.3	Design Principles	39
3.4	Requirements	43
3.4.1	Project Management	43
3.4.2	Videos	45
3.4.3	Label Prediction	49
4	Implementation	55
4.1	Software Arch. & Tech Stack	55
4.1.1	Core Data API	56
4.1.2	Web Application	63
4.1.3	File Storage & Database	70
4.2	Feature Impl. - Video Management	73
4.2.1	Frame-by-frame Video Player	73
4.2.2	Export Tool	77
4.3	Feature Impl. - Label Prediction	80
4.3.1	Cold Label	80
4.3.2	Blitz Label	91
4.3.3	Custom Model	92
4.3.4	Label Prediction Engine	95
4.4	DevOps	97
4.4.1	Testing	97
4.4.2	Deployment	98
5	Evaluation & Results	100
5.1	Quantitative Evaluation	100
5.1.1	Label Prediction - Cold Label	100
5.1.2	Label Prediction - Blitz Label	106

5.2	Performance Testing	111
5.2.1	Web Application Performance	111
5.2.2	Core Data API Performance	113
5.3	User Interview & Feedback	117
5.3.1	Machine Learning Researchers	117
5.3.2	Clinical Sonographer	119
6	Conclusion & Future Work	121
6.1	Summary of Achievements	121
6.2	Future Work	123
	Appendices	132
	Appendix A User Guide	134
A.1	Installation	134
A.2	User Manual	136
	Appendix B Full Surveys	137
B.1	Machine Learning Researchers	137
B.2	Clinical Sonographer	143
	Appendix C Supporting Material	145
C.1	Folder Structure	146
C.2	Web Performance Testing	147
C.3	Experimentation	148
C.3.1	List of Videos For AIGVA Experiment	148
C.3.2	Model Used For AIGVA Experiment	148
C.3.3	Confusion Matrices for AIGVA Experimentation	149
	Appendix D Supporting Code	154
D.1	Web Application	154
D.1.1	Video Frame Player	154
D.1.2	Export Tool	160

D.2 Label Prediction 163

 D.2.1 Cold Label 163

 D.2.2 Blitz Label 171

 D.2.3 Custom Model 175

 D.2.4 Prediction Engine 176

D.3 DevOps 180

 D.3.1 Docker 180

D.4 File Storage and Database 183

 D.4.1 Database Schema 183

Chapter 1

Introduction

"We're entering a new world in which data may be more important than software."

—Tim O'Reilly, founder, O'Reilly Media

1.1 Motivation

While recent advancements in machine learning has shown its potential to solve complex classification problems, there is still a way to go. For example, in March 2018 Elaine Herzberg was killed by one of Uber's self-driving cars because the vehicle failed to brake in time. The system misclassified the pedestrian Elaine first as an unknown object, then as a vehicle, and then as a bicycle with varying expectations of future travel path (NST, 2019). Examples like this highlight the need to build better machine learning models.

In applied machine learning, supervised algorithms still outperform other models for classification tasks. While researchers have found a plethora of novel methods to improve the generalizability of such algorithms, the size and quality of the training dataset is still a crucial factor for model performance. The process of acquiring and labelling data for training machine learning models is, however, a laborious, tedious, and expensive task. Consequently, most of the recent research focus has been on improving the machine learning models rather than on improving the data collection

and labelling techniques.

The data labelling task becomes even more difficult when transitioning into the video domain, due to the substantial increase in data size. This difficulty increases the cost of data labelling, which often results in large amounts of either unused or unlabelled video data. As many of today's most promising areas of machine learning research, such as autonomous vehicle navigation, visual surveillance, and medical diagnostics, lie within the video domain, improving data labelling is crucial to progress the quality of the algorithms. Any efforts to speed up the labelling process could thus bring significant value to the future of research and real-world application.

While the challenge of labelling video data exists in many fields, this dissertation is particularly motivated by the challenges faced by a group of machine learning researchers developing algorithms using ultrasound data. Specifically, this dissertation is written in partnership with the intelligent Fetal Imaging and Diagnosis (iFIND) research project. Previously, researchers in this team have built algorithms to classify fetal ultrasound scans, a task normally performed by sonographers, medical professionals trained in identifying and interpreting ultrasound images. Classifying fetal ultrasound scans is a critical task to check for structural abnormalities and anomalies in the baby (NHS, 2017).

The iFIND researchers have overall achieved promising algorithmic results that are comparable with most sonographers, but the neural networks developed have struggled to classify certain challenging cardiac views. To solve this problem, more ultrasound data is needed on these specific regions. Fortunately, the research group is already in possession of many hours of ultrasound video. The main struggle remaining is to first label this data for it to be of any value. Although theoretically possible, it would be too time consuming and too expensive to request a sonographer, a profession currently in shortage within the UK, to sift through the massive amount of video. To help alleviate this problem, this dissertation thus aims to build a tool that

can label such data in an intelligent and fast way. We hope such a tool can lead to further improvements within both medical imaging diagnostics and more general domains.

1.2 Objectives

The goal of this dissertation is to develop an application that can significantly improve the speed of labelling video data to be used as training data in machine learning applications. This goal can be broken down into four key objectives:

- Develop a web-based solution to speed up the process of labelling videos
- Develop novel methods to predict image-level labels for all frames in a video
- Build a scalable software architecture to handle the heavy data processing of machine learning applications in a web application
- Develop an in-browser video player capable of frame-by-frame navigation

Additionally, this report is conducted in partnership with the intelligent Fetal Imaging and Diagnosis (iFIND) research project. This research project encompasses a multi-disciplinary team of medical imaging specialists, roboticists, machine learning engineers and clinicians. The goal of the project is two-fold: (1) to develop new fetal scanning technologies that can automatically move the ultrasound probe to the right place, and (2) to improve fetal ultrasound imaging by automating image processing. This dissertation aims to contribute to latter goal by creating a tool that can speed up training data creation and therefore aid in the future development of automated image processing models.

1.3 Contributions

The main contributions of this dissertation are the development of the AI-Generated Video Annotation Tool, the creation of two novel methods of frame-level prediction in videos, the building of a scalable software architecture to handle machine learning training and inference, and the development of a frame-by-frame video player for the web.

The AI-Generated Video Annotation (AIGVA) tool

The AIGVA tool is a web based video annotation tool designed to significantly improve the speed of labelling video data for the use of training data in machine learning applications. It achieves this speed up through a combination of effective design principles, web application innovations, and label prediction techniques. The tool consist of three components: (1) a Python-based Core Data API designed to handle the heavy data processing involved in machine learning, (2) a File Storage component used to store both videos, machine learning models and key metadata information, and (3) a React-based web application that a user can interact with.

Two novel methods of frame-level label prediction in videos

Two novel methods of label prediction are developed: Cold Label and Blitz Label. "Cold Label" is a label prediction technique developed to address the cold start problem (i.e. when no labels exist). The "Cold Label" technique utilizes human-in-the-loop enabled active learning methods combined with transfer learning techniques to address this problem. The core idea behind this technique is to ask a user to label a few important samples from a set of videos and subsequently use those labelled samples to train a classifier that is then used to label all frames. "Blitz Label" is a label prediction technique that leverages pre-existing labels to predict new labels. The core idea behind the technique is that if a user already has labelled some videos, a machine learning algorithm can leverage that embedded information to annotate unlabelled videos. Both of these methods are empirically shown to significantly im-

prove the labelling speed.

Scalable software architecture to handle machine learning training and inference using a web based application

Core to the application are several data and process heavy tasks including: training machine learning models, extracting frames from videos, running inference on hundreds of thousands of frames, and processing videos for metadata extraction. As part of the Core Data API, a complex software architecture was built to handle these tasks using a network of worker-broker jobs. This was accomplished using a Redis message broker and Celery task queue as part to handle concurrent execution across several cores. To further enhance scalability, the Core Data API uses a Gunicorn server to handle client requests using multiple worker processes. The result of this architecture is the ability to run several data-heavy tasks concurrently.

A frame-by-frame video player for web applications

To handle frame-level labelling of videos via a web application a video player that can navigate on a frame-by-frame level is built. No existing JavaScript based video players offer this key functionality. Therefore I opted to build my own player from scratch based on the HTML5 Video Media element specification. This player was demonstrated to work efficiently both quantitatively, using browser performance tests, and qualitatively through conducting user interviews. Furthermore, this player was fitted with several custom-built features, based on user feedback, to further enhance labelling efficiency such as label skipping and visual representations of labels throughout the video.

To showcase the AIGVA tool, I have included a series of pictures below showing how tool can be used in action. Note, these images are computer generated and conceptual in nature.



Figure 1.1: AIGVA App Showcase: Training a label prediction model in-browser

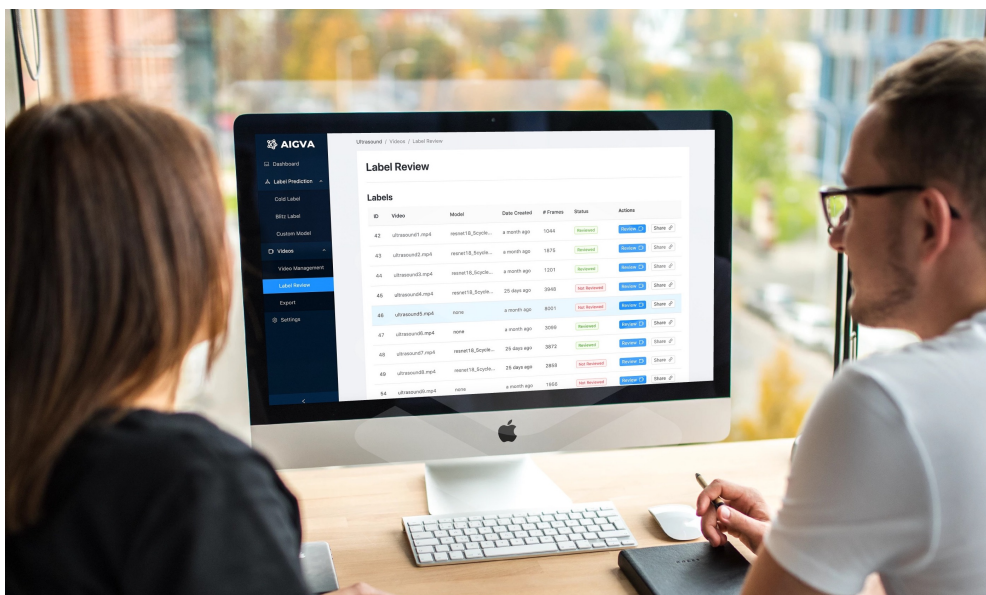


Figure 1.2: AIGVA App Showcase: Collaborative review of labels



Figure 1.3: AIGVA App Showcase: Quickly labelling videos using keyboard shortcuts

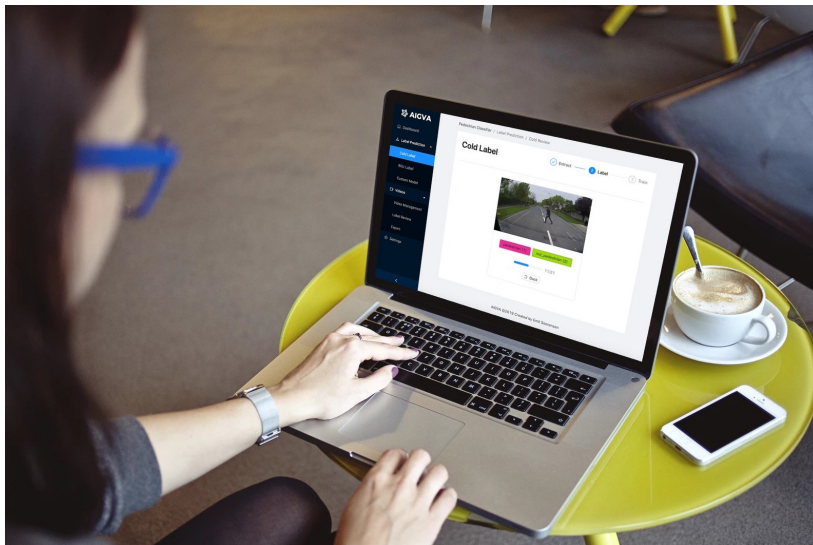


Figure 1.4: AIGVA App Showcase: Manually labelling a few frames per video and training a Cold Label model on those to predict labels for all videos

1.4 Outline

Chapter 2 (Background) reviews the literature on image classification and demonstrates how more labelled training data is key to improving performance. An analysis of existing labelling tools is then conducted. Finally, the need for a better labelling tool is demonstrated by discussing challenges faced by the iFIND ultrasound research project. A brief discussion on ethics closes this chapter.

Chapter 3 (Method) develops the foundation for a new labelling tool called AIGVA. A set of novel label prediction techniques are presented and user stories are created to show how the tool can improve labelling efficiency. Then design principles are developed which are used to create the subsequent functional requirements of the web application.

Chapter 4 (Implementation) discusses the implementation of the AIGVA tool. First, the software architecture and technology stack is developed. Then implementations of two of key features, video management and label prediction, are presented. The DevOps and deployment task is then discussed.

Chapter 5 (Evaluation and Results) presents the evaluation of the AIGVA tool. There are three parts to the evaluation: (1) an empirical evaluation of the label prediction methods, (2) a quantitative assessment of the performance of the application, and (3) a qualitative set of user interviews with machine learning researchers and a clinical sonographer.

Chapter 6 (Conclusion) concludes the dissertation and discusses the future work.

Chapter 2

Background

2.1 Literature Review

To set the development of the AIGVA tool into a wider context, I will review the literature on image classification and its current limitations. I will then describe the current approaches used to address these problems.

2.1.1 The Stagnation of Image Classification Performance

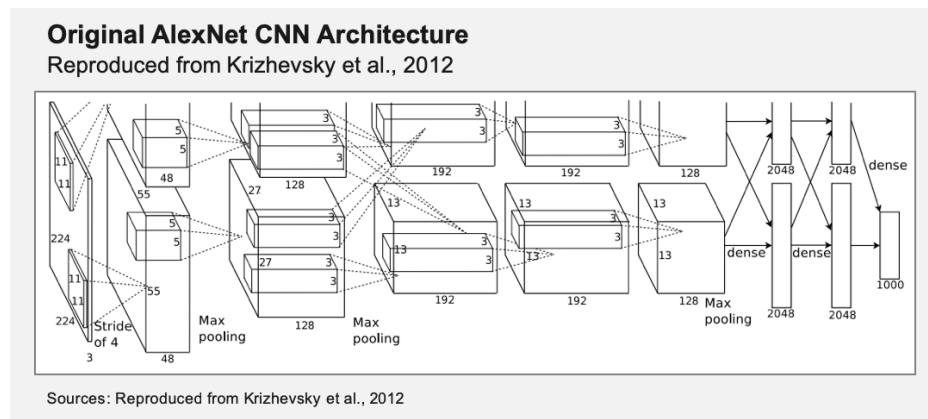


Figure 2.1: Original AlexNet CNN Architecture

Image classification is the task of predicting a label or a set of labels for an image. The task of image classification is one of the central challenges in computer vision and has a myriad of practical uses including: facial recognition, autonomous vehi-

cle locomotion, and medical image diagnostics. Since the introduction of AlexNet in 2012, deep convolutional neural networks (CNNs) have been the predominant architecture for image classification tasks. The innovation of AlexNet was using a deeper and wider architecture than previous CNNs combined with max pooling, fully connected and dropout layers (see Figure 2.1). The performance benchmark for image classifiers is the ImageNet dataset, for which AlexNet achieved a 62.5% top-1 classification rate (Krizhevsky et al., 2012). After AlexNet, many architecture modifications such as VGG (Simonyan and Zisserman, 2014), ResNet (He et al., 2015), DenseNet (Huang et al., 2016) and SqueezeNet (Iandola et al., 2016) have led to significant improvements in both classification rate and model size. The current state-of-the-art performance on ImageNet is EfficientNet-B7, which is 84.4% and was achieved in May 2019 (Tan and Le, 2019).

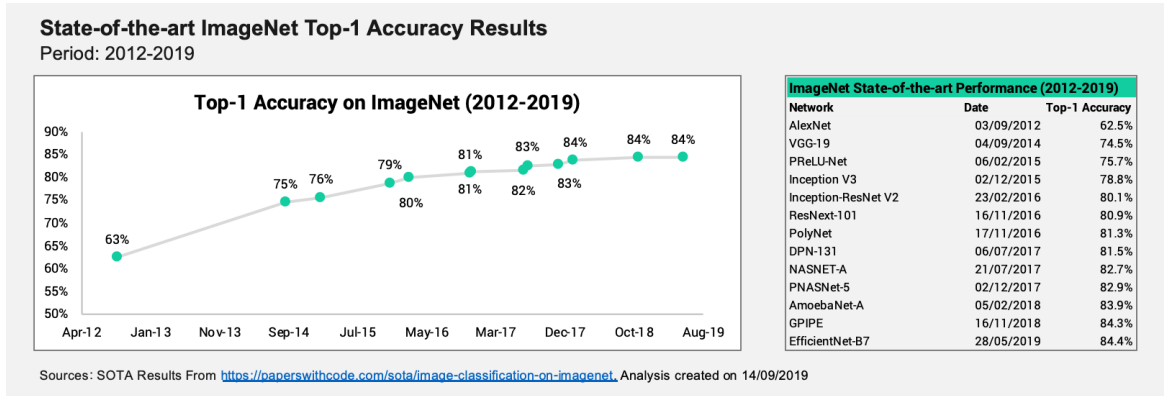


Figure 2.2: State-of-the-art ImageNet Top-1 Accuracy Results

The performance of traditional deep learning based image classification algorithms have slowly begun to stagnate. Figure 2.2 shows the development and performance of state-of-the-art methods for the ImageNet classification task since 2012, with AlexNet on the far left of the chart, and EfficientNet-B7 on the far right. One can observe how the performance has stagnated since mid 2017 with only minor improvements of around a single percentage point. The best performance to date, EfficientNet-B7, only offered a 0.1% total classification improvement to the previous state-of-the-art GPIPE model. Instead of squeezing out additional model performance, much of the work done in the field has focused on "carefully balancing network depth, width, and resolution" to receive near state-of-the-art accuracy with

much lower model size and faster speeds (Tan and Le, 2019). Work has also been done to investigate alternative model training approaches, such as the AmoebaNet-A model, which was trained using an evolutionary tournament-style approach and obtained a 83.9% top-1 classification rate (Real et al., 2018).

2.1.2 Beyond ImageNet: Improving Image Classification Performance

To improve the stagnating image classification performance, researchers are looking at three areas: (1) developing better models with higher capacity, (2) increasing computational power, and (3) acquiring better datasets on which to train (Sun et al., 2017). First, in the area of developing better models, much work has focused on building novel architectures and models. One of the most promising areas within architecture modifications is the use of attention networks (Wang et al., 2017). The core idea behind attention networks is to mimic how humans classify images by focusing on specific substructures within images that have greater explanatory power. Attention networks have been used to produce state-of-the-art results in various tasks such as image-based question answering (Yang et al., 2015), image super-resolution (Zhang et al., 2018), and scan plane detection for fetal ultrasound screening (Schlemper et al., 2018a). Other forms of new architectures include incorporating weights from a pre-trained model through transfer learning (Shin et al., 2016; Zoph et al., 2017) and incorporating probabilistic priors to learn from experience (Ghahramani, 2015). Second, researchers have attempted to increase computational power through hardware and software modifications. As the core computations involved in CNNs are linear algebra manipulations there is an opportunity to distribute and parallelize calculations. Custom hardware is being developed to leverage this parallelizability such as Google’s Tensor Processing Unit (Dean et al., 2018). Similarly, machine learning frameworks that abstract away many of the complexities that arise when dealing with distributed systems running on GPUs and TPUs are being developed such as TensorFlow (Research, 2016).

The final approach to improving image classification is to obtain better training data. This can be done in one of two ways: either by (1) enhancing existing labelled training data, or by (2) obtaining additional labelled training data. First, enhancing existing labelled training data is also called data augmentation, which is a common practice for most machine learning researchers. Augmenting training data, by "cropping, rotating, and flipping input images" can significantly improve the generalizability of models (Perez and Wang, 2017). The key idea behind augmentation is that by manipulating images through various techniques you can still preserve the original label while increasing the variance of your training data. For example, suppose you had an image of cat and flipped it horizontally. After the flip, the picture is still that of a cat, yet is significantly different in terms of its pixel composition, which is what the convolutional neural network uses to train on. Therefore it is possible to artificially generate more training data by performing these manipulations. A more advanced method of image augmentation that goes beyond simple flips and rotations is to artificially generate new images using Generative Adversarial Networks (GANs). These models work by having one neural network attempt to produce counterfeit examples and another neural network trying to predict whether an example is counterfeit. GANs have outperformed traditional methods in augmenting training data in certain contexts (Perez and Wang, 2017).

The second approach to improve image classification by obtaining better data is to acquire additional labelled examples. This involves manually collecting new data and labelling it. Most researchers agree that the "best way to make a machine learning model generalize better is to train it on more data" (Goodfellow et al., 2016). Specifically, retraining a model with more data usually has the effect of reducing variance (Ng, 2015). One recent paper examined the relationship between training data and image classification found "that the performance on vision tasks increases logarithmically based on volume of training data size" (Sun et al., 2017). Another paper found that reducing the amount of ImageNet training data used, unsurprisingly, reduced classification performance (Huh et al., 2016). Obtaining additional

examples is highly effective, but comes with its own set of challenges as we will examine in the next section.

2.1.3 The Challenges of New Data Acquisition

Having established the importance of acquiring new data for training machine learning algorithms we now turn our focus on the challenges of such acquisition. There are two steps involved in acquiring new training data: (1) attaining unlabelled data and (2) then labelling the data. Although these two steps are often done simultaneously, it is useful to distinguish between the two to understand the key challenges. First, the importance of getting unlabelled data is growing as deeper machine learning models require more data to successfully converge. In a recent survey on the challenges in machine learning, the authors found that "data collection is becoming one of the critical bottlenecks" (Roh et al., 2018). In certain fields, acquiring more data can be difficult because of structural restrictions. For example, in medical imaging the data is limited, which makes acquiring sufficient training data to develop classifiers "difficult or even impossible" (Holzinger, 2016). Second, once the data has been acquired, the process of labelling poses its own set of challenges. Labelling certain data may require significant domain expertise such as a sonographer annotating standard fetal scan planes in ultrasound scans. Furthermore, domain experts may not always agree on the labelling of certain challenging examples which produces noise in the dataset. Manually labelling the amount of data needed may also be prohibitively expensive. In summary, to advance the image classification space, "there is a pressing need of accurate and scalable data collection techniques" (Roh et al., 2018).

Not all labels are created equal. To understand where the need for scalable data collection and labelling needs are greatest I have created the framework seen in Figure 2.3. The 2-by-2 matrix plots the two aforementioned challenges of data collection on the two axes: (1) the cost of labelling, which is a proxy for both the time and expertise needed to label an example, (2) the abundance of data, which represents

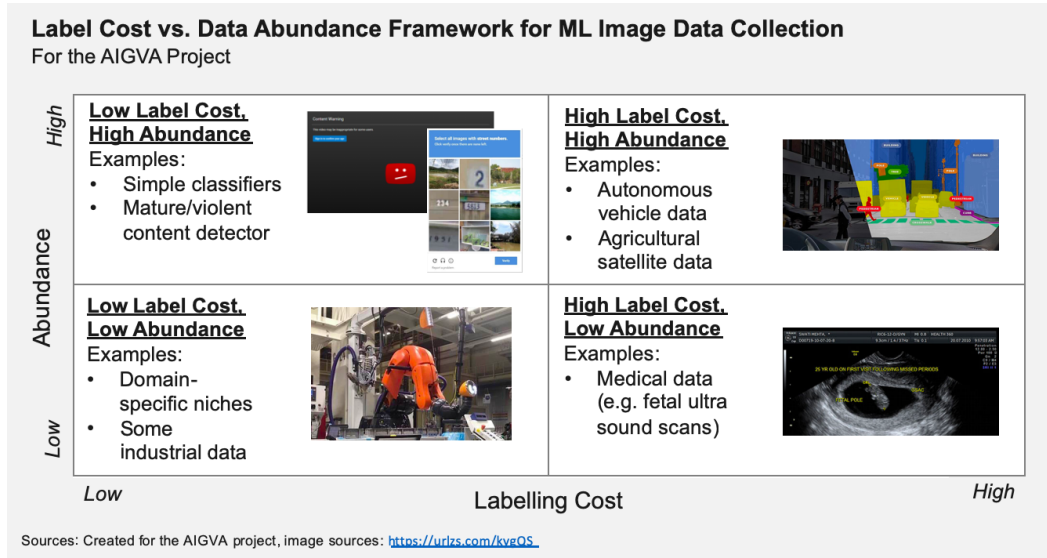


Figure 2.3: Label Cost vs. Data Abundance Framework

the availability of unlabelled examples. There are four categories on the low/high abundance and low/high label cost spectrum. The greatest need for labelling tools is within the two high label cost spaces, since that is where most savings can be found. However, it is important to differentiate between high and low abundance data within the two high label cost categories. High abundance, high label cost data, such as the millions of hours of recorded vehicle driving data used for developing autonomous vehicles, need a labelling solution that offers functionality to find examples of specific high-value events such as car crashes or pedestrian incidents. Low abundance data, high label cost data, such as most medical data, need a labelling solution that makes it as efficient as possible to review all examples, thereby extracting as much meaningful data from the examples as possible.

2.1.4 Methods to Increase Data Labelling Efficiency

"The most accurate way to label examples is to do it manually" (Roh et al., 2018). Unfortunately that is usually not feasible because of the costs involved. The often-used image classification dataset ImageNet consists of millions of images that were manually labelled by humans using Amazon Mechanical Turk (Deng et al., 2009). This process took years and cannot be reproduced by most machine learning re-

searchers and practitioners because of cost constraints. Therefore researchers have proposed various ways to reduce the effort of annotating images. In this section I will discuss three of such methods: human-in-the-loop techniques, active learning and semi-supervised learning.

Human-in-the-loop

Human-in-the-loop is a term used to describe the inclusion of human collaboration in a machine task. The key idea behind human-in-the-loop computing is that "integrating the knowledge of a domain expert can sometimes be indispensable, and the interaction of a domain expert with the data would greatly enhance the knowledge discovery process pipeline." (Holzinger, 2016). Using a human in the loop can be especially useful in certain domains such as medicine, where researchers are often confronted with insufficient data that can be both noisy and incomplete. Here a human may be able to provide valuable insight into the dataset that can improve classification performance. For example, Yu et al. demonstrated how using human-in-the-loop approach can significantly improve the precision of labelling large image datasets (Yu et al., 2015). Additionally, in a 2015 CVPR paper, Russakovsky et al. developed a labelling pipeline using human-in-the-loop techniques and found that using "human verification to update detectors and reduce the search space leads to the rapid production of high-quality bounding-box annotations" (Russakovsky et al., 2015). Work done in the field has also demonstrated how using human-in-the-loop annotations may aid in increasing the interpretability of models (Lage et al., 2018).

Active learning

Close to the idea of human-in-the-loop models is the concept of active learning. The core idea behind active learning is for a model to ask an oracle (i.e. a human) to label the most "interesting" example that a model can learn from. This rests on the idea that not all data is equally valuable to learn from. Therefore one of the core challenges of active learning is figuring out "which questions to pose to humans" and understanding the labelling cost trade-offs involved (Russakovsky et al.,

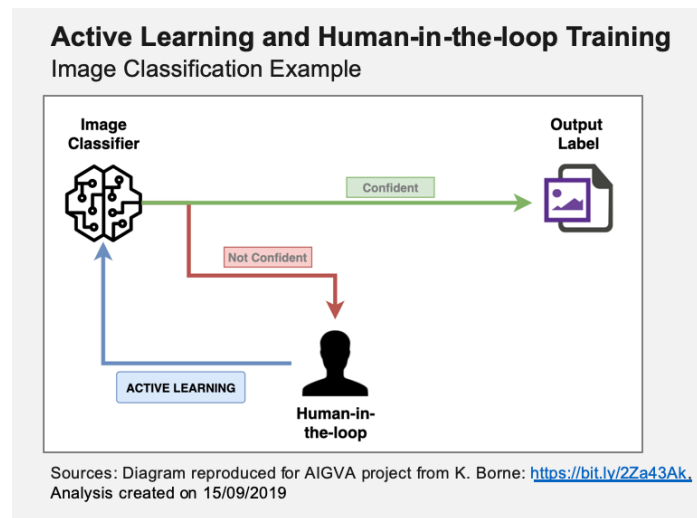


Figure 2.4: Active Learning and Human-in-the-loop Training

2015). Several approaches exist for finding the most valuable examples. First, the method of uncertainty sampling has the model select examples for which it has the least confidence in (Roh et al., 2018). Second, query-by-committee takes uncertainty sampling further by training a committee of models on the same labelled data (Roh et al., 2018). Active learning approaches have been used for image classification to reduce the labelling effort in various ways. Joshi et al. developed an active learning image classification framework using a variation of uncertainty sampling as their query selection algorithm. They empirically tested their framework on the UCI and Caltech-101 dataset and saw "large reductions in the number of training examples required over random selection to achieve similar classification accuracy" (Joshi et al., 2009). In a separate work, Kovashka et al. developed a more complicated active learning framework and found that they could "successfully accelerate" object learning by using significantly fewer images (Kovashka et al., 2011). Human learning and active learning are similar in many respects. The difference is that active learning is a special case of "semi-supervised learning", that makes use of a human in the loop (Roh et al., 2018). See Figure 2.4 for an overview of the key difference between active learning and human-in-the-loop training.

Semi-supervised learning

Semi-supervised learning makes use of existing labelled data to predict unlabelled data. The core idea behind semi-supervised learning is to "exploit labels that already exist" and therefore "generate more labels by trusting one's own predictions" (Roh et al., 2018). With semi-supervised learning the labels may become "weak" meaning that there is some noise in that data. A simple semi-supervised training procedure may work in the following way (Yarowsky, 1995):

1. Train a model on the labelled data
2. Use the model to predict labels for all the unlabelled data
3. Rank the predictions by their confidences
4. Add the most confident predictions to the set of labelled examples
5. Repeat step 1-4 until all data is labelled

Numerous works have examined the performance of semi-supervised technique on image classification and labelling with promising results (Guillaumin et al., 2010; Kingma et al., 2014; Gong et al., 2016). While human-in-the-loop and active learning techniques can be done without any existing labels, semi-supervised learning requires a subset of data to be labelled.

An interesting question within the semi-supervised learning field is just how many (or rather, how few) labels are required to obtain sufficient labelling performance. Early work in the field demonstrated that a handful of examples may suffice if they are of high quality, although this is highly domain dependent (Zhou et al., 2007). In recent years, the question of how to learn from limited information has formed a new subfield within a machine learning called few-shot learning (FSL). FSL models attempt to classify effectively from a limited number of examples. Thus, FSL can significantly help to "reduce the data gathering effort." (Wang and Yao, 2019). Work done on a new few-shot learning architecture called matching networks have proven successful in achieving state-of-the-art results in the field (Vinyals et al., 2016). While

FSL "neither requires the existing of unlabelled samples nor an oracle" it is still often closely related to semi-supervised learning (Wang and Yao, 2019).

2.1.5 Sourcing Image Data From Videos

Sourcing data from videos to be used in image classifiers is a valuable methodology. Although many of the tools used for video labelling are similar to those in image classification, I will briefly discuss related work in this field. The key challenge in handling videos is that in addition to the spatial dimension (single images/frames), videos also contain a temporal dimension (time). Although the temporal information provides an additional challenge, it can also be used to improve the accuracy of labelling entire videos (Karpathy et al., 2014). A key benefit of the temporal dimension is the natural data augmentation that is caused by the jittering of the frames. Researchers have found that even tiny variations in frames can be enough to confuse image classifiers (Zheng et al., 2016) - see Figure 2.5. This suggests that such temporal locality (i.e. similarity of adjacent frames) can be used as valuable training data to improve classifiers.

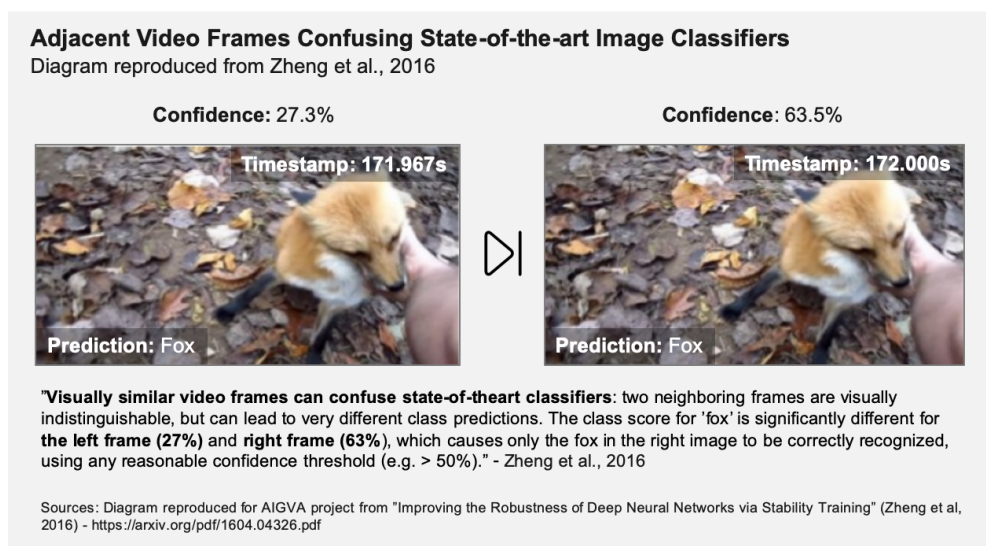


Figure 2.5: Confusing State-of-the-art Classifiers

One area where sourcing training data from videos can be helpful is when dealing with a lack of difficult classes to predict. A common problem faced when training

classifiers is that of class imbalance due to limited training data of certain "hard examples". An oft-cited instances of this problem is in the autonomous vehicle space, where performance of self-driving cars is poor in certain extreme weather conditions, partly caused by a lack of such hard examples in the training data (Zang et al., 2019). There is therefore strong motivation to use videos as a source of such hard examples and recently works have shown that it is possible to "improve detector performance on the source domain by selecting hard examples from videos" (Jin et al., 2018). Closely related to the problem of limited class data is the previously discussed topic of few-shot learning. Early work has also been successfully done to use few-shot learning techniques to successfully classify events in videos with limited training data (Yan et al., 2015).

2.2 Existing Labelling Tools

In this section I will describe and evaluate the existing tools used for annotating video data in machine learning research. The six most popular video annotation tools are: LabelBox, RectLabel, LabelMe, VoTT, VATIC, and CVAT. These tools were examined to understand their strengths and weaknesses. See Figure 2.6 for a breakdown of the analysis. The main takeaway from the analysis is that all tools have their relative advantages for specific domains, but no tool has a combination of native video support, real-time video playback and the ability to auto-generate labels.

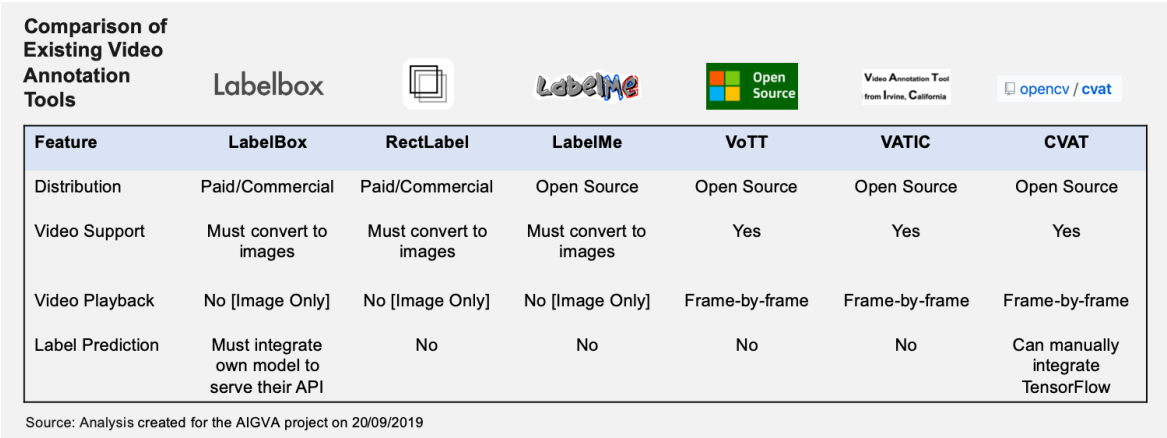


Figure 2.6: Comparison of Existing Video Annotation Tools

LabelBox

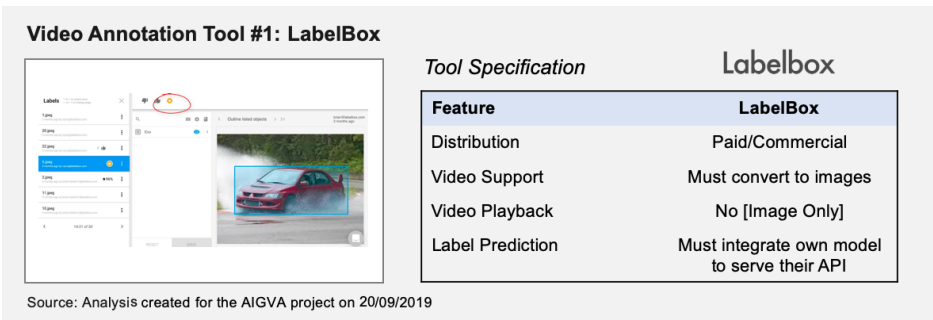


Figure 2.7: Annotation Tool: LabelBox

LabelBox is a commercial software helps customers like Airbus, Bayer and Hitachi label data (Lab, 2019a). The platform can handle various label types including

bounding boxes, polygons, pixelwise and image-level classification. However, there is no support for video data, which must be converted to images and labelled on a frame-by-frame basis. It is possible to use your own custom models to predict labels, but you must create and host your own API service to use this feature.

RectLabel

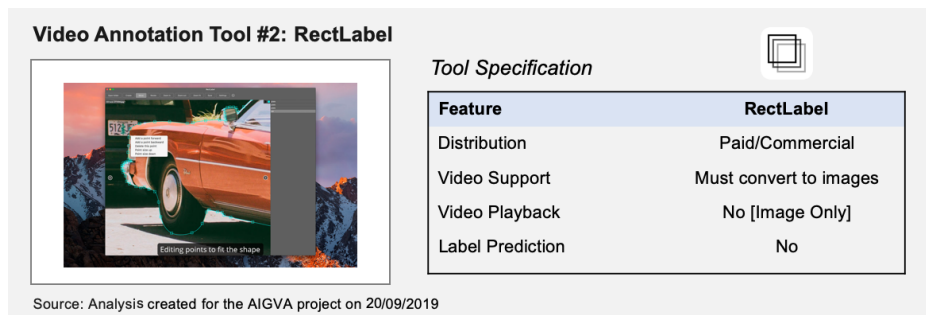


Figure 2.8: Annotation Tool: RectLabel

RectLabel is a commercial image annotation tool that can be used to label images for bounding box object detection and segmentation (Rec, 2019). Similar to LabelBox, it offers no native video support and users must therefore manually convert videos to images if they wish to label them. No label prediction or video playback solutions are offered.

LabelMe

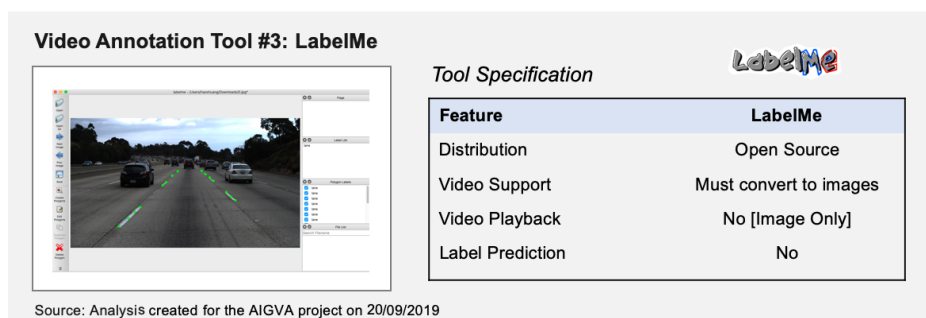


Figure 2.9: Annotation Tool: LabelMe

LabelMe is an open-source computer vision annotation tool created by the MIT Computer Science and Artificial Intelligence Laboratory (Lab, 2019b). The tool supports

instance segmentation, semantic segmentation, bounding box detection and image-level classification. However, like the previous tools it also does not offer video supports and videos must be converted to images to be used.

VoTT

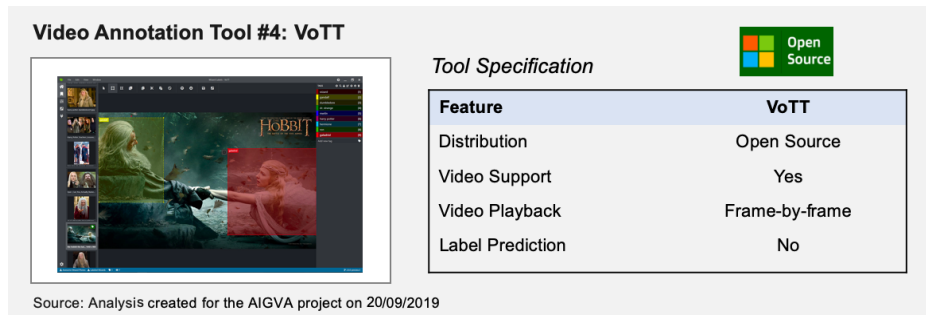


Figure 2.10: Annotation Tool: VoTT

Visual Object Tagging Tool (VoTT) is an open-source app from Microsoft designed for building end-to-end object detection models from images and videos (VoT, 2019). Unlike previous tools, it offers the ability to upload videos without converting them to images. However, to label an entire video a user must label every single frame, which is time-consuming and expensive. There is also no support for label prediction tools either.

VATIC

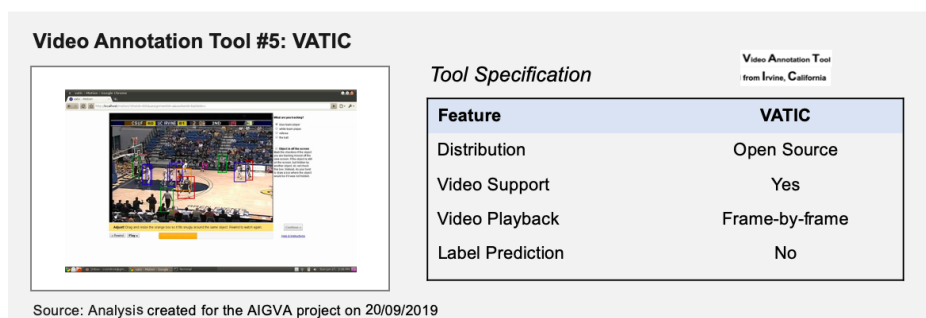


Figure 2.11: Annotation Tool: VATIC

Video Annotation Tool from Irvine, California (VATIC) is an interactive video annotation tool for computer vision research (VAT, 2019). The tool is specialized to

crowd-source labelling tasks to the Amazon Mechanical Turk human intelligence platform. Similar to VoTT, it offers both native video support and frame-level playback. However, labelling an entire video is still slow and the tool offers no label prediction support.

CVAT

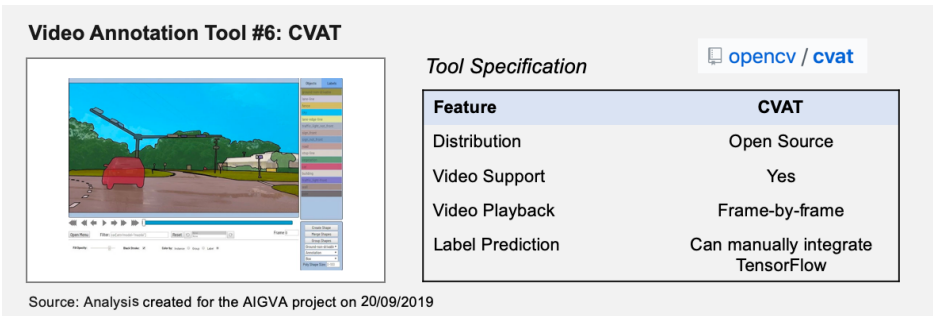


Figure 2.12: Annotation Tool: CVAT

Computer Vision Annotation Tool (CVAT) is an interactive video and image annotation tool for computer vision (CVA, 2019). CVAT is specialized for bounding box and segmentation applications. Unlike previous tools CVAT has the ability to integrate TensorFlow models to run label prediction. However, this feature is not going to be supported in the future and requires users to convert their models to OpenVINO IR format.

2.3 The iFIND Project: Ultrasound Fetal Scan Plane Detection

To motivate the need to source labelled images from unlabelled video data to improve classifiers in a real-world context, I will now discuss the challenges faced by the iFIND ultrasound research team.

The field of image classification in ultrasound imaging presents a series of challenges for researchers. First, the quality of images is limited by the high levels of noise present. Second, there is a limitation on the amount of data available primarily caused by the difficulty of data acquisition driven by privacy issues in the medical field. Third, various image obstructions such as the position of the fetus add a level of idiosyncrasy across datasets. However, researchers have made substantial progress in classifying and segmenting ultrasound fetal scan planes (Baumgartner et al., 2016b; Schlemper et al., 2018b; Baumgartner et al., 2016a; Chen et al., 2015; Toussaint et al., 2018; Cai et al., 2018). As this research project is conducted as part of the iFIND project, we will examine the results of two impactful papers produced by members of the project: Baumgartner et al. (2016b) and Schlemper et al. (2018b).

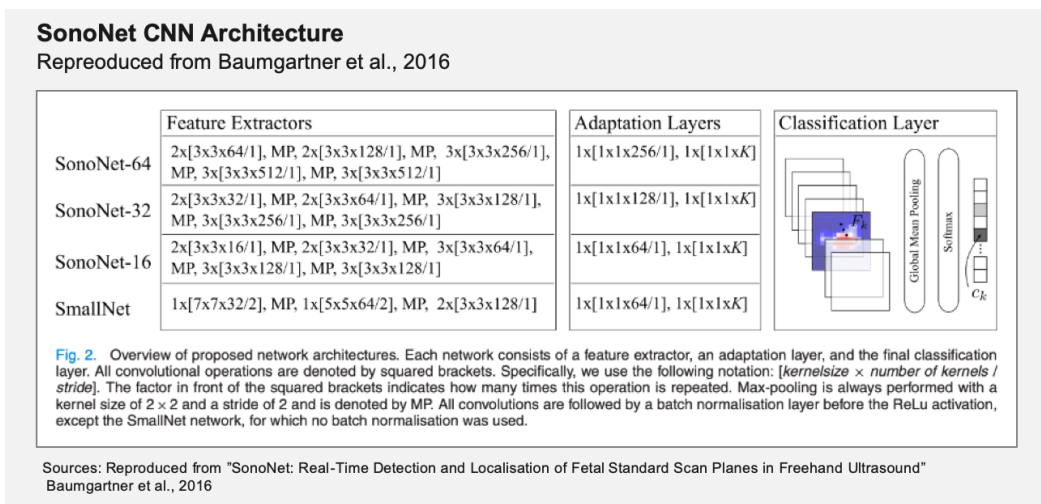


Figure 2.13: SonoNet Architecture Reproduced from (Baumgartner et al., 2016b)

In 2016 Baumgartner et al. presented the SonoNet deep learning model, which was

able to classify 2D fetal ultrasound standard planes. Specifically, they focused on real-time classification of the 13 scan planes recorded by a sonographer. Their network architecture was based on VGG16 (Baumgartner et al., 2016b) and the final convolutional neural network architecture shown in Figure 2.13. The researchers obtained a mean F1-score of 0.798 across the 13 classes. However, the network struggled to classify different cardiac chambers, leading to low recall values for such classes.

In 2018 Schlemper et al. extended the SonoNet architecture by adding attention gates to the network. This addition gave the network the ability to automatically learn to focus on certain substructures within the image. As a result of this addition, researchers were able to improve the results of SonoNet significantly. The introduction of attention gates combined with other modifications increased mean F1-scores to above 0.90. The majority of the improvement came from higher precision scores driven by the reduction of false positives. The researchers suggest that this improvement happens "because the gating mechanism suppresses background noise and forces the network to make the prediction based on class-specific features" (Schlemper et al., 2018b). However, the network still struggled with the most challenging cardiac views. To improve such cardiac view classification, more data is needed. Fortunately, hundreds of hours of video data is available, although it remains to be labelled. This labelling process is currently extremely expensive and time-consuming and therefore the team needs a tool to intelligently label this video data, while wasting as little of the sonographers' time as possible.

2.4 Ethical & Professional Considerations

In this section I will briefly discuss the ethical and professional considerations of this dissertation. Out of the 29 ethical questions as part of the ethics checklist ¹, I only answered yes to one question: "Does your project involve human participants?". I use human participants for the five user interviews conducted. I chose these participants from the BioMedIA research laboratory and through the iFIND ultrasound research project. They all consented to meet and discuss the project in informal meeting sessions. I answered no to the other 28 questions on the checklist.

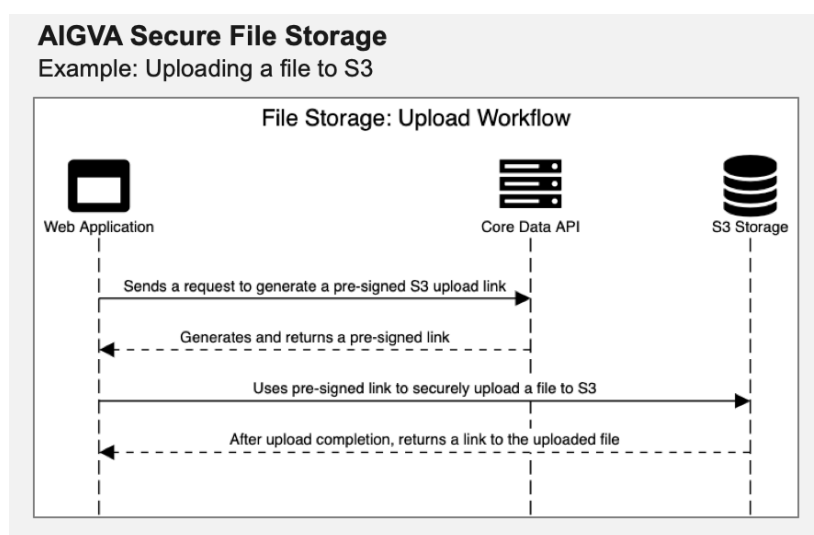


Figure 2.14: AIGVA Securing File Uploads

The other ethical issue relating to my project that I would like to highlight relates to data privacy. I should start by noting that I collect no personal data as part of the project. Here I use the European Union's GDPR definition of personal data that defines it as any data that relates to an "identified or identifiable living individual" (GDP, 2019). However, as part of the tool I have built I do host data on the cloud. To ensure this data stays private I have built an extensive system that verifies uploads by keeping track of user tokens. Specifically, when a user attempts to upload an item to the cloud, the web application requests the Core Data API to generate a pre-

¹Ethics Guideline: <https://www.doc.ic.ac.uk/lab/msc-projects/ProjectsGuide.html#ethics>

signed upload link. Once received by the web application, this link grants temporary write/read access to the client. Now the web application is able to upload a file such as a video directly to S3 without needing to send any data to the web server, as seen in Figure 2.14. This significantly increases security by limiting the data accessibility. This topic is discussed further in the implementation section 4.1.3.

Chapter 3

Method

In this chapter I develop several label prediction techniques that can speed up the labelling process. Then I show a set of potential user stories to highlight how these methods can be integrated into a web application. I then take these user stories and develop a set of design principles that will be used to build the application. Using these design principles I proceed to create the functional specification of the web application including visual mockups of all pages.

3.1 Label Prediction

Labelling is a tiresome and manual process. In this section I develop several ideas that aim to speed up the process of manually labelling videos by automatically predicting labels. These ideas rely on much of the literature discussed in the previous chapter. The three ideas developed are: Cold Label, Blitz Label, and Custom Model Label. The names and ideas require some explanation but a summary of the approaches is shown in Figure 3.1.

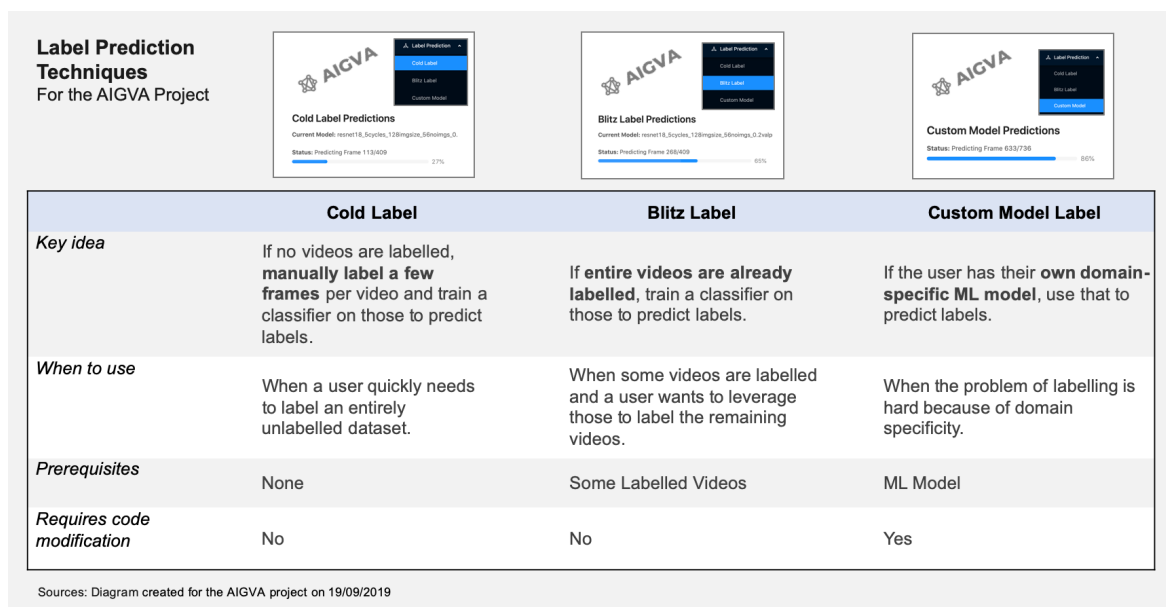


Figure 3.1: Summary of the Label Prediction Approaches

3.1.1 Cold Label

”Cold Label” is a label prediction technique developed to address the cold start problem. The cold start problem is the common issue faced in machine learning when no labels exist. For example, the problem could arise when researchers start a new project in a new domain and receive a large set of unlabelled videos that need labelling. The ”Cold Label” prediction technique attempts to solve this problem by asking a user to label a few samples from a set of videos and subsequently use those labelled samples to train a classifier. This classifier can then be used to predict all frames in entire videos. A summary of the method can be seen in Figure 3.2.

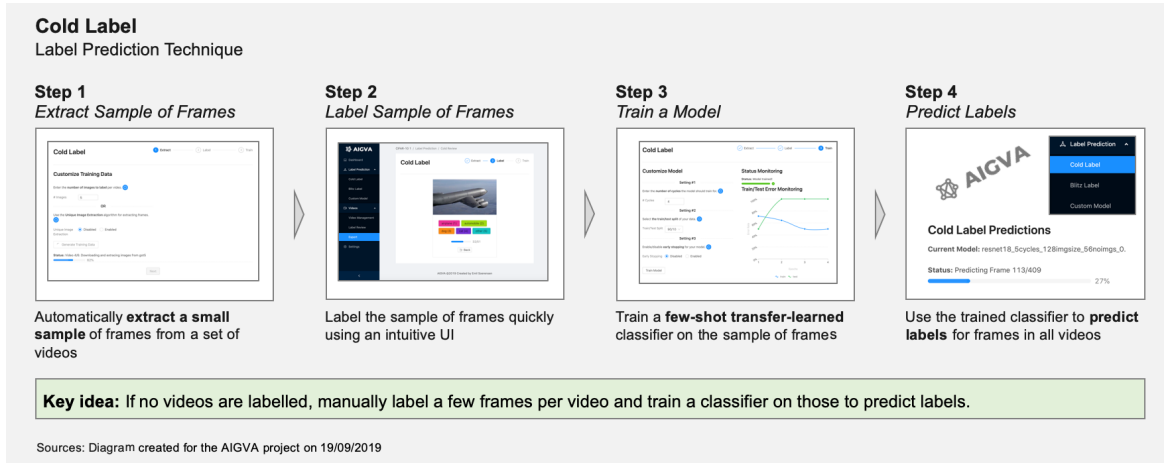


Figure 3.2: Summary of the Cold Label Prediction Technique

The hypothesis underlying the "Cold Label" idea is that it should work for several reasons. First, it should work because videos have inherent similarity within them. Specifically, across the temporal (time) dimension, individual video frames are similar to their neighbor frames. For example, in a sitcom, a character may be speaking for 5 seconds before the camera switches to another character. If the sitcom was playing at 25fps¹, then the character would be on screen for 125 frames. The idea is that there is enough information content in 1-2 of the 125 frames for a model to label them all. And, to continue with the example, should the character reappear on the screen in the same or another video, then a model could also generalize to label these frames.

Second, the "Cold Label" method should work because of advances in transfer learning and few-shot learning. Specifically, as discussed in Section 2.1.4, recent improvements have shown that it may now be possible to train a classifier on a small set of samples and have it generalize well. Third, the "Cold Label" method should work because we take the ideas from human-in-the-loop computing to query an oracle (i.e. a human) to label a subset of frames. By guaranteeing the "correctness" of the labels through human intervention, we can hopefully overcome the cold start problem. We can even incorporate the ideas of active learning by attempting to find the "best" frames to label with the highest information content possible. Although

¹25fps is the PAL (phase alternating line) British television frame-rate standard

traditional active learning deals with querying the model for selecting the optimal sample to label, we can reverse engineer this by attempting to guess what the best sample may be. We explore this idea in the implementation section by developing a "Unique Image Extraction" algorithm (see 4.3.1).

3.1.2 Blitz Label

“Blitz Label” is a label prediction technique that leverages pre-existing labels to predict new labels. The core idea behind the technique is that if we already have labelled some videos, we can use that embedded information to annotate unlabelled videos. For examples, suppose as part of a project a researcher had to label 100 videos. Suppose the researcher had already labelled 50. Suppose he could train a classifier on the 50 labelled videos, and use that classifier to predict labels for the remaining 50 unlabelled videos. That is the intuition driving the development of the “Blitz Label” technique. A summary of the method can be seen in Figure 3.3.

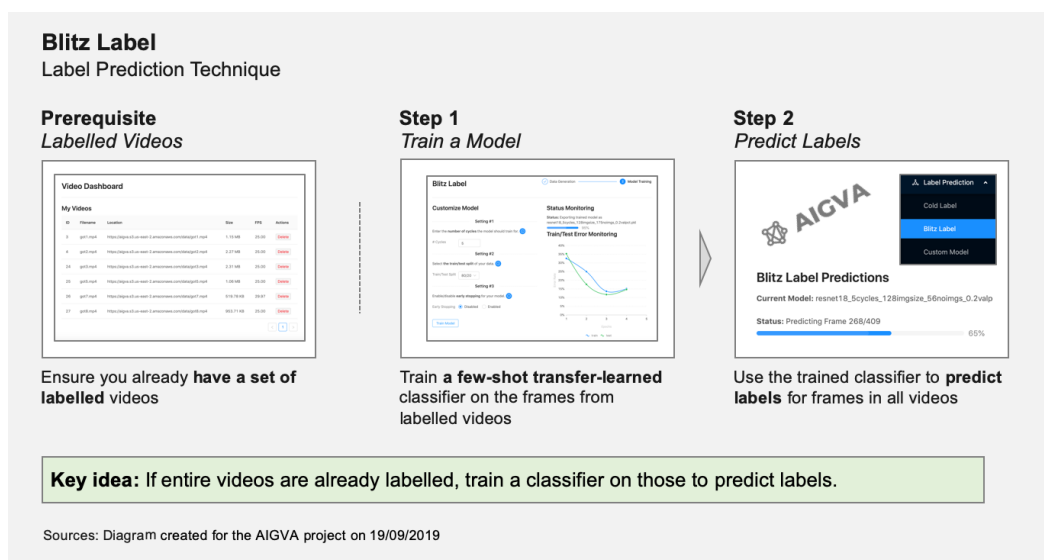


Figure 3.3: Summary of the Blitz Label Prediction Technique

The hypothesis underlying the "Blitz Label" technique is that it should work for reasons similar to why "Cold Label" should work. The core intuition is the same: videos have inherent similarity that could be exploited by utilizing developments in few-shot and transfer learning. The key difference is that no human labelling is

required as part of the "Blitz Label" process, as long as the videos already have some labelling.

3.1.3 Custom Model

The "Custom Model Label" is a label prediction technique that enables machine learning researchers to use their own models as part of the AIGVA platform. The core problem driving the development of technique is that some domains, such as medical image analysis, require very complex classification models. Therefore, giving a user the option to integrate their own machine learning models can tackle this problem. While machine learning researchers could write their own inference tools for label prediction, they would not get the benefits of the AIGVA web application, which also provides a visual tool to examine predictions that can quickly be shared with domain experts. A summary of the method can be seen in Figure 3.4.

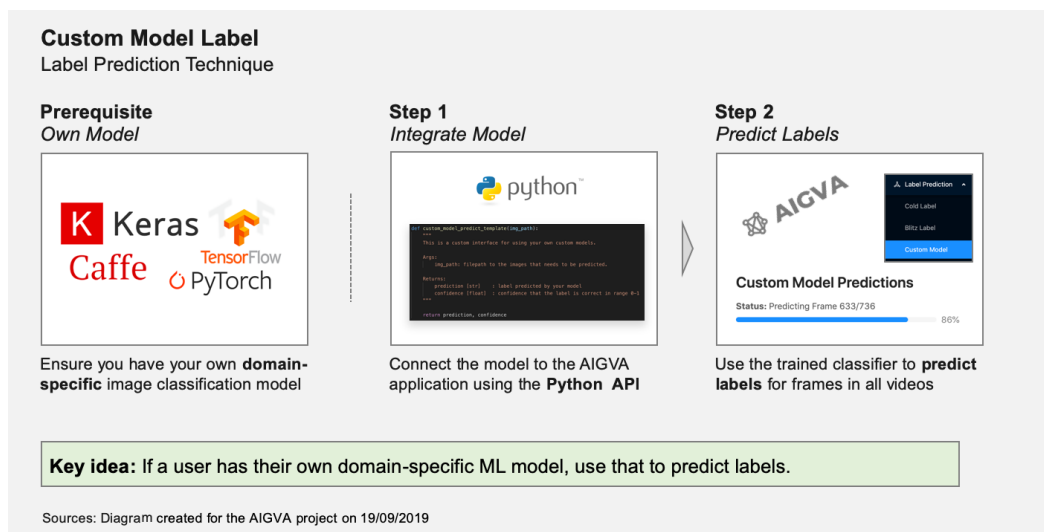


Figure 3.4: Summary of the Custom Model Label Prediction Technique

The hypothesis underlying the "Custom Model Label" idea is that it should provide researchers the ability to use their own models. This will hopefully remove any domain-specific problems that arise when using either one of the Cold or Blitz label methods.

3.2 User Stories

To aid in developing the AIGVA tool, a set of user stories have been created. The idea is to illustrate how different users of varying technical backgrounds can benefit from using the tool.

3.2.1 User Story #1: Building a Pedestrian Classifier for an Autonomous Vehicle Startup

This user story has been written to show how a *non-technical user* can use the tool to significantly speed up the process of labelling videos.

User Story #1.1: Building a Pedestrian Classifier

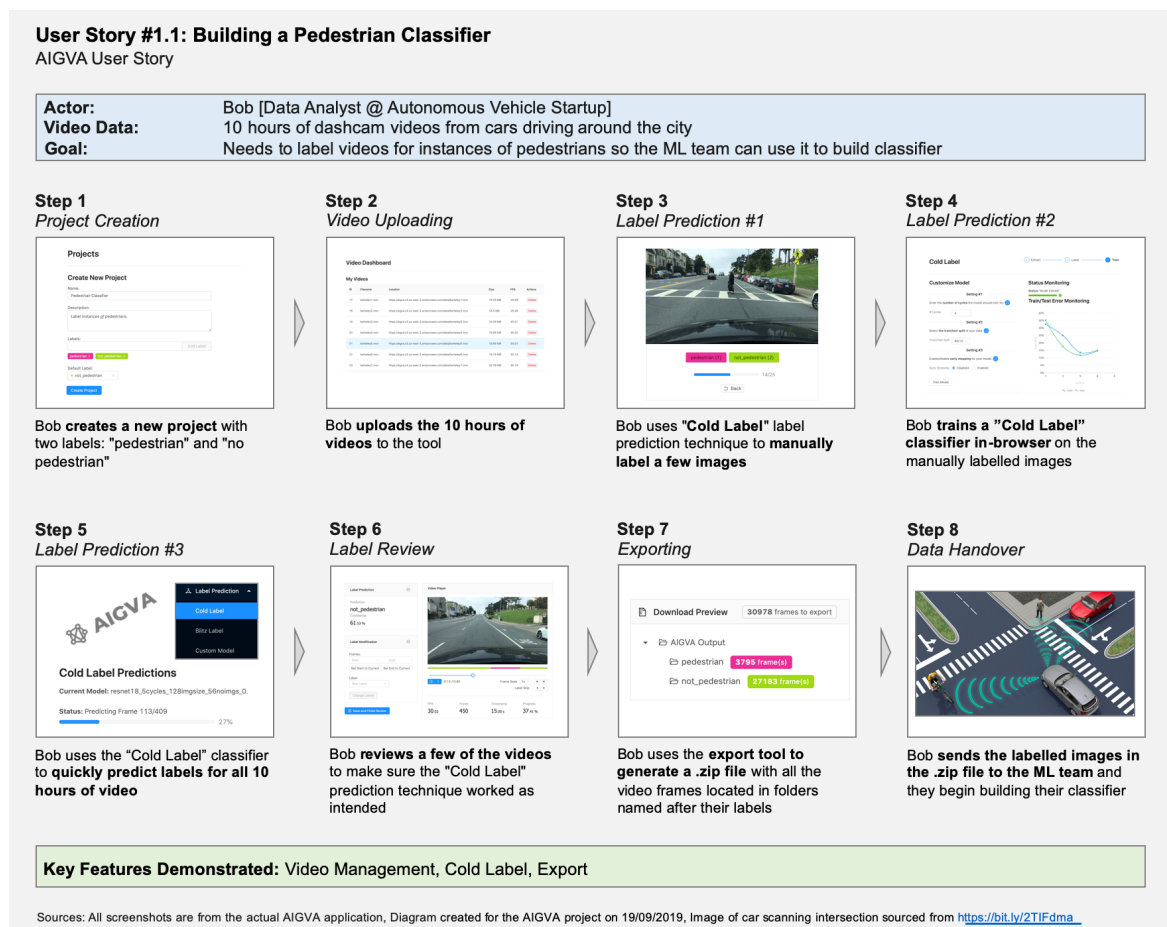


Figure 3.5: User Story 1.1: Building a Pedestrian Classifier

Bob, a data analyst at an autonomous vehicle startup, has been given a labelling task by his boss. The company needs a pedestrian classifier in their driving software and therefore requires some training data so they can build these models. Bob has been given 10 hours of dashcam footage from cars and been asked to label all frames for instances of pedestrians.

1. Bob creates a new project with two labels: "pedestrian" and "no pedestrian"
2. Bob uploads the 10 hours of videos to the tool
3. Bob uses "Cold Label" label prediction technique to manually label a few images
4. Bob trains a Cold Label classifier in-browser on the manually labelled images
5. Bob uses the Cold Label classifier to quickly predict labels for all 10 hours of video
6. Bob reviews a few of the videos to make sure the "Cold Label" prediction technique worked as intended
7. Bob uses the export tool to generate a .zip file with all the video frames located in folders named after their labels
8. Bob sends the labelled images in the .zip file to the ML team and they begin building their classifier

User Story #1.2: Extending the Pedestrian Classifier

After much testing of their new pedestrian classifier, the machine learning team unfortunately find that their model is not accurate enough to be deployed in their driving software yet. To improve the algorithm, they need more training data. Fortunately, their data collection team has acquired 20 additional hours of dashcam video data. Bob is given these 20 hours of unlabelled data and is again asked to label them for instances of pedestrians.

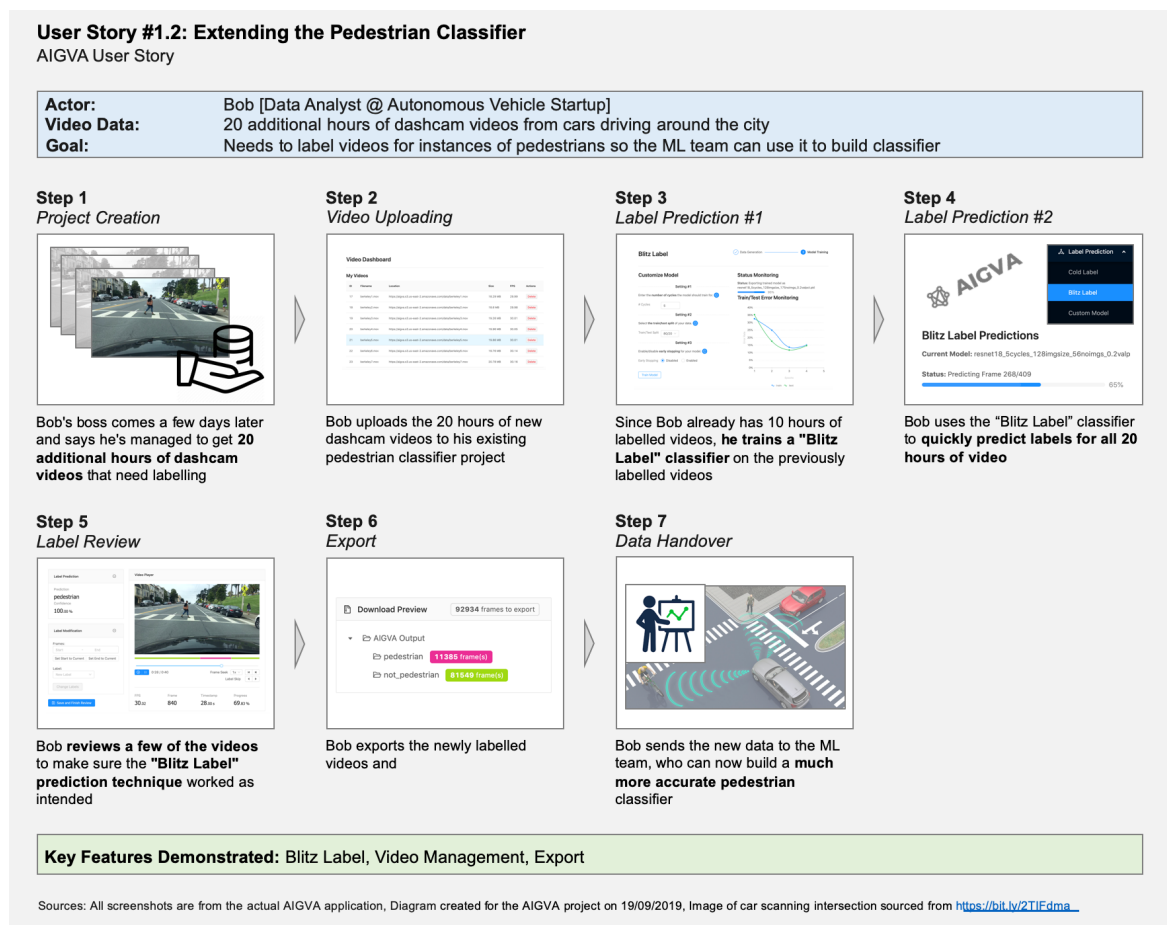


Figure 3.6: User Story 1.2: Extending the Pedestrian Classifier

1. Bob's boss comes a few days later and says he has managed to get 20 additional hours of dashcam videos that need labelling
2. Bob uploads the 20 hours of new dashcam videos to his existing pedestrian classifier project

3. Since Bob already has 10 hours of labelled videos, he trains a "Blitz Label" classifier on the previously labelled videos
4. Bob uses the Blitz Label classifier to quickly predict labels for all 20 hours of video
5. Bob reviews a few of the videos to make sure the "Blitz Label" prediction technique worked as intended
6. Bob exports the newly labelled videos and
7. Bob sends the new data to the ML team, who can now build a much more accurate pedestrian classifier

3.2.2 User Story #2: Improving Standard Plane Detection in Fetal Ultrasound Classifiers

This user story has been written to show how a *technical user* can use the tool to significantly speed up the process of labelling videos.

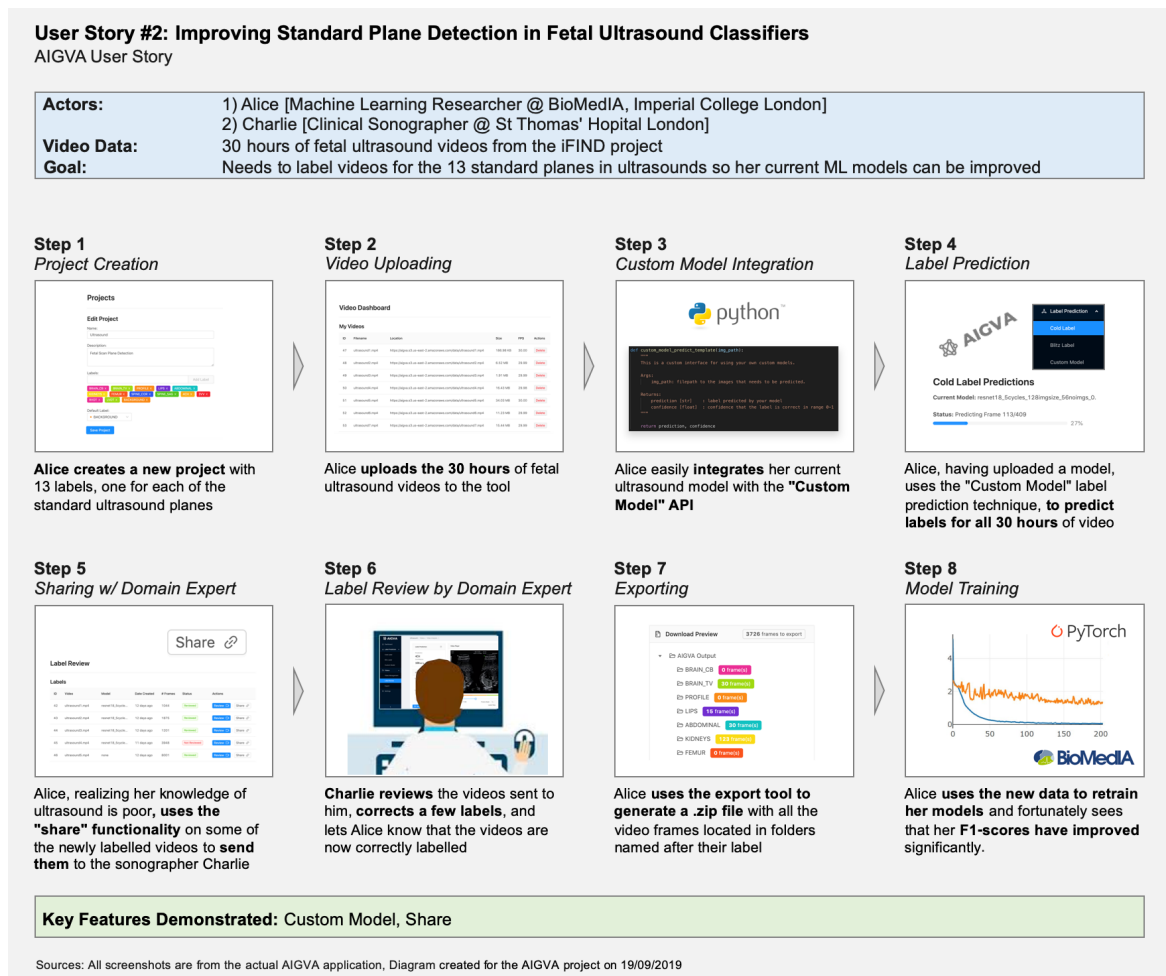


Figure 3.7: User Story 2: Improving Standard Plane Detection in Fetal Ultrasound Classifiers

Alice, a machine learning researcher at Imperial College London's bio-medical image analysis group, BioMedIA, is working on a project to develop an algorithm to detect standard scan planes in fetal ultrasound videos. Over the past few months she has built a classifier that works well on classifying the 13 standard scan plane. However, it is not quite accurate enough to be deployed in the real-world. Fortunately, she has just been give access to 30 hours of new ultrasound videos which, if added to the

training data, she believes can improve the accuracy of the model enough for it to be deployed. Unfortunately, this data is unlabelled and her sonographer colleague Charlie is very busy and does not have much time to label the data for her.

1. Alice creates a new project with 13 labels, one for each of the standard ultrasound planes
2. Alice uploads the 30 hours of fetal ultrasound videos to the tool
3. Alice easily integrates her current ultrasound model with the "Custom Model" API
4. Alice, having uploaded a model, uses the "Custom Model" label prediction technique, to predict labels for all 30 hours of video
5. Alice, realizing her knowledge of ultrasound is poor, uses the "share" functionality on some of the newly labelled videos to send them to the sonographer Charlie
6. Charlie reviews the videos sent to him, corrects a few labels, and lets Alice know that the videos are now correctly labelled
7. Alice uses the export tool to generate a .zip file with all the video frames located in folders named after their label
8. Alice uses the new data to retrain her models and fortunately sees that her F1-scores have improved significantly.

3.3 Design Principles

User interface design is critical when building an application that needs to be used for both technical and non-technical audiences. Galitz has developed an extensive set of guidelines for designing great user interfaces (Galitz, 2007). I will use the relevant principles from Galitz's works on web development to ensure I cover all key design considerations when building the AIGVA tool. Below I will briefly discuss each of these principles and how I plan to include them in the AIGVA tool.

Select the Proper Interaction and Display Devices

I will limit the use of the AIGVA application to work on laptop and desktop devices only. Because of the smaller screens of mobile and tablet devices, designing for both large and small screens "results in the need for different modes" of the application (Benyon, 2014). Creating several modes of the AIGVA tool would cause significant development overhead and other features would have to be sacrificed. Therefore I will limit the supported screen-size to be 1008px², although the application may work on smaller sizes.

I will also design the tool as a web application rather than a desktop application. This is because of two key reasons: (1) web applications do not have to be installed and new features updates can constantly be pushed to them, and (2) web applications are easier to use for a non-technical audience because of the familiar surroundings of a web browser. Because Google Chrome is the most used browser, I will develop the tool to be used with this browser in mind.

Organize and Layout Windows and Pages

The usability of the layout is crucial to develop an easy to use and consistent interface. Since the tool is directed to a both a technical and non-technical audience, the overarching goal of the layout is to be consistent with web application conventions

²Screen sizes and breakpoints: <https://bit.ly/2LkLlft>

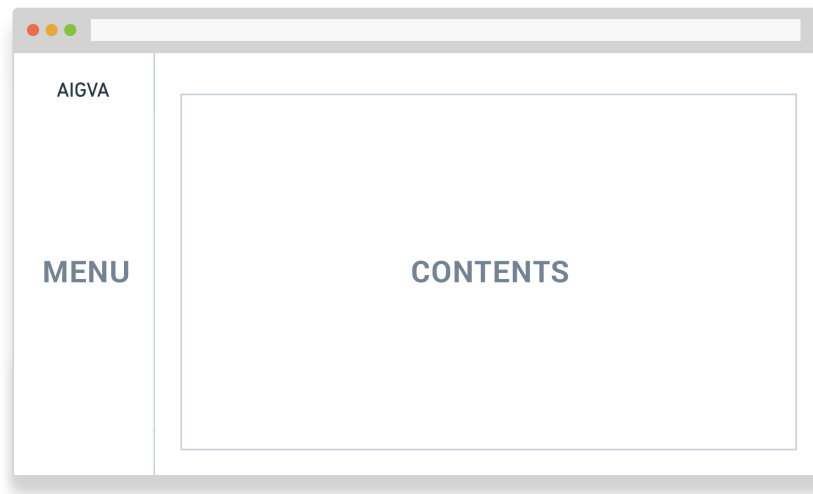


Figure 3.8: AIGVA Mockup: Application Layout

so as not to confuse the audience. For that reason, I have opted for the classic side-bar/content pattern as seen in Figure 3.8. On the left side of the screen, the menu is located for easily accessible navigation. On the remaining part of the screen, the contents is surrounded by a border. This border gives consistency to the user as important information will always be displayed in the same location, without the user needing to scroll. Research has found that "information displayed with a border around it is easier to read, better in appearance, and preferable" (Galitz, 2007).

The AIGVA tool will also use breadcrumbs to display where the user is currently located. Breadcrumbs are page hierarchies that display the trail a user has taken to get to a certain page (i.e. Ultrasound Project > Videos > Export). Using breadcrumbs in web applications have been found to create "more efficient navigation and/or improved user satisfaction" (Galitz, 2007).

Choose the Proper Screen-Based Controls

The controls and components are the building blocks of an application. Instead of building these from scratch I will utilize the Ant Design UI component library (Ant, 2019). This library is one of the most popular and well documented React extensions and comes with hundreds of pre-built components. See Figure 3.9 for examples of

these components such as buttons, input fields, cards, statistics, breadcrumbs and links. Using this component library will help to ensure that the design is consistent and sensible throughout the AIGVA tool.

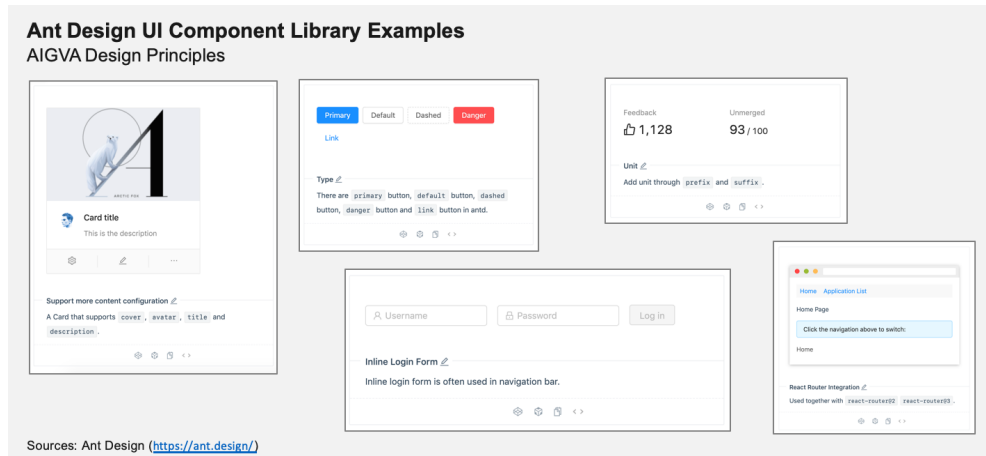


Figure 3.9: AIGVA UI Component Library

Another control consideration is the inclusion of keyboard shortcuts. In a study "keyboard shortcuts have been found to be significantly faster and more accurate than mouse clicks" (Galitz, 2007). Therefore I will include the option to use keyboard shortcuts on pages where significant user interaction is expected.

Provide Effective Feedback and Guidance and Assistance

The human learning loop relies on feedback. When users interact with an object or component and get no response, they become frustrated. To combat this, I will use Galitz's hierarchy of feedback to guide design choices (Galitz, 2007):

- **If the wait is less than a second:** provide an immediate visual cue when the operation is completed within 100ms (i.e. button click changes color briefly)
- **If the wait is more than a second but less than 10 seconds:** provide an animated "busy icon" until the operation is complete
- **If the wait is greater than 10 seconds:** provide a time estimate in the form of a status or progress bar

Create Meaningful Graphics, Icons, and Images

Where applicable, I will use visual cues in the form of icons and graphs to display information. Research has found that the use of such graphics can help "hold the users attention, add interest to a screen, [and] support computer interaction" (Galitz, 2007).

Choose the Proper Colors

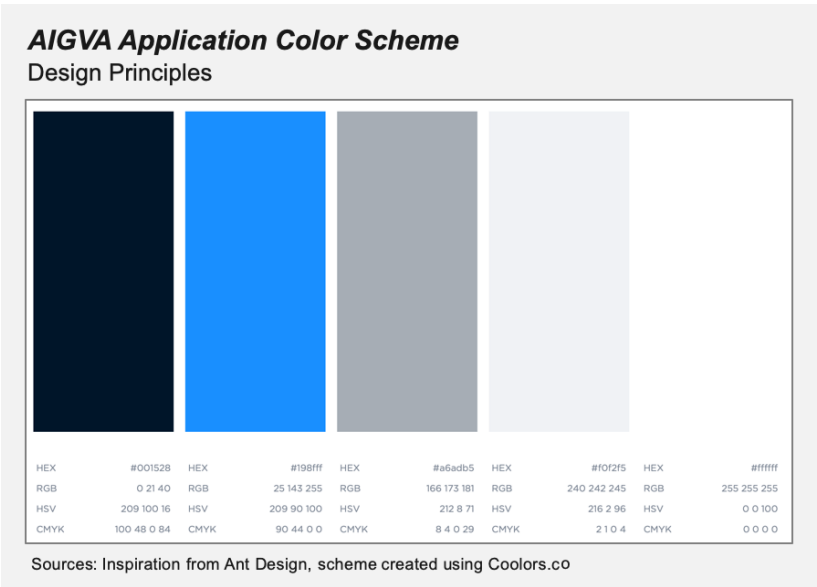


Figure 3.10: AIGVA Color Scheme

Color is a crucial element in designing strong visual hierarchies. A study on color in applications found that it "has a stronger perceptual influence than proximity or grouping" (Galitz, 2007). To make the application accesible, I designed a simple blue-gray-white color scheme (see Figure 3.10). Using the *Coolors.co* tool I ensured that the color scheme was clearly visible to users with the most common forms on color blindness: protanomaly, deuteranomaly, and tritanomaly.

3.4 Requirements

In this section, I will describe the functional specification and requirements of the various components that make up the AIGVA tool. The purpose of this section is to guide development and inform the reader as to the scope of the project. I hope to aid the reader in understanding the specifications by providing visual mockups of key application pages. These mockups were also used to guide the development of the web application.

The tool consists of three functional groups: project management, video management, and label prediction.

3.4.1 Project Management

Project Dashboard

A user is presented with a dashboard as the homepage containing the following:

- A section showing the current project with information about the number of videos and labels associated with that project
- A graph showing the distribution of labels of the current project
- A table with information about all the projects created and the ability to switch to another project
- The ability to create a new project by clicking a button
- The ability to edit an existing project by clicking a button

Figure 3.11 shows a mockup of the project dashboard.

Project Add/Edit Page

A user is able to create and modify projects on the add/edit project page. A project must have the following attributes:

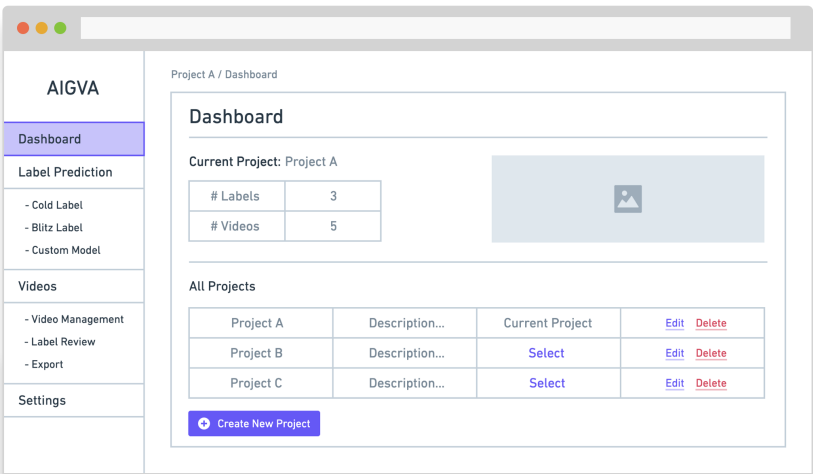


Figure 3.11: AIGVA Mockup: Project Dashboard Page

- A name defining the project
- A short description
- A set of labels that are non-overlapping
- A default label to be used when new unlabelled videos are uploaded

Figure 3.12 shows a mockup of the project add/edit page.

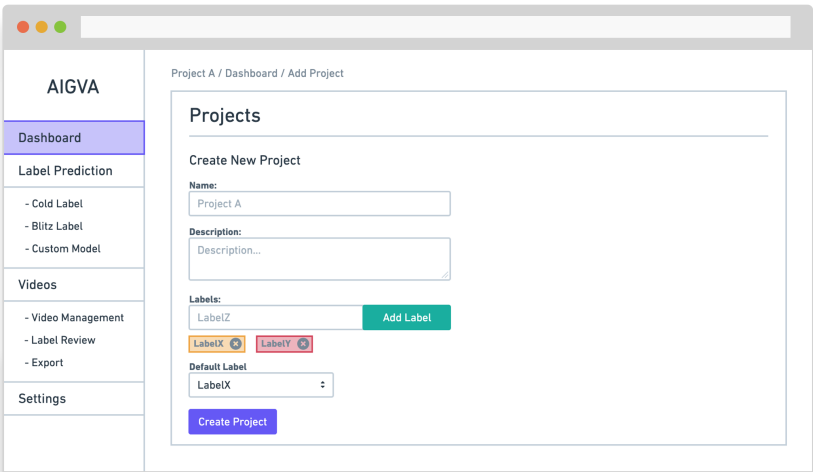


Figure 3.12: AIGVA Mockup: Project Add/Edit Page

3.4.2 Videos

Video Management Page

The AIGVA tool has a video management page with the following functionality:

- A table showing all the uploaded videos, and their associated metadata (i.e. frames per second, file location etc.). The table should be paginated, with maximum 10 videos per page
- A user can upload a video using a simple in-browser upload tool
- A user can delete a video by clicking a button which also deletes labels associated with the video
- A video must be one of the following supported formats: .mp4, .mpg and .webm

Figure 3.13 shows a mockup of the video management page.

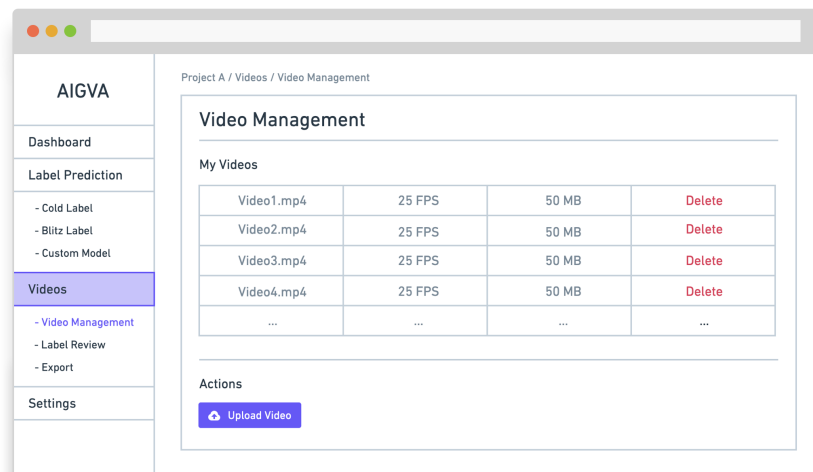


Figure 3.13: AIGVA Mockup: Video Management Page

Label Review Page

The label review page must contain a table with the following information and actions:

- Metadata about each label-set including: the video associated with the label-set, the number of frames in a label-set, reviewed status (true/false), the date the label-set was created, the model (if any) used to generate the labels
- A "review" button that takes a user to the in-browser video player
- A "share" button that generates a shareable link to the in-browser video player for that label-set. When clicked, the tool should automatically copy the link to the users clipboard

Figure 3.14 shows a mockup of the label review page.

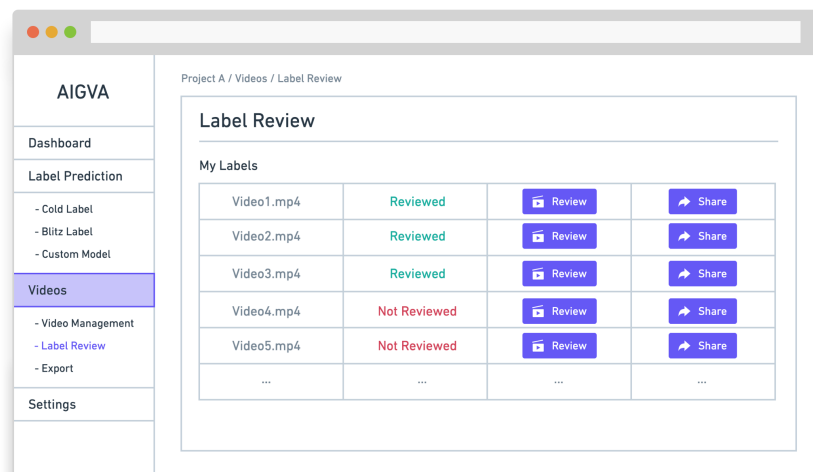


Figure 3.14: AIGVA Mockup: Label Review Page

Video Analysis Page

The video analysis page must have an in-browser video player, a label prediction section, and a label modification tool:

- A video player section with navigation and information presented to the user
 - Video navigation should be possible in several ways:
 - * A play and a pause button triggering the video to play/pause
 - * Using a slider that can be used to scan through the movie (like YouTube)

- * Frame-by-frame skipping using either buttons or arrow keys with several options available for the number of frames to skip (skip 1 frame, $x/4$ frames, $x/2$ frames, x frames, where x is FPS for that particular video)
- * Skip-to-next-label navigation using either buttons or arrow keys that automatically scans the set of labels for the "next different" frame
- The following video summary statistics should be presented and updated in real-time: FPS, current frame, timestamp, and progress (as %)
- A label preview bar under the video player that matches the length of the slider should indicate to users the distribution of labels across the video
- Audio should be enabled
- A label prediction section that shows two statistics in real-time:
 - The label prediction for the current frame
 - The confidence of the label prediction for the current frame
- A label modification tool that enables a user to change labels:
 - A user should be able to modify labels by inputting both a start and an end frame (which can be the same), then selecting a label from the list of available labels for that project, and ultimately clicking a "Change Labels" button
 - Changing labels will refresh the page with the updated set of labels and set their confidence to 100% since they were labelled by the user

Figure 3.15 shows a mockup of the video analysis page.

Label Export

The AIGVA application should have a page that makes it possible to export labels in two ways: (1) downloading the labels only or (2) downloading the labels and the corresponding images:

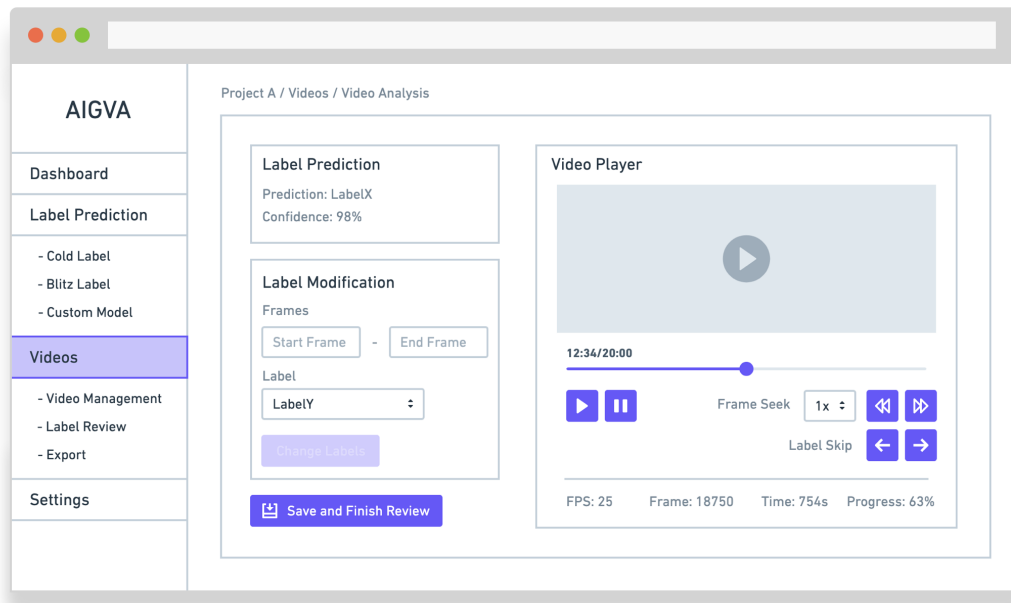


Figure 3.15: AIGVA Mockup: Video Analysis Page

- The first way of exporting labels is a "JSON only" export that generates a JSON file containing frame-by-frame labels for all videos in a project
- The second way of exporting labels is a "full download" that generates a .zip file containing both images and their labels
 - The output of a "full download" must be a zipped file with a folder for every label containing individual video frames (in the .jpg format) belonging to that label. This is so they can directly be fed into an ML pipeline
 - The user must be able to configure a "full download" by selecting the number of frames to skip, if any (i.e. skipping every 8th frame will reduce the total number of images by a factor of 8)
 - The application should show a preview folder structure of a "full download" by estimating the number of labels per class depending on the number of frames to skip, if any

Figure 3.16 shows a mockup of the export page.

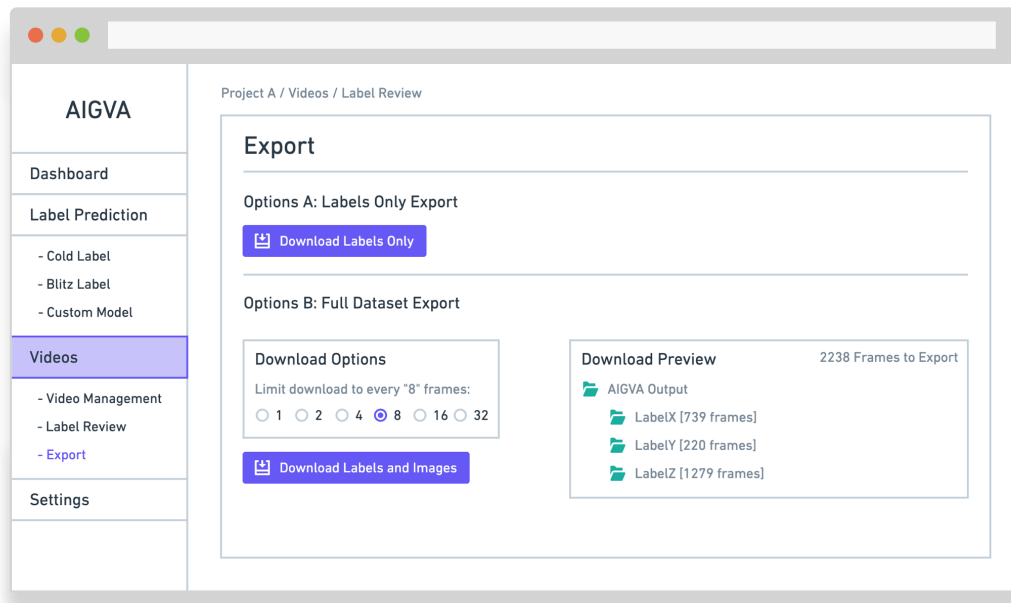


Figure 3.16: AIGVA Mockup: Video Analysis Page

Miscellaneous

- A user must always see their relative location in the application by viewing a "breadcrumb" at the top of every page (i.e. Project Name / Label Prediction / Blitz Label)
- The AIGVA tool should have a collapsible sidebar for navigation
- A settings page should exist where a user can customize various settings for the AIGVA application
- If a user tries to access an invalid subdomain on the AIGVA page, the user must be redirected to a blank 404 error page

3.4.3 Label Prediction

Real-Time Prediction Pages

A user must be able to predict labels using three methods: cold labelling (no labelled videos necessary), blitz labelling (some labelled videos necessary) and custom model

labelling (own custom machine learning model required).

- A user must be able to select one of the three types of machine learning models and then use it to predict labels for a video
- While a model is predicting labels for a video, the user must be kept informed as to the progress of the predictions

Figure 3.17 shows a mockup of the prediction page (which is near-identical for Cold, Blitz and Custom Model Label).

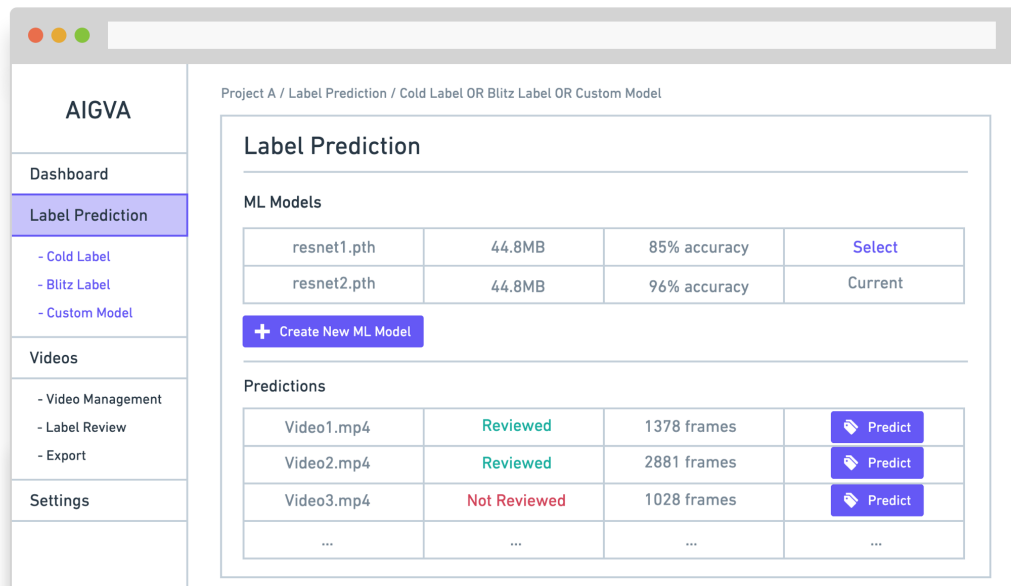


Figure 3.17: AIGVA Mockup: Label Prediction Page (Cold, Blitz and Custom Model Label)

Cold Labelling

”Cold Labelling” is when a user predicts labels by training a machine learning model on a few manually labelled frames per video. This works in three steps:

- First, a user should specify the number of frames they are willing to label per video and the AIGVA tool should then automatically extract that number of frames from each video uploaded

- Second, when a list of images have been extracted from all videos, a user should be able to manually label the frames using a simple graphical user interface (GUI) with the following features:
 - For every image to label, the GUI tool should include buttons for all labels associated with that project and keyboard shortcuts to quickly label an image
 - The GUI tool should provide a progress bar to let the user know how many labels that are left to manually label
 - The GUI tool should have a "back" button in case an item was wrongly labelled
- Third, after a user has manually labelled the training data, the user should be able to train an image classification machine learning model on the data and monitor its progress

Figures 3.18, 3.19, and 3.20 show mockups of the three Cold Label steps.

Blitz Labelling

"Blitz Labelling" is when a user predicts labels by training a machine learning model on previously reviewed labels in their project. It works in two steps:

- First, a user should be able to extract training data from reviewed videos:
 - A user must have some reviewed videos that have been previously labelled either manually, by a "Cold Label" model, by another "Blitz Label" model, or by a "Custom Model". These videos must have been reviewed by a human to verify the labels as "ground truth"
 - A user should specify the number of frames they want to skip per video and the AIGVA tool should then automatically extract that number of frames from each video marked as "reviewed"

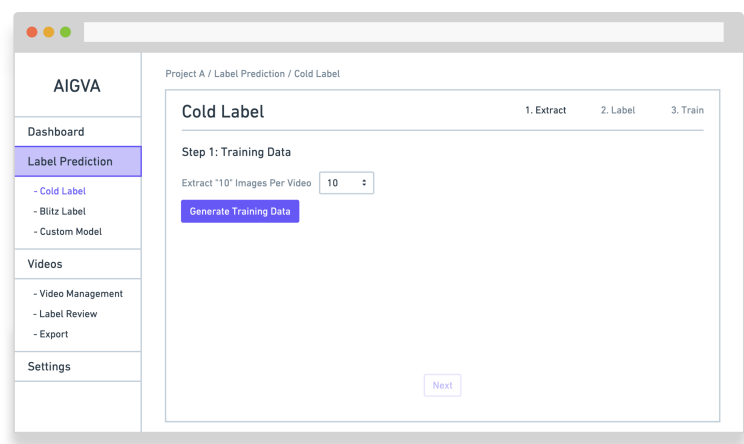


Figure 3.18: AIGVA Mockup: Cold Label (Step 1: Extract)

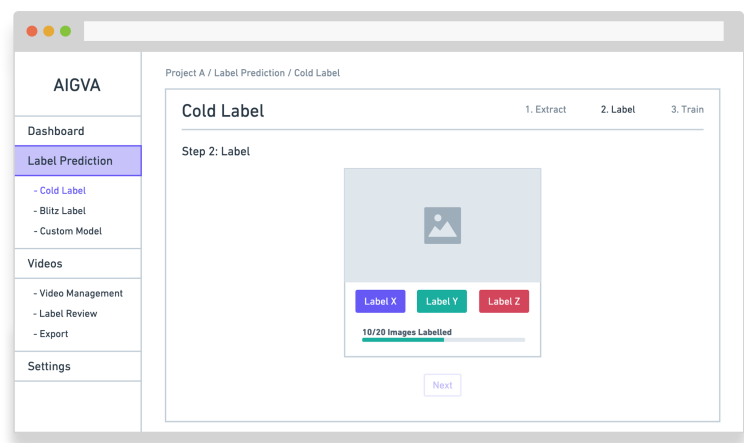


Figure 3.19: AIGVA Mockup: Cold Label (Step 2: Label)

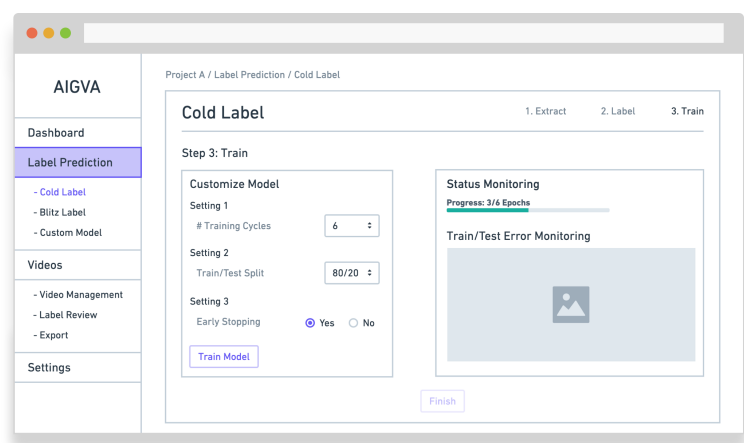


Figure 3.20: AIGVA Mockup: Cold Label (Step 3: Train)

- Second, after a user has extracted the training data, the user should be able to train an image classification machine learning model on the data and monitor its progress

Figures 3.21 and 3.22 show mockups of the two Blitz Label steps.

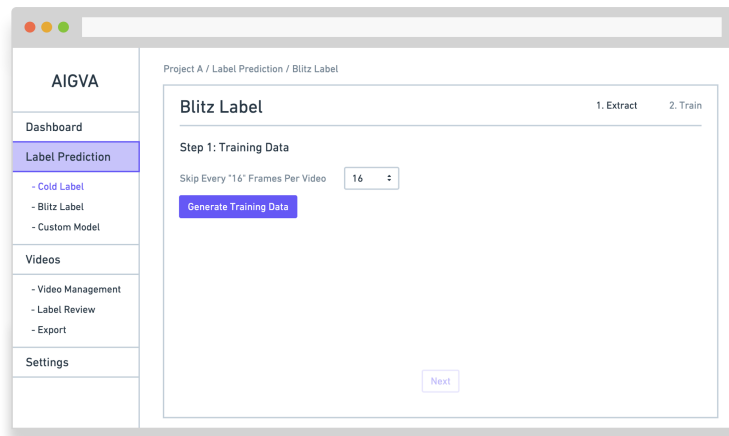


Figure 3.21: AIGVA Mockup: Blitz Label (Step 1: Extract)

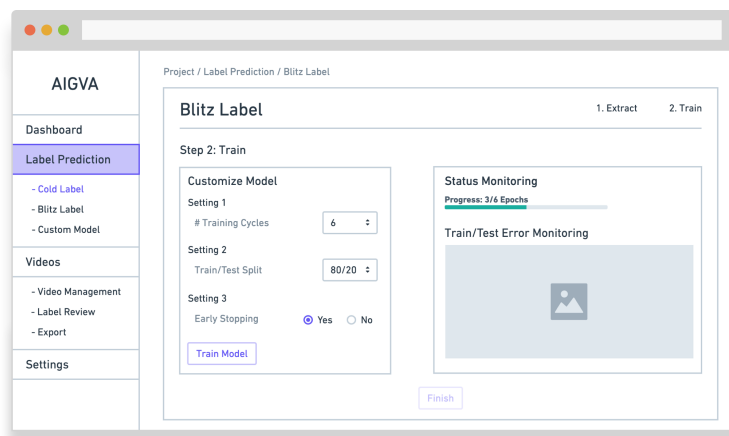


Figure 3.22: AIGVA Mockup: Blitz Label (Step 2: Train)

Custom Model Labelling

”Custom Model Labelling” is when a user predicts labels by using their own customized machine learning model and writing a small amount of code to wrap their model into the AIGVA tool

- If a user already has a machine learning model, they should be able to integrate it into the AIGVA tool by implementing a simple Python interface that can be used to predict frames

Chapter 4

Implementation

4.1 Software Architecture & Technology Stack

In this section of the report, I will discuss the technologies used to accomplish building the application. I will describe the overall architecture of the tool and how individual components interact. Modern web development is complicated for many reasons. There are an overwhelming amount of tools available that it becomes hard to know when to use which tool. Furthermore, the interplay between frameworks is opaque and often breaks, and tools themselves keep evolving from week to week. In this section I will argue why certain frameworks and tools were chosen over others in the context of building the AIGVA tool. I will try to illustrate these benefits by providing small code sections and diagrams where applicable.

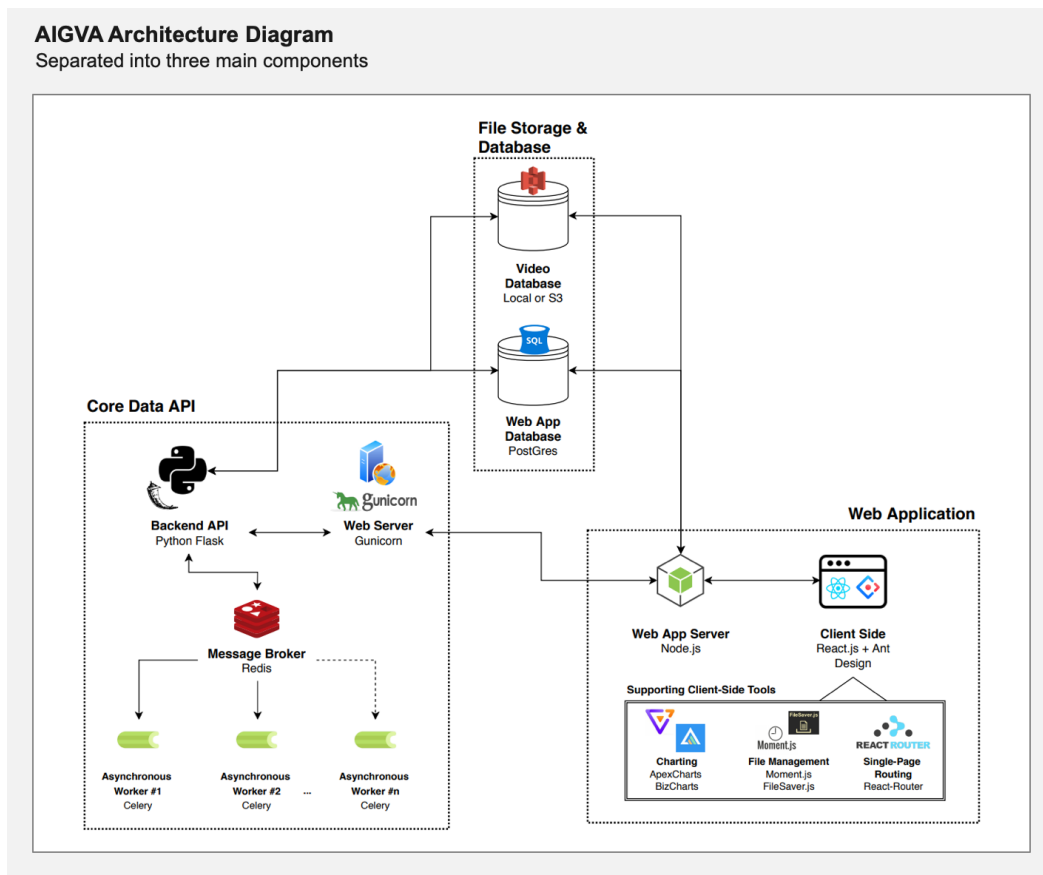


Figure 4.1: AIGVA Architecture Diagram

The entire AIGVA application is divided into three separate components: (1) the Core Data API responsible for video processing, machine learning and other data related tasks, (2) the Web Application responsible for providing the UI and browser tool that a user can interact with, and (3) the File Storage and Database layer responsible for hosting all video data and application related data.

4.1.1 Core Data API

The Core Data API is the "heavy lifter" of the application. It provides various API endpoints for the web application to use when any significant data processing is needed. This includes: video processing, image extraction, machine learning training and inference, export functionality and other features.

Backend Web Framework

A web framework is a piece of software designed to build APIs and web applications. For the Core Data API, choosing the appropriate framework is the most important architectural decision as it defines the speed of development. For that reason, I have defined several key requirements a framework must have.

Key requirements:

- Speed:
 - Rationale: The application will have to handle many requests/second when providing progress updates to data heavy tasks or serving many users simultaneously
 - Ranking metric: Avg. latency (lower is better)
- Data throughput:
 - Rationale: Certain application-specific data packets such as videos and labels are large and thus a framework needs to accommodate high data throughput
 - Ranking metric: Data throughput per second (higher is better)
- Machine learning support:
 - Rationale: Most modern ML is done using Python based frameworks, so having a Python based web-framework makes synchronization easier. The SonoNet ultrasound model is also written in PyTorch, a Python based ML framework.
 - Ranking metric: Python-based
- Well documented and widely used:
 - Rationale: A more well documented framework is easier to work with and will have a larger user-base, which makes resolving development problems faster

- Ranking metric: # GitHub stars (higher is better)

Having formulated the requirements, I analyzed four of the most popular web frameworks: Flask, Django, Ruby on Rails and Express.js. The table below shows the comparison.

Web Framework Comparison For the AIGVA Project				
	Winner Flask	django	RAILS	Express.js
Requirement	Flask	Django	Ruby on Rails	Express.js
Speed (avg. latency) ¹	51.37 ms	145.44 ms	16.20 ms	48.06 ms
Data Throughput (per sec) ¹	48.31 MB	22.91 MB	12.08 MB	102.49 MB
ML Support (Python Based)	Yes	Yes	No	No
Github Stars ²	45.8K	43.2K	43.7K	44.9K

Sources: ¹Benchmark <https://github.com/the-benchmark/web-frameworks>, ²<https://github.com/> Analysis created on 7/09/2019

Figure 4.2: AIGVA Web Framework Comparison

I selected Flask as the web framework for the project since it performed well on all four criteria. Only Flask and Django were Python based, which ruled out Ruby on Rails and Express.js. In terms of speed and data throughput Flask outperformed Django on both parameters. Furthermore, Flask had more GitHub starts, which was used as a proxy for how well documented the frameworks were. The authors of Flask describe that it the tool "designed to make getting started quick and easy, with the ability to scale up to complex applications" (Git, 2019). Those design principles make it an ideal tool to use for developing the AIGVA tool.

API Framework

The Core Data API is designed to communicate with the web application in the most efficient way possible. The most common way of designing such communication patterns is using the Representational State Transfer (REST) model since they "emphasizes scalability of component interactions" (Fielding, 2000). API's adhering to such a pattern are often termed RESTful API's. While new tools such as GraphQL promise greater scalability, I have chosen to design the API using REST because of the breadth of its documentation and ease of integration with Flask.

To implement the RESTful API I have opted to use the Flask-Restless framework since "it provides simple generation of ReSTful APIs for database models defined using SQLAlchemy." (Fla, 2019). That is, Flask-Restless creates elegant API's using the object-relational-mapping tool SQLAlchemy that I've chosen to define my models using. To demonstrate the simplicity of exposing such an API, I will show an example of how the route is created for Projects below.

Listing 4.1: API Source Code (Located in: backend/app/app.py)

```
1 class Project(db.Model):
2     id          = db.Column(db.Integer, primary_key=True)
3     name        = db.Column(db.Unicode)
4     desc        = db.Column(db.Unicode)
5     main_label   = db.Column(db.Unicode)
6     label_types = db.Column(db.JSON)
7     date         = db.Column(db.DateTime)
8     videos       = db.relationship("Video", backref=db.backref('
project'))
9     models       = db.relationship("Model", backref=db.backref('
project'))
10    labels        = db.relationship("Label", backref=db.backref('
project'))
11
12 manager = flask_restless.APIManager(app, flask_sqlalchemy_db=db)
13 manager.create_api(Project, methods=['GET', 'POST', 'PUT', 'DELETE'])
```

In the code example above, there are two section to note. First, the project model, which is written using the object-relational-mapper SQLAlchemy. This defines the schema of a project, which has different attributes such as a name, description, label types etc. Second, the entire RESTful API which is written in two lines of code. The first line uses the APIManager to initialize the API endpoints by binding the Flask application to the database. The second line creates an endpoint for the project model by specifying which HTTP methods are allowed. When the Flask app is running the API is live and it is possible to send requests to the /api/project slug. For example, to get a list of all projects the client simply needs to send a GET requests to CoreDataAPI:host/api/project/ and the server will return a JSON

object containing all the projects.

Web Server

Although Flask is shipped with its own development server, it is not suitable for production. Therefore a webserver is highly recommended. A webserver sits between Flask and the outside world. I selected Gunicorn since it is a popular choice for Flask and Docker. Specifically, Gunicorn is a "Python WSGI HTTP Server for UNIX" (Gun, 2019) and it works by having multiple worker processes handle requests instead of a single-threaded worker, which Flask's development server uses.

Worker-Broker Multi-Processing

One of the key challenges faced when building the AIGVA application was how to handle concurrent data processing requests. Consider the complex task of predicting labels for a video. Completion of this task may take minutes as it depends on: the efficiency of the data pre-processing pipeline, the complexity of the machine learning model, and the length of the video. Now consider a user who wants to predict labels for 10 videos. The naive solution is to have the user select to predict labels for one video, wait for the server to finish prediction, then select to predict labels for the second video, until they have completed all 10. This is time consuming and inefficient. Instead, it would be much more efficient to have the server handle 10 tasks concurrently.

To create a server architecture that can handle concurrent jobs, one needs to use a message broker and task queue. The task queue is responsible for receiving tasks from the Flask application and processing them asynchronously, leaving the Flask application to handle other requests. When a job is sent to the task queue client, the client issues the task to a worker, which functions as a new process. This enables pseudo-concurrency. Critical to this implementation is a message broker, which enables the task queue client to talk to its workers and vice versa. This is an entirely separate server.

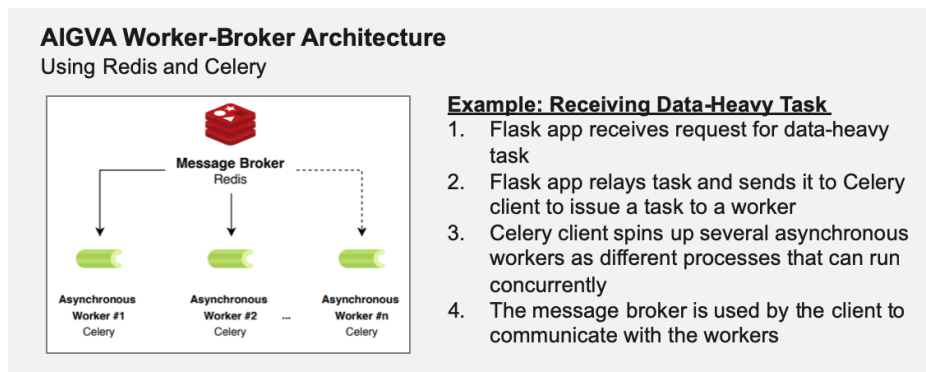


Figure 4.3: AIGVA Redis-Celery Architecture

To build such a system, I opted to use Celery as a task queue and Redis as the message broker. This is a common stack, because of its efficiency and compatibility with Flask. Celery is "a simple, flexible, and reliable distributed system to process vast amounts of messages, while providing operations with the tools required to maintain such a system." (Cel, 2019) I also selected Celery because of its ability to provide status updates to the frontend client-side, which provides a more enriched user-experience as it keeps the end-user informed as to the status of their tasks. Redis is "an in-memory data structure store, used as a database, cache and message broker." (Red, 2019). Minimal configuration is required for it to work efficiently with Celery and Flask, which was the primary reason why it was chosen.

Video Processing & Image Extraction

A key requirement of the Core Data API is the ability to process videos to perform various tasks such as video downloading, metadata acquisition, and seeking for image extraction. Several Python libraries offer such capabilities including Sci-kit Video, MoviePy, imageIO and OpenCV. Since the video processing functionality is another critical component of the application, I conducted a thorough analysis based on the following requirements.

Key requirements:

- Frames per second extraction:
 - Rationale: A method to quickly infer the frames-per-second of a video is

core to the application as it is required when uploading a new video to the database

- Ranking metric: Availability
- Frame-level image extraction:
 - Rationale: The ability to efficiently iterate over frames in a video is key in several core application features such as predicting labels and exporting images
 - Ranking metric: Availability
- Well documented and widely used:
 - Rationale: A more well documented framework is easier to work with and will have a larger user-base, which makes resolving development problems faster
 - Ranking metric: # GitHub stars (higher is better)

Python Video Processing Libraries For the AIGVA Project

	MoviePy	OpenCV	imageIO	scikit-video
Requirement				
FPS Extraction	Yes	Yes	No	Yes
Frame-Level Image Extraction	No	Yes	Yes	Yes
Github Stars ¹	5.8K	37.1K	0.6K	0.3K

Sources: <https://github.com/>, Analysis created on 7/09/2019

Figure 4.4: AIGVA Video Processing Comparison

Out of the four libraries examined, OpenCV was the clear winner and I have therefore chosen to use it (Ope, 2019). It was one of only two libraries to have both FPS and frame-level image extraction. The other library, scikit-video, only has 0.3K GitHub stars compared to OpenCV's 37.1K stars. Furthermore, the ecosystem for deploying OpenCV on Docker is more developed, which makes deployment more straightforward. One of the strengths of the OpenCV codebase is its simplicity to extract metadata. For example, when uploading a new video to the database it is

important to know both the FPS and frame counts. This is important to do server-side as doing so on the client-side would be too computationally expensive. With OpenCV, it is possible to infer these two statistics in just four lines as seen in the code snippet from the backend route responsible for computing such statistics.

Listing 4.2: Video Metadata Extraction (Located in: backend/app/routes_videos.py)

```
1 cap          = cv2.VideoCapture(file_location)
2 fps          = cap.get(cv2.CAP_PROP_FPS)
3 frame_count  = int(cap.get(cv2.CAP_PROP_FRAME_COUNT))
4 cap.release()
```

4.1.2 Web Application

The web application is the "view" of the AIGVA tool. It functions as the only layer the user interacts with. Therefore usability, accessibility and performance are critical features. The web application is structured around a single-page web application and supported by numerous frontend libraries.

Web App Server & Client Side

Choosing the frameworks of the web application server and client side is the most important aspect of the frontend architecture as all other choices depend upon this. The most popular framework choices for building a scalable frontend are: React, Vue, Angular, and using plain JavaScript/HTML without any additional frontend libraries. A good framework must meet several criteria, the most important of which are listed below.




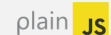
Key requirements:

- Modernity:
 - Rationale: Using a popular and modern tech-stack means more tutorials and learning materials is going to be available
 - Ranking metric: Downloads in last 6 months (higher is better)

- Performance:
 - Rationale: Performance of the framework is largely determined by load times of the client-side, which is a direct result of the size of the framework
 - Ranking metric: Framework size (lower is better)
- Well documented and widely used:
 - Rationale: A more well documented framework is easier to work with and will have a larger user-base, which makes resolving development problems faster
 - Ranking metric: # GitHub stars (higher is better)
- Personal experience:
 - Rationale: Modern web frameworks have a steep learning curve. Although limited, I have some experience using a few web frameworks
 - Ranking metric: Experience

I have chosen the React framework as it best fits the criteria specified (Rea, 2019b). Although plain JavaScript is more customizable than React, Vue and Angular, using plain JavaScript significantly slows development speed and I have therefore decided not to use it. In terms of performance and documentation, React and Vue are better choices than Angular. I have some personal experience using React and prefer its syntax to Vue, thus I have opted for React.

There are additional benefits to coding using a library like React. First, React integrates the model-view-controller pattern through its use of JSX syntax for templates. This enables object-oriented-like coding as it is easy to re-use components, which are the building blocks in the React library. Second, React supports using the Node Package Manager (NPM), which simplifies the installation of many of the dependencies used to build AIGVA. Third, React uses server-side rendering, which makes it

Frontend JavaScript Framework Comparison For the AIGVA Project				
	 Winner			
Requirement	React	Vue	Angular	Plain JavaScript/HTML
Downloads (Last 6 Months) ¹	5.0M	1.1M	0.4M	N/A
Size of Framework ²	100 KB	80 KB	500+ KB	0 KB
Github Stars ²	134.0K	145.6k	50.1K	N/A
Personal Experience	Yes	No	No	Yes

Sources: ¹NPM Trends <https://www.npm trends.com/angular-vs-react-vs-vue>, ²Github <https://github.com/>, Analysis created on 08/09/2019

Figure 4.5: AIGVA Frontend JavaScript Framework Comparison

ideal to build single-page applications when coupled with certain packages.

To use React as the web application framework, I use the create-react-app development environment and Node.js server. Rather than setting up the build and environment from scratch, the create-react-app tool "offers a modern build setup with no configuration" (Cre, 2019). This further speeds up development as the tool takes care of complicated tasks such as transpiling JavaScript using Babel and enabling WebPack. Coding in React supports JavaScript ES6 features that are designed to make the code more re-usable and readable to other developers. An example of this is the use of modules, that makes it possible to write cleanly structured code, sorted by folders. For the AIGVA codebase, this means that every separate view/page will have its own set of components associated as seen in the folder directory below (see Appendix C.1 for a full overview).

Listing 4.3: Web Application Component Structure (Located in: frontend/src)

```

1      src
2          components
3              App
4              Charts
5              Dashboard
6              Export
7              LabelReview
8              MLBlitzLabel
9              MLColdLabel
10             MLCustom
11             Navigation
12             Projects
13             Settings
14             VideoAnalysis

```

```

15         VideoFramePlayer
16         VideoManagement
17         constants

```

The frontend needs to communicate with the backend using asynchronous API calls. Although JavaScript provides the `fetch` method to handle API calls, I will use the `Axios` package instead, which is a "Promise based HTTP client for the browser." (Axi, 2019). `Axios` is preferable to the built-in `fetch` method for three reasons: (1) `Axios` automatically transforms response data to JSON, which is how a large portion of AIGVA data is structured both the front and backend, (2) `Axios` supports a wider range of browsers, which makes the tool more accessible to a greater audience, and (3) `Axios` makes it simple to chain API calls.

Listing 4.4: Web Application Video Upload Component (Located in: `/frontend/src/components/VideoManagement/UploadVideo.js`)

```

1  onUploadFinish = e => {
2      const file_name = e.signedUrl.split('?')[0]
3      const pid = this.props.store.get('pid');
4      const mid = this.props.store.get('mid');
5
6      let labels      = null;
7      let frame_count = null;
8
9      axios.get(API.BASE_API_URL + 'project/' + pid)
10     .then(response => {
11         const default_label = response.data.main_label
12         return axios.get(API.BASE_API_URL + 'process_video', {
13             params: {
14                 file_location: file_name,
15                 default_label: default_label,
16             }
17         })
18     })
19     .then(response => {
20         labels      = response.data.labels
21         frame_count = response.data.frame_count
22         return axios.post(API.BASE_API_URL + 'video', {

```

```
23         name:          this.state.file ,
24         location:       file_name ,
25         size:           this.state.size ,
26         fps:            response.data.fps ,
27         pid:            pid ,
28     })
29 })
30 // Additional API calls ...
31 }
```

To show the benefit of Axios, I have included a code snippet from the frontend component responsible for uploading videos. The `onUploadFinish` method gets called when a video has been uploaded to the file storage site. The step-by-step API process is shown in the table below. To summarize: (1) Axios gets the default label from the Core Data API and uses it to make an additional call to (2) Core Data API using the `process_video` route, which runs a task in the background to process the video for metadata and label generation. (3) The labels and video metadata are then used to create a video object in the database using a separate API call. The separation of API calls, enabled by Axios, makes for less verbose code on both the front- and backend as each call aims to do only one thing.

Charting

For charting I use a combination of two libraries: BizCharts and ApexCharts as shown in Figure 4.7. BizCharts is optimized for user interactions (Biz, 2019). For example, on the AIGVA dashboard users can view a summary of the total number of labels per class, which the user can hover over to get more detail. ApexCharts is designed to handle larger input data arrays (Ape, 2019). This is useful on the video analysis page, where a helpful bar is shown below the video to indicate which labels appear across the video. As videos may become long with frames changing often, the browser needs to draw a complex chart quickly, which is precisely what ApexCharts does well.

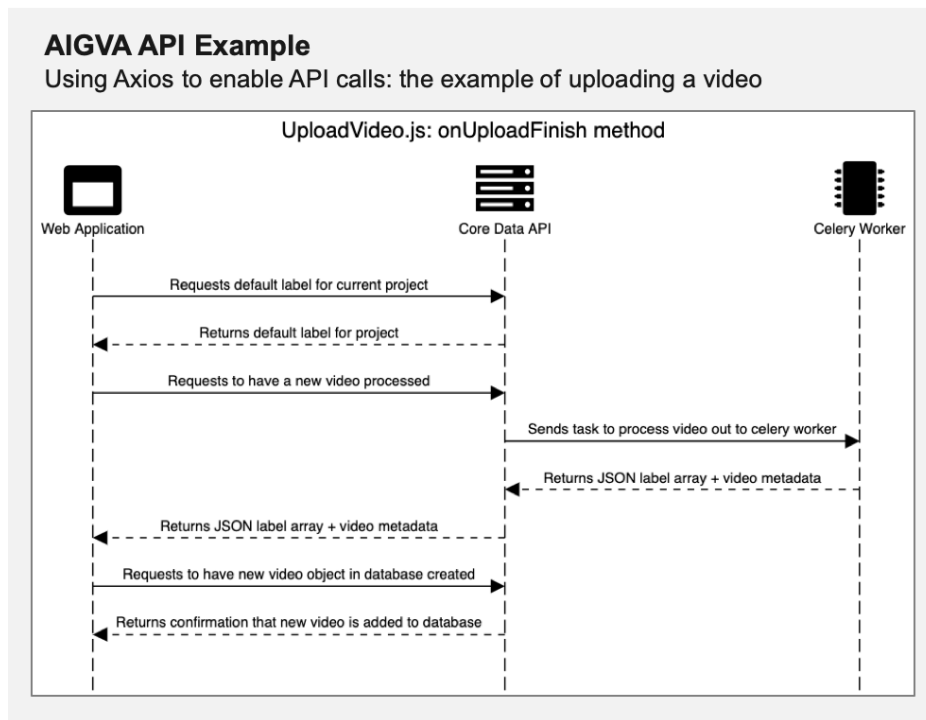


Figure 4.6: AIGVA API Example - onUploadFinish Method

File Management

Effective file management includes both successful uploading and downloading of files. These tasks are central to the AIGVA application. To create a successful architecture, I use two libraries to solve common issues with file management. First, I use Moment.js to "parse, validate, manipulate, and display dates and times" (Mom, 2019). Without Moment.js, rendering user friendly date-time values is cumbersome and requires verbose code. With Moment.js, it is possible to write one-liners like `moment(item.date).fromNow()` to render the time since a video was uploaded or a model was created in a human readable format. Second, I use FileSaver.js to tackle the problem of downloading files. FileSaver.js is a "solution to saving files on the client-side, and is perfect for web apps" (Fil, 2019). When a user wants to export data, they have to download a large .zip file sent over as the blob datatype (binary large object). Without FileSaver.js, a complex object creation/deletion process needs to happen on the client-side.

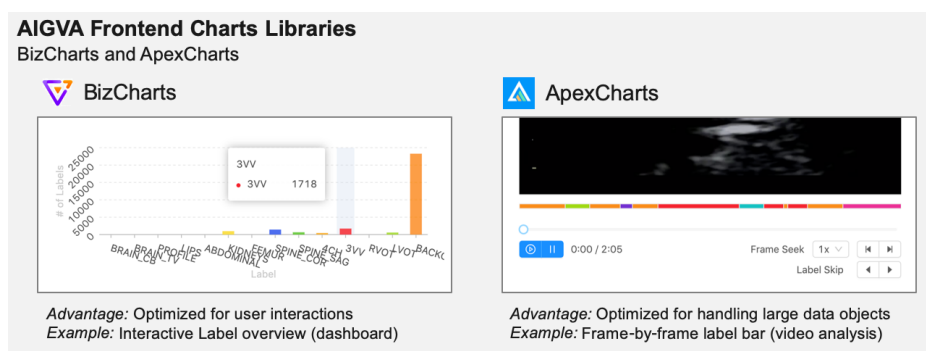


Figure 4.7: AIGVA Charting

Single-Page Application

To make AIGVA more user friendly and fast, I wanted it to be a single-page application. A single-page application is a website that dynamically re-renders a page rather than making a new sever call and refreshing the browser. When coupled with a routing library, React is an ideal framework to build single-page applications. While there are several routing libraries available, I have chosen the most popular and well documented library available, the react-router library. Specifically, I use the react-router-dom version, which is adapted to be used in modern browsers (Rea, 2019a).

A consequence of designing the AIGVA tool as a single-page application is needing to keep track of certain global parameters. For example, we need to know which project we have selected across routes. Keeping track of global parameters means needing access to a global store. A common approach to creating a global store is to use a state management library like MobX or Redux. However, as the requirements for a global store are limited for the AIGVA tool, adding a global state management library would add unnecessary size and complexity to the application. Instead, I opted to take advantage of Reacts higher order components feature. This allows you to wrap sub-components in other components using functions. The higher order components would keep their own state that could be modified using functions accessible to lower order components.

4.1.3 File Storage & Database

The file storage and database is the "memory" of the AIGVA tool. I have decided to separate the file storage and the database to adhere to the best practice of modern web development. The file storage is an Amazon S3 Bucket responsible for hosting static files such as machine learning models, images and videos. The database is a PostgreSQL relational database responsible for hosting metadata specific to the application such as labels, video locations, project details and machine learning model information.




File Storage

Although it is possible to store static files like images and videos in a PostgreSQL database, it is inefficient to do so and "not recommended at all" (Dyl and Przeorski, 2017). Instead using a file system is a better idea since it is both faster and uses less disk space. Two options exist for implementing a file system: (1) hosting files locally, or (2) using a cloud service. Using a cloud service is favorable for a tool like AIGVA as storage is more scalable and easier to configure. The two key requirements for a cloud storage provider are listed below.

Key requirements:

- Price:
 - Rationale: : The AIGVA tool stores data-heavy files like machine learning models and videos so the cheaper provider the better
 - Ranking metric: Monthly per-GB Prices (lower is better)
- React support:
 - Rationale: Uploading directly from the front-end is the most efficient option and native React support is therefore favorable
 - Ranking metric: Availability

The three most popular cloud service file storage providers are: AWS S3, Google Cloud and Microsoft Azure. I chose AWS S3 over Google Cloud and Microsoft Azure since it is comparable in price to the others and is the only one with a dedicated NPM react upload component. AWS S3 is a "simple storage service for static assets such as images on Amazon's servers" (Dyl and Przeorski, 2017).

File Storage Cloud Providers For the AIGVA Project			
	 amazon S3	 Google Cloud	 Azure
Requirement	Amazon Web Services	Google Cloud	Microsoft Azure
Price (\$ Monthly per-GB) ¹	\$0.023	\$0.023	\$0.026
React Support ²	Yes	No	no

Sources: ¹CloudBerry <https://bit.ly/2yNPIQa>, ²NPM <https://www.npmjs.com/package/react-s3>. Analysis created on 9/09/2019

Figure 4.8: AIGVA File Storage Comparison

To secure S3 uploads, I use the option to pre-sign the uploads by using the Core Data API to generate an upload link. Once received by the web application, this link grants temporary write/read access to the client. Now the web application is able to upload a file such as a video directly to S3 without needing to send any data to the web server, as seen in Figure 4.9. This significantly increases both speed and security.

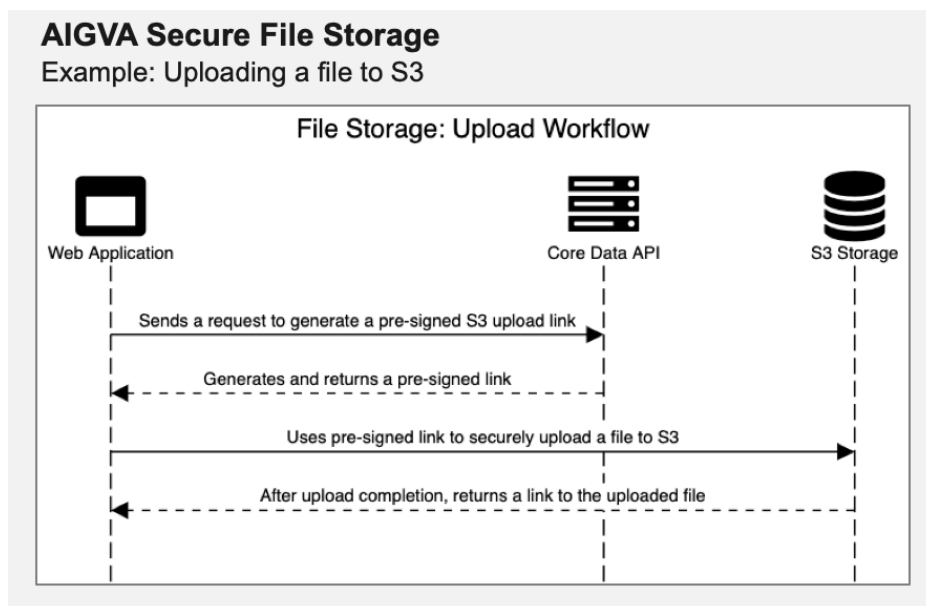


Figure 4.9: AIGVA Securing File Uploads

Database

I opted for a relational database over a NoSQL database for the AIGVA tool. NoSQL is great for prototyping because of the lack of a fixed schema. However, with NoSQL it is more cumbersome to do certain aggregations and joins that are needed in the AIGVA tool. I have chosen the PostgreSQL implementation of SQL since it is one of the most popular and widely documented implementations (Pos, 2019). Furthermore, PostgreSQL supports the JSON datatype, which is used to store the labels in the tool.

Listing 4.5: API Object-Relational Mapping Example (Located in: backend/app/app.py)

```
1 # Get associated videos from DB
2 video_links=[]
3 videos = Video.query.filter_by(pid=pid)
4 for video in videos:
5     video_links.append(video.location)
```

To integrate the PostgreSQL database into the Core Data API architecture I employ two tools: (1) an object-relational mapper, and (2) a database migration tool. First, I use the SQLAlchemy object-relational mapper, which works as an abstraction layer. Specifically SQLAlchemy provides "transparent conversion of high-level object-oriented operations into low-level database instructions." (Grinberg, 2014) This makes working with databases much simpler from a development perspective as you can interact with them like objects. See the code example above for an example of how the SQL database can be queried for videos associated with a specific project, all without writing a line of SQL. This increases safety as it prevents SQL injections. Second, I use the Flask-Migrate library to handle schema changes during development. As mentioned before, one of the disadvantages of working with relational databases is their inflexibility to schema changes, which is inevitable during development. To mitigate this issue, I use the Flask-Migrate database migration framework "to track of changes to a database schema, allowing incremental changes to be applied" (Grinberg, 2014). For a full overview of the database schema, see Appendix D.4.1.

4.2 Feature Implementation - Video Management

In this section I will discuss the challenges of implementing one of the core components of the tool, the video management component. This consists of the frame-by-frame video player and the export tool.

4.2.1 Frame-by-frame Video Player

One of the major challenges of the project was to create an in-browser video player that met the requirements of the application. The key feature that the video player needed to support was frame-by-frame navigation. The browser would need to constantly be aware of what frame was being played so that the UI displaying the label prediction and frame statistics can be updated (see Figure 4.10). None of the existing web video players such as React-Player and Video-React offered this support. Therefore I needed to write my own in-browser video player.

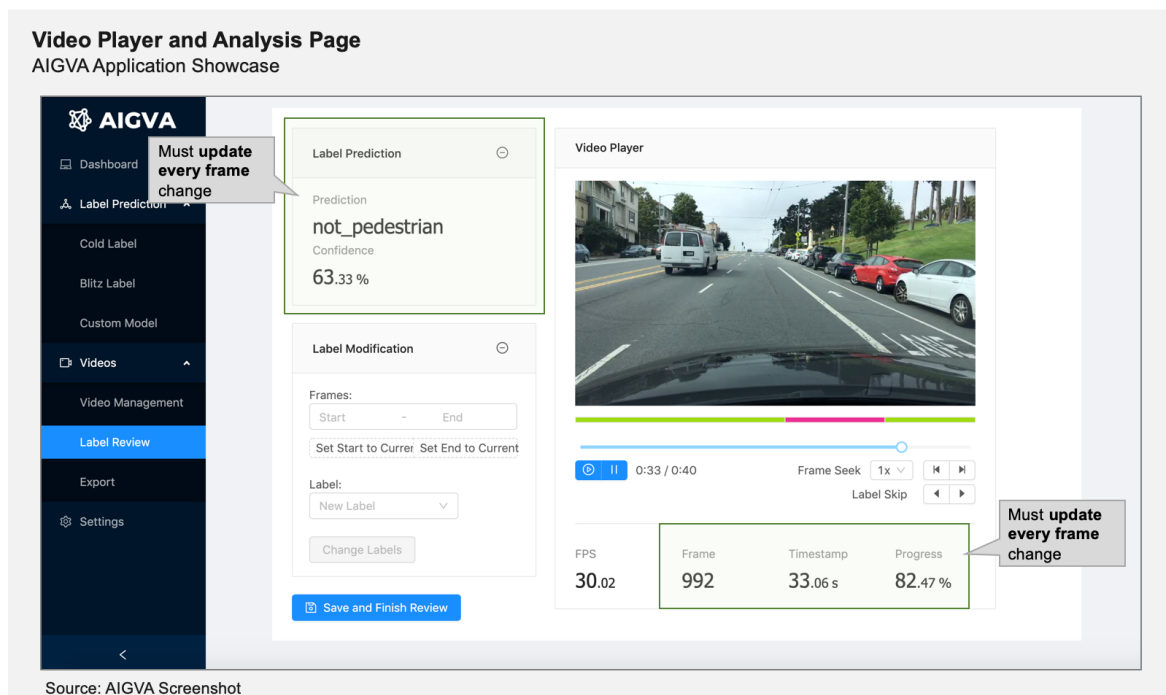


Figure 4.10: AIGVA Video Player and Analysis Page

After much experimentation, I was able to implement a video player that met all the aforementioned criteria. First, I attempted to display sequences of pictures instead

of videos, however that proved too slow. Then, I also attempted to modify existing libraries to include frame-level navigation, however that proved too time-consuming. Ultimately, the solution that worked was to build my own player from scratch based on the HTML5 Video Media element specification ¹. The specification, part of the official HTML5 release, provides a set of attributes and events you can use to implement features from scratch. I utilized these low-level building blocks to develop the following features:

- Play/pause buttons
- Video range slider that allows users to navigate to any time in the video
- A "frame change" event listener that triggers a UI update
- Frame seeking buttons that could skip X number of frames forwards or backwards
- Label skip feature that queries the label set for the next different label and updates the video
- Keyboard shortcuts to navigate between frames and labels

To illustrate the idea behind the implementation I have included part of the code for the video player to show the event listener logic - see Listing 4.6. The video frame player exploits React component lifecycle, which triggers the `componentDidMount()` function to run when a component is first rendered. This calls the `startUpdatingFrames()` method which initializes an event listener that queries the state every interval. Every iteration of the event listener is a call to the `updateFrames()` method, which is responsible for two things: (1) extracting frame-level information from the HTML5 Media Element, and (2) triggering a global UI update which updates the state of information that depends on the current frame (all the items within the green boxes on Figure 4.10).

¹<https://developer.mozilla.org/en-US/docs/Web/HTML/Element/video>

To determine the current frame, we need both the current time which we obtain from the HTML5 "currentTime" attribute and the FPS which we obtain from the components state. Unfortunately, we cannot acquire the FPS from within the browser - or at least we cannot do so without making the application painfully slow. To overcome this problem a function was built in the Core Data API that can automatically extract this information on upload using OpenCV (see Appendix D.1.1). Then when we load the video analysis page, this information is simply queried from the database, alongside the labels and other metadata needed. When the user exits the page, the `componentWillUnmount()` method gets invoked through React's component lifecycle logic. This method kills all active event listeners, which is critical to prevent memory leaks from dangling event listeners. Further implementation details can be found in Appendix D.1.1, but have not been included here for the sake of brevity.

Listing 4.6: Custom-Built Video Frame Player Snippet (Located in: `frontend/src/components/VideoFramePlayer/index.js`)

```
1 class VideoFramePlayer extends Component {
2   constructor(props){
3     super(props)
4
5     this.vidRef = React.createRef();
6     this.state = { ... INITIAL_STATE }
7   }
8
9   startUpdatingFrames() {
10    // Check that no other intervals exist to prevent multiple
11    // listeners
12    if (!this.timerID){
13      this.timerID = setInterval(() => this.updateFrames(),
14      INTERVAL_MS);
15    }
16  }
17
18  stopUpdatingFrames() {
19    clearInterval(this.timerID);
20  }
```

```
19
20   componentDidMount() {
21       // Initialize event listener to update UI on frame change
22       this.startUpdatingFrames();
23       this.vidRef.current.addEventListener('canplay', this.
handleCanPlayEvent)
24   }
25
26   componentWillUnmount() {
27       // Kill event listener to prevent memory leaks on component
dismount
28       this.stopUpdatingFrames();
29       this.vidRef.current.removeEventListener('canplay', this.
handleCanPlayEvent)
30   }
31
32   handleCanPlayEvent = () => {
33       this.setState({can_play: true})
34   }
35
36   updateFrames() {
37       const current_frame = Math.floor(this.vidRef.current.
currentTime.toFixed(5) * this.state.fps);
38       const current_time = this.vidRef.current.currentTime
39       const current_progress = current_time/this.vidRef.current.
duration
40       this.setState({
41           frame: current_frame,
42           time: current_time,
43           progress: current_progress,
44       })
45       this.props.sendFrames(current_frame);
46   }
```

4.2.2 Export Tool

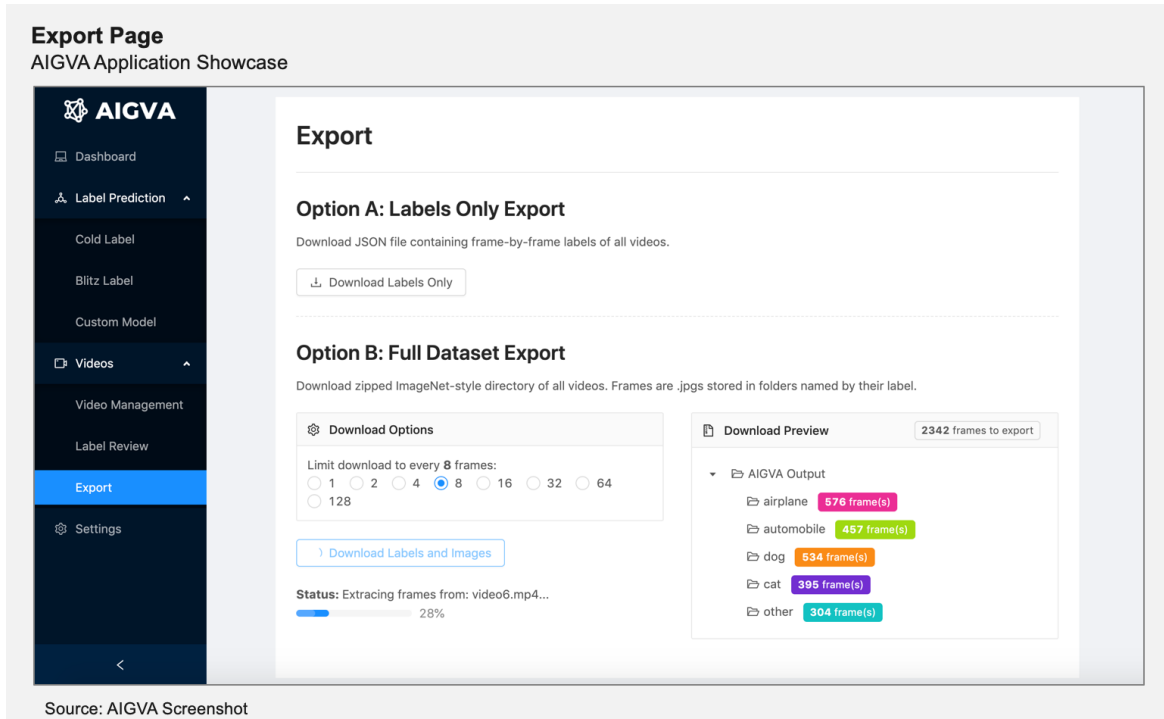


Figure 4.11: AIGVA Export Page

Building an efficient export functionality also proved an interesting technical challenge. I implemented two version of the export functionality: (1) a labels only export, and (2) a full dataset export which included both labels and images. The first "labels only" functionality was designed to offer a fast way to download frame-by-frame labels in a JSON file. This was a relatively simple implementation as it only required a single API call to the Core Data API to extract labels and then leverages FileSaver.js to download the file on the client side. The second "full dataset" export proved more challenging. This export was requested early on by a key stakeholder who wanted and ImageNet style output that placed images in "folders named by their label, so that they can be fed directly into any ML pipeline" ². I will briefly explain its implementation it highlights how the three key components of the software architecture (the web app, the Core Data API and the File Storage) interact.

²Email correspondence with Prof. Bernhard Kainz

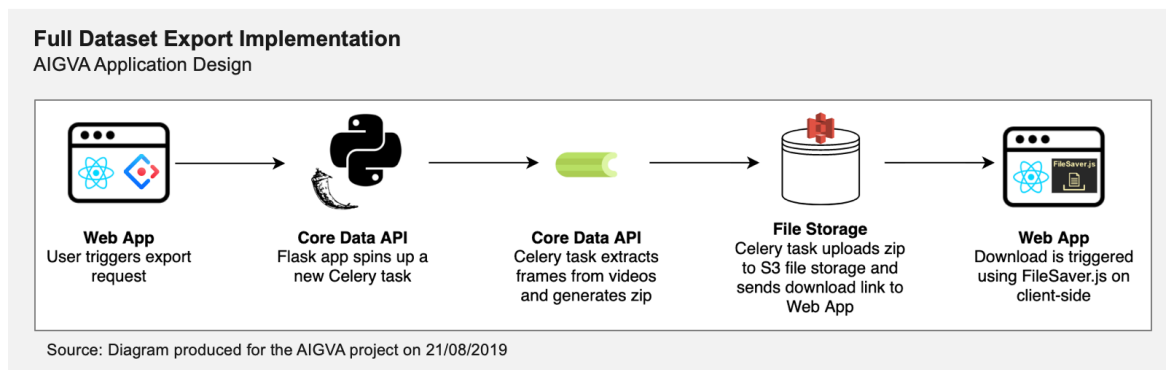


Figure 4.12: AIGVA Export Implementation Details

A "full dataset export" was implemented in five steps as shown in Figure 4.12. First, a user triggers an export in the browser. This hits the `api/full_download` route of the Core Data API. Then, within the Core Data API, a new celery task is created to handle this task and a URL is immediately returned to the Web App, which the browser uses to display the progress bar and let the user know when the task is finished. The Celery task then begins executing, which creates a folder structure with a subfolder for each label (see Appendix D.1.2 for details). This task then iterates over all the videos in the project and executes the `api/export_video_to_folder` method - see Listing 4.7. This method uses the OpenCV library to iterate over frames in a video to store them in a folder. One interesting hyperparamter to note here is the "skip factor". This is an optimization developed to skip over every Nth frame. The user can specify this factor in the web app as shown in Figure 4.11 in the "Download Options." By default, this option is set to 1, meaning no frames are skipped. But since this function operates in linear time, increasing the the skip factor to N will reduce them time taken by a factor of N. Once all frames have been extracted, the Core Data API compresses the folder to a .zip file and uploads it to the video file storage. Ultimately, a the link to this zip folder is sent to the web application and a download is triggered using the FileSaver.js framework.

Listing 4.7: Video Export Method (Located in: `backend/app/helpers.py`)

```

1 def export_video_to_folder(video_link, labels, skip_factor=1,
  compression_factor=40):
2     """

```

```
3     Export a video to folder.
4
5     Args:
6         video_link [str]: link to video download
7         labels [json]: array of labels for video
8         skip_factor [int]: number of images to skip
9         compression_factor [int]: image compression factor (0 is none
10         , 100 is full)
11     """
12
13     cap = cv2.VideoCapture(video_link)
14
15     while(cap.isOpened()):
16         frameId = cap.get(1)
17         ret, frame = cap.read()
18         if (ret != True):
19             break
20         if (frameId % math.floor(skip_factor) == 0):
21             filename = "./exports/%s/%s_frame%d.jpg" % (
22                 labels[int(frameId)]['Label'], video_name, int(
23                     frameId))
24             cv2.imwrite(filename, frame, [
25                 int(cv2.IMWRITE_JPEG_QUALITY),
26                 compression_factor])
27
28     cap.release()
```

4.3 Feature Implementation - Label Prediction

In this section I will discuss the challenges of implementing one of the most challenging undertakings as part of the AIGVA project, to develop the infrastructure to support label prediction and to implement the three labelling techniques: Cold, Blitz and Custom Model labelling. All three labelling techniques have one thing in common, they all result in a machine learning model that is used to predict labels. To reflect this commonality and to prevent code duplication, they all share the same prediction engine. Therefore I will first discuss how I implemented the three labelling techniques, up to the point of them producing a machine learning model, and then I will describe how the prediction engine works collectively.

4.3.1 Cold Label

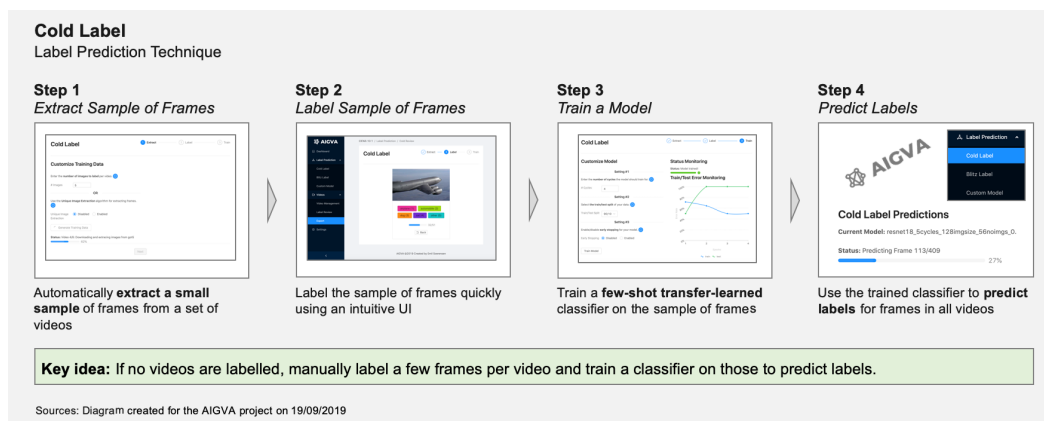


Figure 4.13: AIGVA Cold Label Prediction Technique Review

Recall the intuition behind the Cold Label technique: if no videos are labelled, manually label a few frames per video and train a classifier on those to predict labels. This process has three steps: (1) extract a sample of frames, (2) manually label sample of frames, and (3) train a classifier on the labelled frames. I will discuss the implementation challenges of these three sections.

Cold Label: Step 1 - Extraction
AIGVA Application Showcase

Cold Label 1 Extract 2 Label 3 Train

Customize Training Data

Enter the number of images to label per video. ①

Images

OR

Use the Unique Image Extraction algorithm for extracting frames.

① Unique Image Extraction ☒ Disabled ☐ Enabled

Option to enable Unique Image Extraction

Status: Video 4/6: Downloading and extracting images from got5
62%

Source: AIGVA Screenshot

Figure 4.14: AIGVA Cold Label Implementation - Step 1

(1) Extracting Sample of Frames

The implementation of the overall image extraction architecture is straight-forward. First a user triggers a request to create a new Cold Label model. Then a Celery task is created to sample extract frames from videos. There are two possible methods of frame/image extraction: uniform sampling or unique image extraction. The methods will be described in detail below. Once images have been extracted they are uploaded to the file storage and links to these are sent to the web application, which is now ready for step 2 - manually labelling a sample of frames.

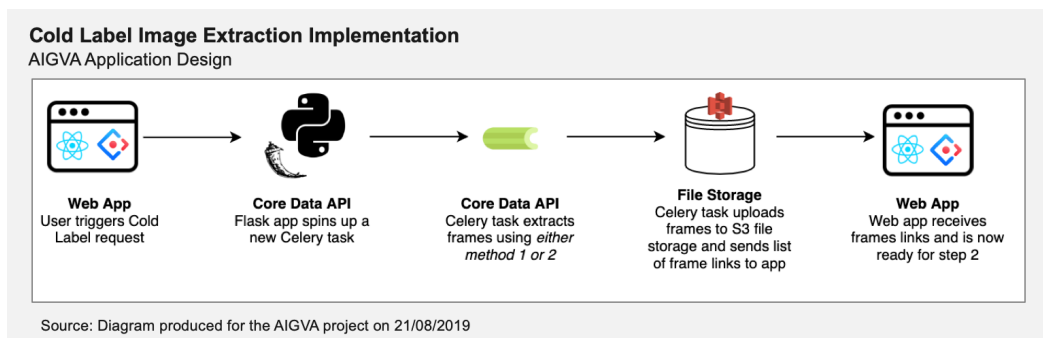


Figure 4.15: AIGVA Cold Label Implementation Details

Image Extraction Method #1: Uniform Sampling

One simple implementation to extract images is to have the user to specify the number of frames they wish to extract and then uniformly sample that number of frames from a video. For example, say a user had a 10 second long video running at 30 frames per second (300 total frames). No suppose, to save time, the user only wanted to label 5 images. Then the uniform sampling method would extract: frame 0, frame 60, frame 120, frame 180 and frame 240. See Algorithm 1 below written as pseudocode (See Appendix D.2.1 for full implementation). Note the calculation of the "skip factor" which is calculated as the total number of frames divided by the number of frames the user wants to label. As the video is read as a stream, a modulo check is implemented using the skip factor, which limits the number of images to what is desired by the user.

Algorithm 1 Uniform Image Sampling [Image Extraction Method 1]

Require: *video, images_to_save*
capture \leftarrow *openStream(video)*
total_frames \leftarrow *capture.getFrames()*
skip_factor \leftarrow *total_frames/images_to_save*
while *capture.isOpened()* **do**
 ret, frame \leftarrow *capture.read()*
 if *ret* \neq **true** **then**
 break
 end if
 i \leftarrow *capture.getFrameID()*
 if (*i* mod *skip_factor*) == 0 **then**
 saveImage(frame)
 end if
end while
capture.closeStream()

Image Extraction Method #2: Unique Image Extractor

The clear downside to the previous method is that it entirely ignores the content of the video. Suppose a video changes frames often, then it would make sense to label more than 5 frames for the same 10 second video (e.g. ultrasound). If only 5 frames are labelled, chances are that the classifier will perform poorly as it was trained on an insufficient training set. Now suppose instead the 10 second video was that of a non-moving picture, or rather, a still-frame as is often the case in the real world (e.g. CCTV footage). In the latter case it would not make much sense to extract 5 frames, since 1 frame should suffice. This idea requires a method to extract a set of the most "unique" images from a video. This method is closely related to the concept of active learning. Recall, how active learning methods attempt to find the most "interesting" samples and sends them to a human to label. While it is out of the scope of this project to build a full active learning algorithm, one can approximate such an approach by guessing what the most "interesting" samples are. In the context of videos, it means finding the most meaningful/unique still frames out of a video.

To build an algorithm to extract the most unique images from a video, a process was designed that compared adjacent frames for a significant differences. If the difference was above a certain threshold, meaning that a frame was likely "unique", then the algorithm would extract that image - see Algorithm 2. The main challenge was defining what "significant difference" between frames meant. I experimented with several approaches to compare images such as the use of cross correlation and using feature extraction. However, these method proved inconsistent and computationally expensive. Instead, I opted for the use of perceptual hashing. This is a method that computes a hash vector from an image matrix. Unlike cryptographic hashes where tiny nudges in the input can drastically change the hash, perceptual hashes "are close to one another if the features are similar" (pHa, 2019). I opted for OpenCV's implementation of perceptual hashing which outputs a 1×8 dimensional vector (Ope, 2019).

Algorithm 2 Unique Image Sampling [Image Extraction Method 2]**Require:** *video, skip_factor, hamming_threshold*

```

1: capture  $\leftarrow$  openStream(video)
2: last_hash  $\leftarrow$  null
3: while capture.isOpened() do
4:   ret, frame  $\leftarrow$  capture.read()
5:   if ret  $\neq$  true then
6:     break
7:   end if
8:   i  $\leftarrow$  capture.getFrameID()
9:   if (i mod skip_factor) == 0 then
10:    current_hash  $\leftarrow$  perceptualHash(frame)
11:    if last_hash == null then
12:      saveImage(frame)
13:      last_hash  $\leftarrow$  current_hash
14:    else
15:      dist  $\leftarrow$  hammingDistance(current_hash, last_hash)
16:      if dist  $\geq$  hamming_threshold then
17:        saveImage(frame)
18:        last_hash  $\leftarrow$  current_hash
19:      end if
20:    end if
21:  end if
22: end while
23: capture.closeStream()

```

To obtain a similarity score, a distance measure was required to compare the two perceptual hash vectors. I experimented with Euclidean distances but they performed poorly. After research on the topic, I found the poor performance of Euclidean distances documented in the literature, with researchers having found that Hamming distances are more optimal for semantic comparisons (Norouzi et al., 2012). Results were much more consistent with a Hamming distance measure and I therefore I opted for this distance measure instead. Note that the Hamming distance between vectors $x = (x_1, x_2 \dots x_n)$ and $y = (y_1, y_2 \dots y_n)$ is written as $D_{Hamming}(x, y)$

$$D_{Hamming}(x, y) = \sum_{i=1}^n \delta(x_i, y_i) \quad (4.1)$$

$$\delta(x_i, y_i) = \begin{cases} 0 & x_i = y_i \\ 1 & x_i \neq y_i \end{cases} \quad (4.2)$$

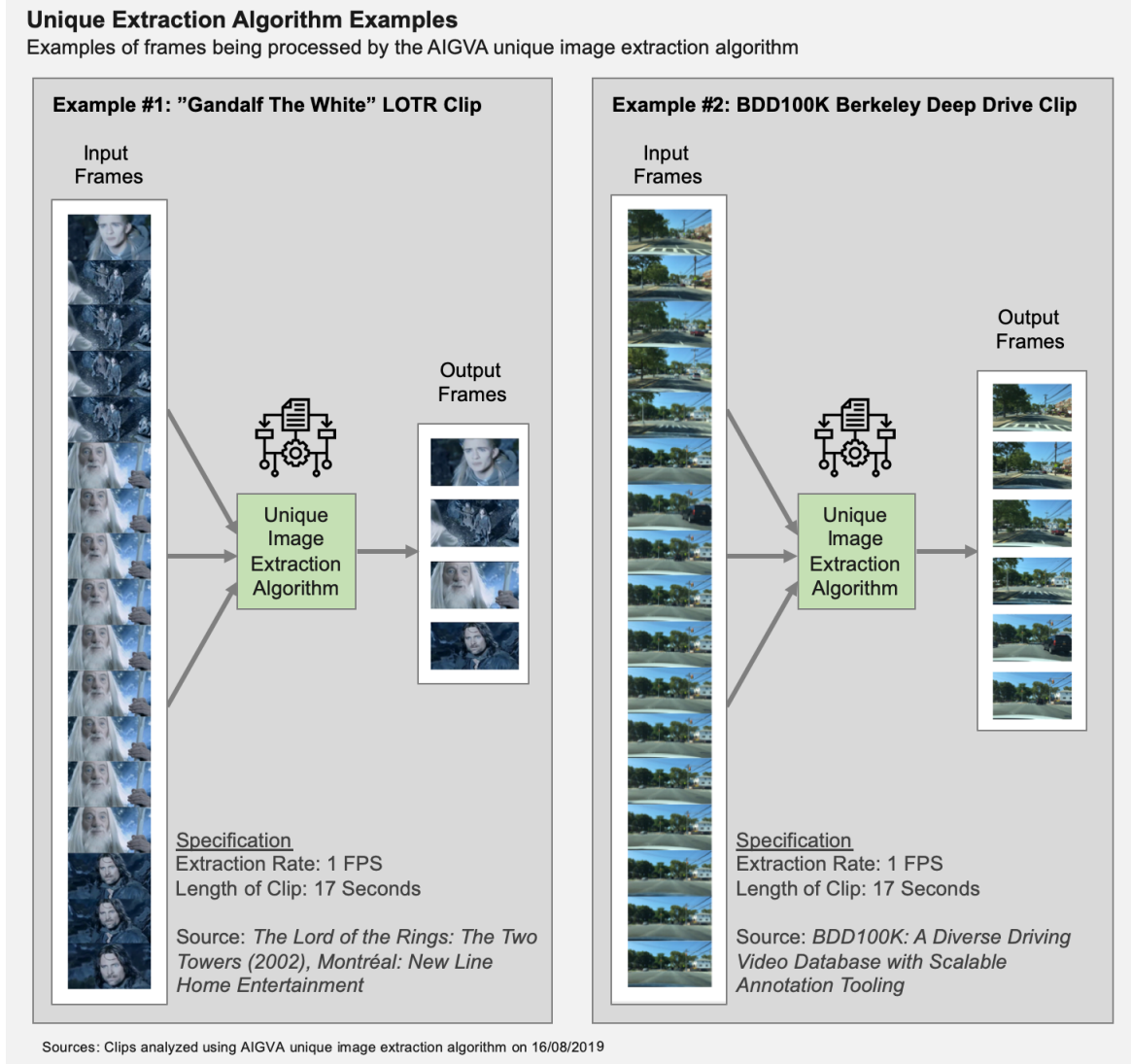


Figure 4.16: AIGVA Unique Image Extraction Method Using Perceptual Hashing

To demonstrate the use of the unique image extraction algorithm, an experiment was conducted with two examples. See Figure 4.16 for the results of the experiment. In the first example, a 17 second clip from the feature film Lord of the Rings was sampled at a frame-rate of 1 FPS. The 17 frames were processed by the unique image extraction algorithm and returned a set of four images. By visual inspection, we can observe that it correctly extracted four of the most unique images. In another more

tricky example, autonomous vehicle data was used from the Berkeley Deep Drive dataset. This was also a 17 second clip sampled at 1 FPS which produced 17 frames. This time the algorithm returned six frames. Note again how the output frames all significantly different from each other. Note also how the algorithm correctly only extracted only one from the last 11 frames where the car is waiting at an empty intersection.

(2) Label Sample of Frames

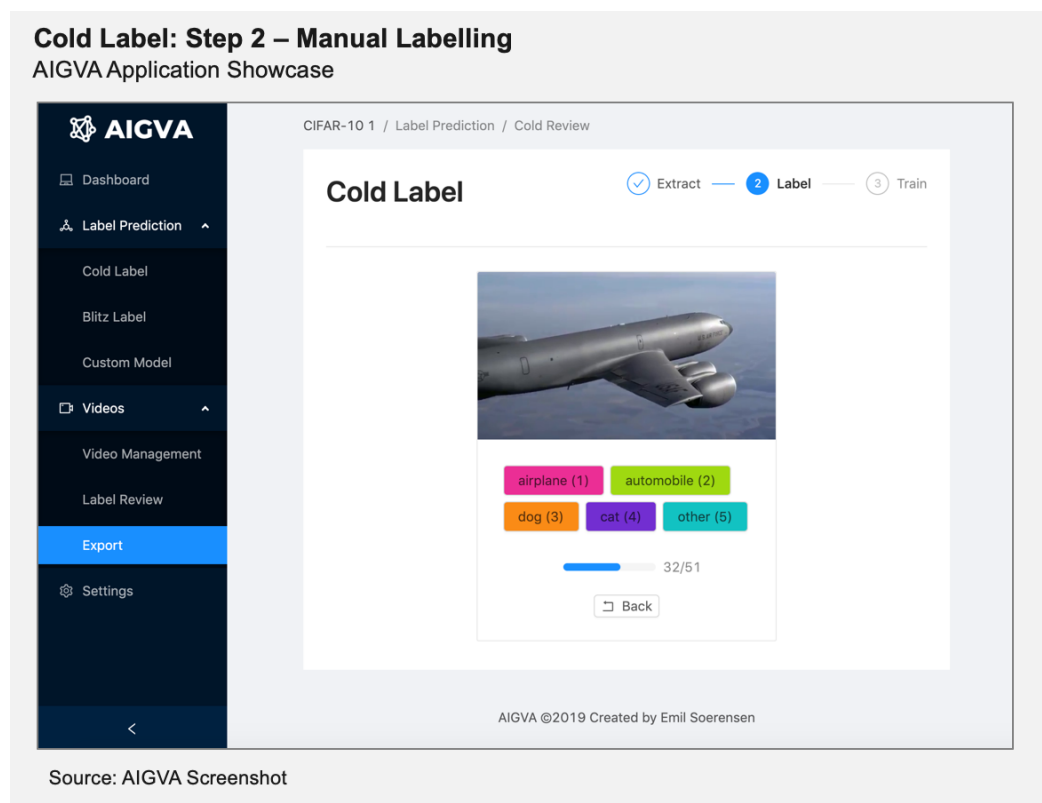


Figure 4.17: AIGVA Cold Label Implementation - Step 2

Once a set of sample images had been extracted using either of the two aforementioned methods, the user can manually label these in the browser. To implement this functionality I built a label review tool that displays all the labels associated with a project alongside the image to be labelled. To speed this process up, keyboard shortcuts were made available. An example of a user labelling a set of extracted frames can be seen in Figure 4.17.

(3) Training a Model

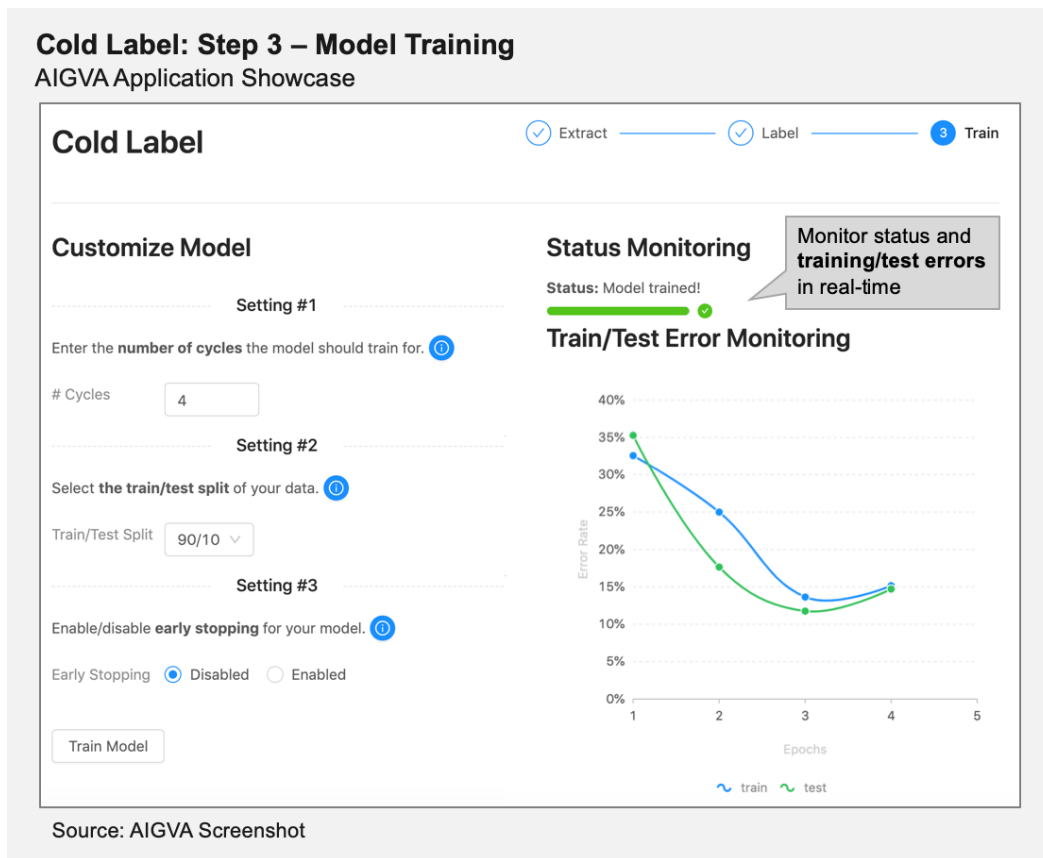


Figure 4.18: AIGVA Cold Label Implementation - Step 3

After labelling the extracted images, the user will train a machine learning model which can later be used to predict labels for entire videos. This was one of the more challenging implementations as it meant building the infrastructure to support training and monitoring a machine learning model from the browser. There were three parts to the implementation: selecting the right machine learning model, customizing the selected model, and training the model in the browser.

Model Training #1: Selecting The Model

Selecting a model requires balancing trade-offs in accuracy, training time and model size. Using the most accurate model will not necessarily create the best tool if it means training time is significantly longer. Therefore, to select the best model an experiment as run to compare CNNs across these metrics. The experiment used

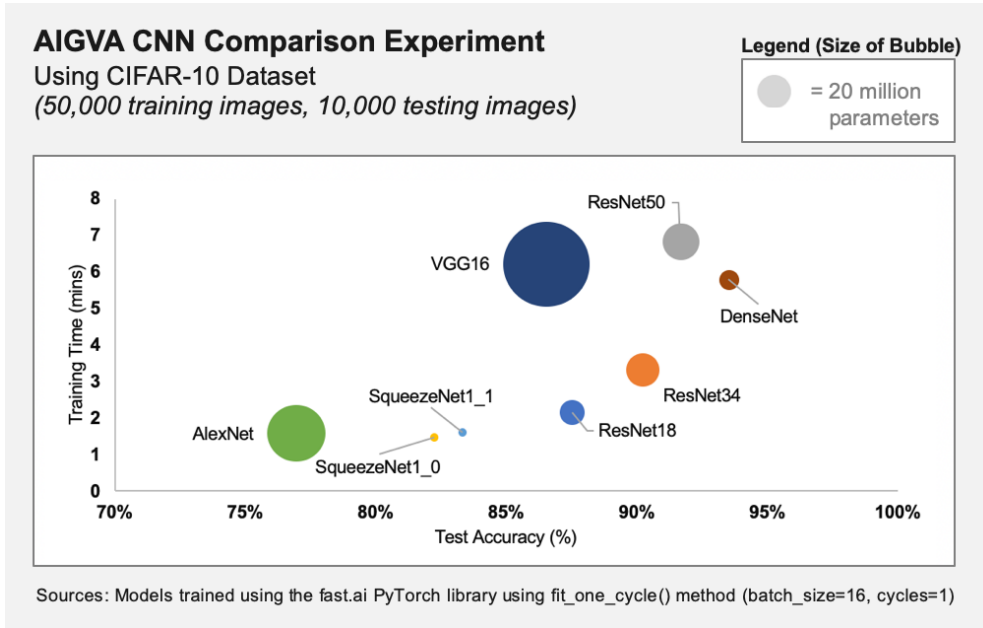


Figure 4.19: AIGVA Label Prediction CNN Architecture Comparison Chart

the CIFAR-10 dataset which consists of 60000 32x32 colour images in 10 classes, with 6000 images per class, split into 50000 training images and 10000 test images (Krizhevsky et al., 2009). I used a set of popular CNN model architectures on this dataset including ResNet, DenseNet, VGG, SqueezeNet and AlexNet. The experiment was run on a P4000 GPU and all the networks were pre-trained on ImageNet and then subsequently fine-tuned on the CIFAR-10 dataset during training. The reason for pre-training on ImageNet is to leverage the benefits of transfer learning. Specifically, researchers have shown that using CNNs pre-trained on natural images (i.e. ImageNet) train faster and are more robust than CNNs trained from scratch, while still retaining performance (Tajbakhsh et al., 2016). I used the fast.ai PyTorch `fit_one_cycle` method which optimizes learning rate, momentum and weight decay to optimize hyperparameters³.

The results of the experiment is shown in Figure 4.19 and Table 4.1. ResNet34 was chosen over the other variants because of the balanced performance across all three metrics. In terms of accuracy, DenseNet performed the best achieving 93.55% accuracy on the test set. However, DenseNet also took nearly 6 minutes to train. Comparatively, ResNet34 achieved nearly the same accuracy, 90.26%, in nearly half

³`fit_one_cycle` training optimization: <https://docs.fast.ai/training.html>

Model Name	Accuracy (%)	Time To Train (s)	Model Parameters
ResNet18	87.54	130	11689512
ResNet34	90.26	199	21797672
ResNet50	91.68	409	25557032
SqueezeNet1_0	82.27	89	1248424
SqueezeNet1_1	83.30	97	1235496
AlexNet	76.95	95	61100840
VGG16	86.54	373	138365992
DenseNet	93.55	347	7978856

Table 4.1: AIGVA Label Prediction CNN Architecture Comparison Table

the time. Since the first prototype of the AIGVA tool is implemented on CPUs, training speed matters significantly more than if it was implemented on GPUs. Therefore I will opt to use ResNet34 over DenseNet as the main model. The other variants of ResNet, 18 and 50, outperform ResNet34 in terms of training time and accuracy respectively. However ResNet18 also comes with a 2.5% performance penalty and ResNet50 takes more than twice as long to train compared to ResNet34. Other models including SqueezeNet, VGG and AlexNet all performed worse compared to ResNet34 in terms of accuracy and additionally the training speed gains were not large enough to consider switching from the ResNet34 model.

Model Training #2: Customizing The Model

One of the most challenging aspects of practical machine learning is to select the hyperparameters. The goal behind the implementation of hyperparameter selection was to simplify it as much as possible for the user. For that reason, control was limited to three of the most important factors affecting the training of a machine learning model: the number of training cycles, the train/test split, and the option to enable early stopping. The number of training cycles will affect the time taken for the model to train and while a higher value will reduce the training error it may cause overfitting. To prevent overfitting the option to enable early stopping was included. The implementation of both the number of training cycles and early stopping is shown in Listing 4.8.

Listing 4.8: Core Data API Model Training Code Snippet (Located in: Backend/app/app.py)

```

1 for epoch in range(input_cycles):
2     ...
3     # Break if early stopping and test acc lower than prev
4     if ((input_early_stopping) and
5         (test_accuracy < prev_test_accuracy)):
6         break

```

Model Training #3: Training The Model

Implementing model training controlled from the browser utilizes all the key innovations of the backend as shown in Figure 4.20. First, a user triggers a model training request and the Core Data API spins up a new Celery task to handle training the model. Next, the Celery task containing all the model training code runs for several epochs, constantly reporting the status to a separate server. To keep the user informed throughout the training process, the web app queries this separate server for status information. The web app uses this information to display both an overall model training progress bar and a graph displaying the real-time test/train error. After model training is completed, the Core Data API uploads the model file to the S3 file store and creates a new model object in the PostgreSQL database with a link to the file store. Ultimately, the web app is notified of the training completion and is now ready for prediction. See Appendix D.2.1 for full implementation details.

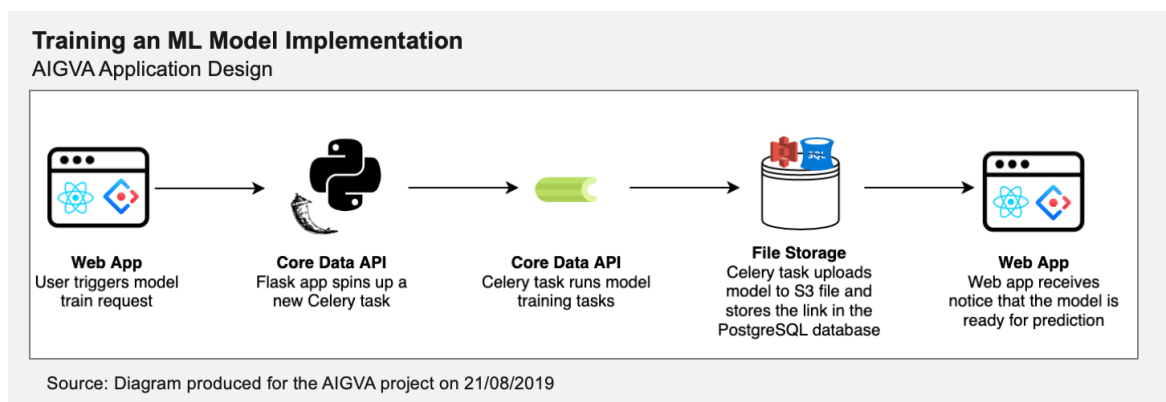


Figure 4.20: AIGVA Model Training Implementation

4.3.2 Blitz Label

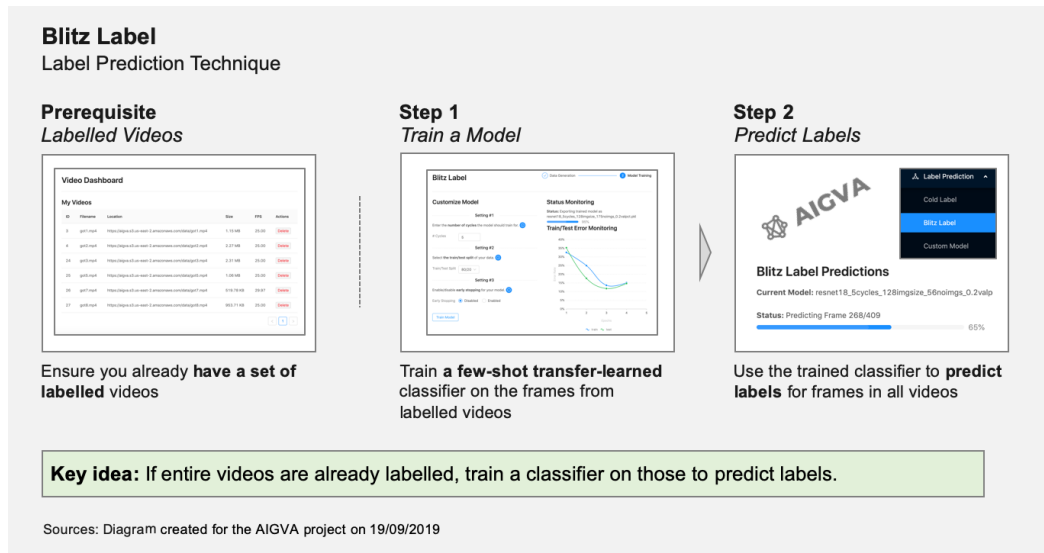


Figure 4.21: AIGVA Blitz Label Prediction Technique Review

Recall the intuition behind the Blitz Label technique: if entire videos are already labelled, train a classifier on those to predict labels. This process has two steps: (1) ensure you have pre-labelled videos, (2) train a classifier on the labelled frames. I will briefly discuss the implementation challenges of these two sections.

(1) Pre-Labelled Videos

Once a user has marked a video as "Reviewed", then it gets added to the list of videos eligible to be included for "Blitz Label". To utilize these reviewed videos to create a dataset for a Blitz Label model, a user is first prompted to specify how many frames they wish to skip. The idea behind this implementation was to offer the option to not extract every single image since it may lead to overfitting. When the user clicks "Generate Training Data" a request is sent to the Core Data API to create a Celery task to handle this process. Upon completion of the task, the "Next" button becomes activated and a model can be trained.

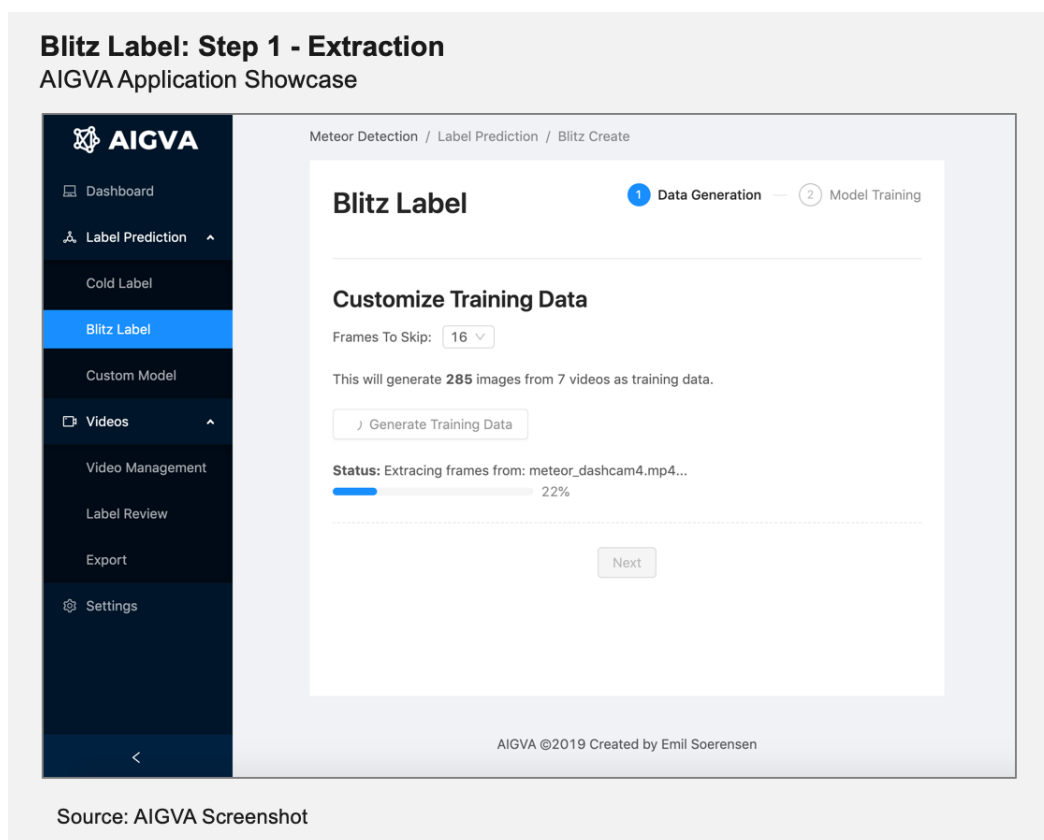


Figure 4.22: AIGVA Blitz Label Implementation - Step 1

(2) Training a Model

Training a Blitz Label model works in the same way as a Cold Label model. The only difference is the input data is given in the form of a .zip file instead of a links to individual images (see full implementation in Appendix D.2.2 for details). Because the difference is only 5-10 lines of code, I will refer the reader to the previous Section 4.3.1 for the sake of brevity.

4.3.3 Custom Model

Recognizing that there are limitations to the Cold Label and Blitz Label models, it was decided to add the option for machine learning researchers to use their own models. To ensure that this was done in the most effective way possible, I reached out to several stakeholders to discuss this topic before building it. It quickly became apparent that a simple Python interface that the researchers could adapt to their own

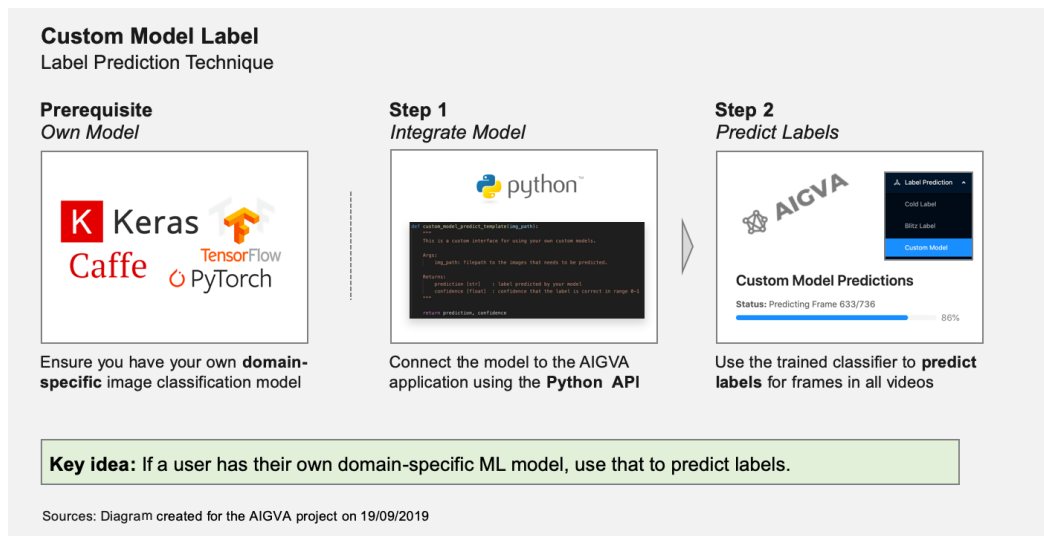


Figure 4.23: AIGVA Custom Model Label Prediction Technique Review

models was the simplest method. However, this also required the Dockerization of the application to ensure deployment would be effortless - see section 4.4.2 for a discussion on this.

Listing 4.9: Custom Model API (Located in: Backend/app/custom_{model}.py)

```

1 def custom_model_predict(img_path):
2     """
3     Custom interface for using your own custom models.
4
5     Args:
6         img_path: filepath to the images that needs to be predicted.
7
8     Returns:
9         prediction [str] : label predicted by your model
10        confidence [float] : confidence that the label is correct in
11        range 0-1
12    """
13    # Write your code here
14
15    return prediction, confidence

```

The Python interface to be filled in by machine learning researchers is shown in

Listing 4.9. The method is invoked from the label prediction engine and passes a path to the image that is to be predicted. The engine expects the method to return two values: (1) the label predicted by the model, and (2) the confidence that the label is correct - see Appendix D.2.3 for an example implementation of a custom model. Once this interface has been implemented the Custom Model option becomes available in the web application. Label prediction can then be executed like it is when using a Cold or Blitz Label model.

4.3.4 Label Prediction Engine

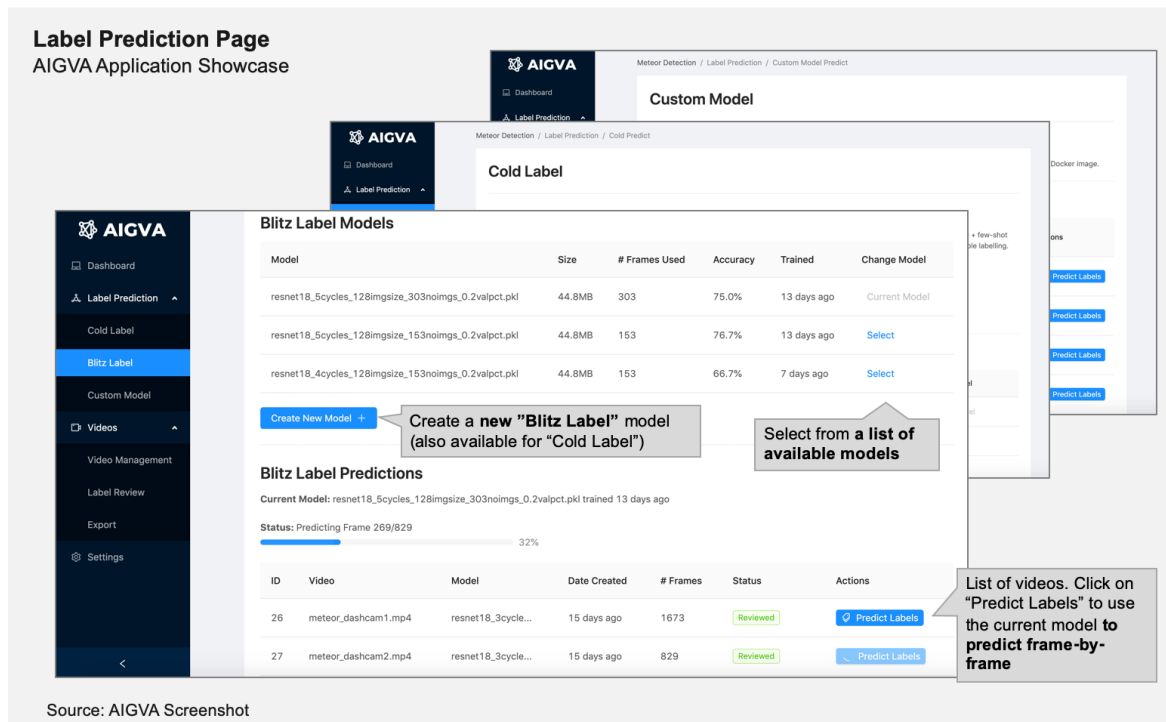


Figure 4.24: AIGVA Label Prediction Pages

The prediction engine is common to all three models. When a user navigates to any of the label prediction pages, they will be met with similar pages that gives them access to models of that type, either Cold, Blitz or Custom, and gives them the ability to run these models to predict labels (see Figure 4.24).

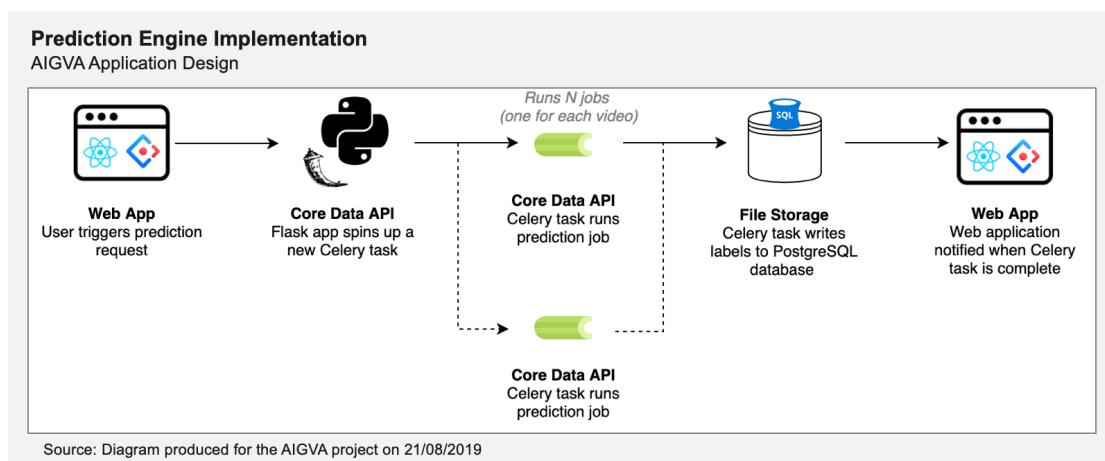


Figure 4.25: AIGVA Label Prediction Implementation

The high level implementation of the label prediction engine is split into five steps

(see Figure 4.25). First, a user clicks on a "Predict Labels" button from either of the three label prediction pages. The Core Data API receives this request and creates a new Celery task. One of the core requirements of the label prediction engine is the ability to run multiple predictions concurrently. This was the requirement that motivated the asynchronous Message-Broker architecture discussed in Section 4.1.1. Here we can observe how the architecture serves its purpose. Specifically, if multiple videos are selected by the user to be predicted, then multiple Celery tasks will be created. Then each task will run a label prediction job on its video. This is a complicated task that involves loading both the video and model into memory, followed by iterating over frames and running inference - see Appendix D.2.4 for the full implementation of this Celery task. When inference is complete the newly predicted labels are written to the PostgreSQL database and the web application is notified that the task is completed.

4.4 DevOps

DevOps refers to the process of integrating software development with quality assurance and IT operations with the single goal of successfully to deploying the end product to users (Kim et al., 2016). Having already discussed the development activities of the project in depth, we now turn to the two main activities of DevOps: quality assurance by testing and IT operations through deployment of the application.

4.4.1 Testing

The two crucial aspects of the application to test are the web app and the Core Data API. To test the web app I conducted several rounds of user interviews. Such usability testing helped to "validate design decisions" and let users be "exposed to the capabilities of the system early" (Galitz, 2007). The results of these user interviews are discussed in the evaluation section 5.3.

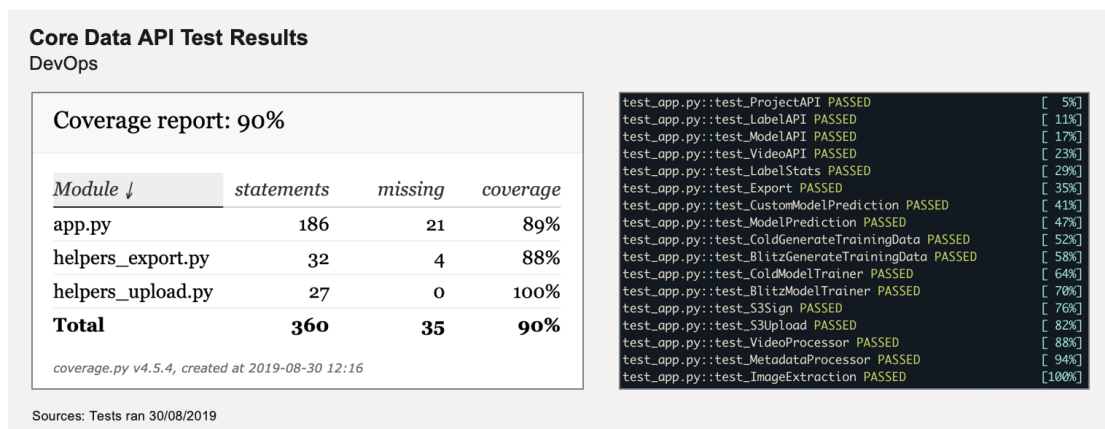


Figure 4.26: Core Data API Test Coverage and Results

To test the Core Data API I used the `pytest.py` and `coverage.py` libraries. All routes and helper functions were tested and the results are presented in Figure 4.26. The tests perform well and cover a high percentage of the code across the Core Data API. Although an extensive continuous integration suite could have been set up, it was decided early on that focus should be on developing an extensive prototype. The nature of prototypes state that they "should be capable of being rapidly changed" (Galitz, 2007). For that reason a CI pipeline would have slowed development signif-

icantly, since the architecture of the application developed over the three months of development.

4.4.2 Deployment

The objective of good deployment is to make it as simple as possible for users to install the AIGVA tool on their own system. For that reason, the AIGVA tool is deployed using the Docker container service. A container is a "special type of virtual machine that run on top of the kernel of the host operating system" and because "virtualization stops at the kernel, containers are much more lightweight and efficient than virtual machines" (Grinberg, 2014). The core idea behind Docker is to make it possible for anyone to run an instance of your container without having to configure a plethora of packages and drivers.

Listing 4.10: Docker-compose.yml Configuration File (Located in: Backend)

```
1 version: '3'
2 services:
3     flask:
4         privileged: true
5         build:
6             context: .
7             dockerfile: Dockerfile
8         command: ./scripts/run_server.sh
9         ports:
10            - '5000:5000'
11        links:
12            - redis
13            - celery
14        volumes:
15            - ./app
16
17    celery:
18        privileged: true
19        build:
20            context: .
```

```
21         dockerfile: Dockerfile
22         command: ./scripts/run_celery.sh
23         links:
24             - redis
25         volumes:
26             - .:/app
27
28     redis:
29         image: redis:latest
30         hostname: redis
```

To containerize the AIGVA tool two separate Docker configuration files were created: a Dockerfile and a docker-compose. First, the Dockerfile (see Appendix D.3.1) is responsible for building up a default Ubuntu virtual machine to handle running the AIGVA tool. This means various setups such as: installing the right Python versions, running the "pip installs" needed to run the Flask server and Celery tools, compiling the C dependencies needed to run OpenCV, configuring Linux user privileges using the `chmod` and `add-user` commands, and building the Redis server. Second, once the Ubuntu VM was set up, a docker-compose file was needed to run all the servers required to run the AIGVA tool (see Listing 4.10). Recall that the Core Data API consists of three key components: the Flask server, the Celery workers, and the Redis message broker. Each of these three components require their own server to run on. For that reason, the docker-compose tool was utilized to start all three servers and link them to each other using a net of dependencies.

Chapter 5

Evaluation & Results

In this section I will evaluate the AIGVA tool and assess the extent to which it achieved the objectives of the project. I will assess the tool both quantitatively and qualitatively in several ways. First, I empirically test the AIGVA tool to quantitatively show the impact and usefulness of the tool. Second, I quantitatively conduct rigorous performance testing of the application to understand both the capabilities and limitations of the tool. Third, I conduct a user survey with key stakeholders and potential users of the project to qualitatively assess its usefulness.

5.1 Quantitative Evaluation

To quantitatively test the usefulness of the AIGVA tool I have conducted two experiments to investigate the performance and efficiency of several labelling techniques.

5.1.1 Label Prediction - Cold Label

The first experiment I conducted was on the Cold Label prediction technique. The core idea behind the experiment was to label frames of certain class instances in a set of videos and then subsequently use those extracted frames to train a classifier and test its performance. The hypothesis underlying the experiment was that more accurate labelling methods would produce better classifiers on unseen test data.

Methodology

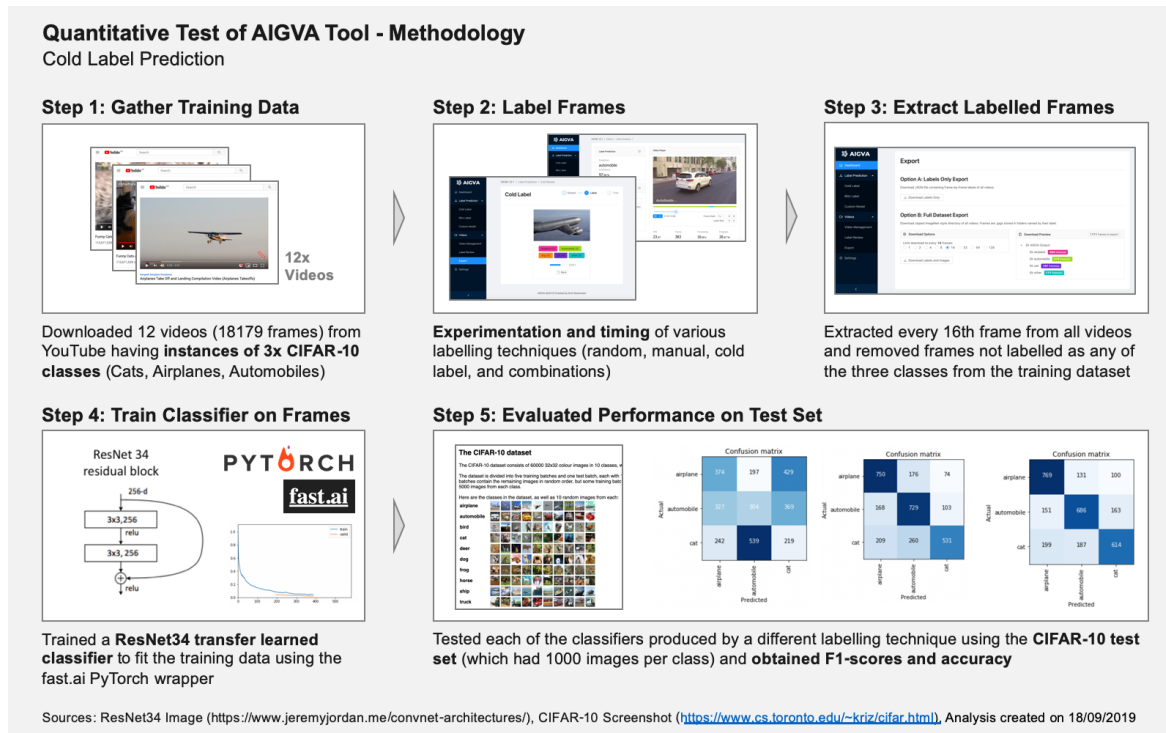


Figure 5.1: Quantitative Test of AIGVA Cold Label - Methodology

The experiment consisted of five steps as shown in Figure 5.1 First, I downloaded 12 YouTube videos which had instances of three categories from the CIFAR-10 dataset (airplane, automobiles and cats) - see Appendix C.3.1 for the full list of the videos. Second, I used a variety of techniques to label the images as seen in the list below. Each labelling technique was carefully timed from beginning to end and split into three buckets where applicable: (1) "labelling time" was time spent manually labelling any number of images, (2) "prediction time" was time spent running predictive algorithms to automatically label videos, (3) "correction time" was time spent correcting predicted labels if that was part of the experiment. Third, frames were extracted using the AIGVA export tool - here only a 16th of the images were extracted so as not to overfit the training data as it was made up of only 12 videos. Fourth, an image classification model was trained on the output of each labelling technique using the fast.ai PyTorch framework. The classifier used was a ResNet34 model pre-trained on ImageNet with the main layers frozen so it was possible to

fine-tune it to the training data (see Appendix C.3.2 for full model specifications). Finally, I tested each of the classification models on the CIFAR-10 test set data for each of the three classes. I obtained the F1-score, accuracy and confusion matrix for each of the labelling techniques. Furthermore, to keep the training consistent, I trained each model three times for the same number of epochs and averaged the accuracy and F1-scores to prevent any major outliers from affecting the results.

Labelling Techniques Used:

1. Randomized: randomly assigning each frame to a category to provide a baseline
2. Manual #1 (Frame-by-frame): hand-labelling every single frame
3. Manual #2 (Using AIGVA tool): hand-labelling using the AIGVA label review video player
4. Cold Label #1 (5 images / video): using the Cold Label tool to manually label 5 images for each of the 12 videos, then training a classifier on those images and using that classifier to predict all frames
5. Cold Label #2 (10 images / video): using the Cold Label tool to manually label 10 images for each of the 12 videos, then training a classifier on those images and using that classifier to predict all frames
6. Cold Label #3 (w/ Unique Image Extractor): using the Cold Label tool with the unique image extractor to label all the "unique" frames per video, then training a classifier on those images and using that classifier to predict all frames
7. Cold Label #1 + Correction: using the previous method, but then spending time to correct for mistakes
8. Cold Label #2 + Correction: using the previous method, but then spending time to correct for mistakes

9. Cold Label #3 + Correction: using the previous method, but then spending time to correct for mistakes

Results & Disucssion

Overall, the results show that the AIGVA tool can significantly speed up the process of frame-by-frame labelling by as much as 14-19x. The results of the experiment are shown in Table 5.1 and confusion matrices for each of the experiments can be found in Appendix C.3.3. The first observation to note is that the randomized label approach performs as expected, with an accuracy of 30%, meaning that it is just as good as random guessing. This provides a good baseline and sanity check for our test.

Labelling Technique	Time Spent (Mins)				Model Performance	
	<i>Labelling</i>	<i>Predicting</i>	<i>Correcting</i>	<i>Total</i>	<i>Accuracy</i>	<i>F1-Score</i>
Randomized	0	0	0	0	0.30	0.35
Manual #1 (Frame-by-frame)	303	0	0	303	0.67	0.66
Manual #2 (Using AIGVA tool)	28	0	0	28	0.67	0.66
Cold Label #1 (5 images / video)	1	11	0	12	0.63	0.61
Cold Label #2 (10 images / video)	2	11	0	13	0.64	0.65
Cold Label #3 (w/ Unique Img.)	5	11	0	16	0.65	0.65
Cold Label #1 + Correction	1	11	10	22	0.67	0.66
Cold Label #2 + Correction	2	11	8	21	0.67	0.66
Cold Label #3 + Correction	5	11	5	21	0.67	0.66

Table 5.1: AIGVA Cold Label Results

The second observation to note is the performance of the two manual labelling techniques, Manual #1 and Manual #2. We observe a significant performance improvement of the manually labelled techniques to the randomized approach, with the classification accuracy jumping to 67%. This confirms that there is explanatory power in our training data, despite the training data just being sourced from 12 YouTube videos. Comparing the two manual approaches we observe a significant time saving of 11x from the 303mins¹ taken to label the 18179 frames using the Manual #1

¹Manual #1 time spent is an estimation assuming it takes 1 second to label a frame and since 18179 frames needed to be labelled. Thus, data for Manual #1 is the same as data for Manual #2 since both approaches are human labelled and will produce the same result.

technique to the 28mins taken using the Manual #2 technique. This suggests that using the AIGVA tool without the label prediction features can significantly speed up the labelling process.

The third observation to note is the impact of using the three Cold Label techniques (without corrections). We see the test accuracy drop from 67% using the manual techniques down to 63%-65% using the three Cold Label techniques. However, the Cold Label techniques are also faster than the manual techniques. For example, if you are willing to sacrifice a 2% performance loss (from 67% to 65%), total labelling time will be reduced to only 16mins. This is 19x faster than having to hand-label every frame. Furthermore, as expected we see a positive relationship between the amount of time taken to label a subset of images and the subsequent performance of the classifiers. In the best performing Cold Label technique, Cold Label #3, the "Unique Image Extraction" algorithm extracts approximately 3 times the amount of images for the user to label compared to Cold Label #2². This suggests that the user of the AIGVA tool must carefully balance the trade-off between time spent labelling and the "correctness" of the labels required. However, as we will explore in the final observation, this trade-off becomes less important if the user decides to manually validate the labels after they have been predicted using the Cold Label techniques.

The final observation to note, is the performance increase you obtain when using any of the three Cold Label techniques and subsequently correcting the mistakes made manually. It took between 5-10mins to correct the labels for each of the Cold Label techniques. This was of course dependent on the correctness of the labels. Interestingly, once the corrections were factored in and all labels were correct, the total time taken did not vary by more than a minute between the Cold Label options (21-22mins). Hence the "time-spent labelling versus accuracy" trade-off discussed in the previous becomes less relevant. Using either of the Cold Label techniques to obtain initial "weak" labels and then manually adjusting them proved to be the

²Manually labelled 55 images for Cold Label #1, 110 images for Cold Label #2, 335 images for Cold Label #3

fastest approaches obtain a complete set of correct labels. This is because a user spends much less time manually correcting a video once it has some labels, assuming the labels are mostly correct. Overall, using this combination of Cold Labelling and correcting cut the time taken to label all videos correctly by 14x compared to the manual frame-by-frame approach.

Limitations

It is worth noting the limitations of this experiment. Because of the laborious and time-consuming nature of the task, the experiment was constrained to use only 12 YouTube videos as training data. These videos were relatively simplistic in nature, with relatively few class changes within each video. Furthermore, the image classification task was limited to only three classes of moderate visual difference: cats, airplanes and automobiles. Despite these choices to limit the experiment, the total time to conduct the experiment was still a full day. All of these limitations mean that we have to be skeptical when generalizing these results to both larger datasets and more difficult domains. However, in my opinion, I think that these results offer an indication at the usefulness of the tool.

There are also several limitations to the Cold Label technique overall. First, despite the advances in few-shot learning and transfer learning, there is still a minimum threshold of data required for a classifier to perform well. This threshold varies from domain to domain. For instances, only few examples may be needed to distinguish between a cat and a dog, whereas more instances are needed to distinguish between different breeds of dogs. Second, the complexity of the model used may be insufficient to capture the difference between classes. Some domains such as detecting scan planes in fetal ultrasounds are hard, and therefore are likely too difficult for this type of label prediction. Third, if classes are very imbalanced, sampling a few images from each video may skip key frames of underrepresented classes.

5.1.2 Label Prediction - Blitz Label

To quantitatively test the Blitz Label technique I conducted an experiment to investigate the performance of several labelling approaches. The idea behind the experiment was to time several labelling approaches and then examine how many of the frames they predicted correctly.

Methodology

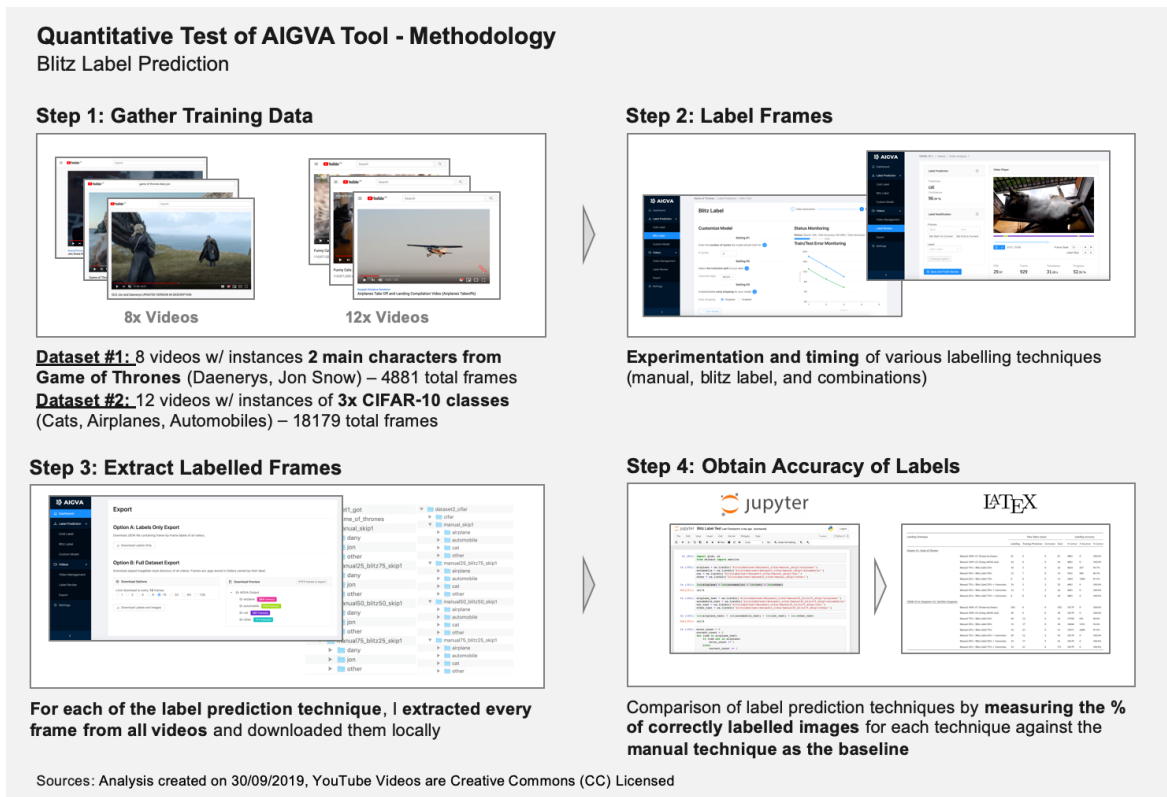


Figure 5.2: Quantitative Test of AIGVA Blitz Label - Methodology

The experiment consisted of four steps as shown in Figure 5.2. First, I prepared two datasets: (1) a dataset consisting of clips from the TV show *Game of Thrones* with instances of the two main characters, and (2) the dataset used in the previous experiment of YouTube videos which had instances of three categories from the CIFAR-10 dataset. I chose to conduct the experiment on these two datasets to examine the difference in performance across different video domains.

Datasets Used:

1. Dataset #1: 8 videos w/ instances of 2 main characters from Game of Thrones (Daenerys, Jon Snow) - 4881 total frames
2. Dataset #2: 12 videos w/ instances of 3 CIFAR-10 classes (Cats, Airplanes, Automobiles) - 18179 total frames

Second, I used a variety of techniques to label the images as seen in the list below. Each labelling technique was carefully timed from beginning to end and split into three buckets where applicable: (1) "labelling time" was time spent manually labelling any number of images, (2) "prediction time" was time spent running predictive algorithms to automatically label videos, (3) "correction time" was time spent correcting predicted labels if that was part of the experiment. Third, all frames were extracted using the AIGVA export tool and downloaded locally. Fourth, the downloaded frames were processed using a Jupyter notebook. Here I compared the performance each of the labelling techniques by determining the number of frames that were correctly and incorrectly labelled. I was able to calculate this because I initially labelled all of the frames manually as part of the experiment.

Labelling Techniques Used:

1. Manual 1 (Frame-by-frame): hand-labelling every single frame
2. Manual 2 (Using AIGVA tool): hand-labelling using the AIGVA label review video player
3. Manual 75% / Blitz Label 25%: manually labelling 75% of the data and training Blitz Label model on it, and then using the model to predict frames for the remaining 25%
4. Manual 50% / Blitz Label 50%: manually labelling 50% of the data and training Blitz Label model on it, and then using the model to predict frames for the remaining 50%

5. Manual 25% / Blitz Label 75%: manually labelling 25% of the data and training Blitz Label model on it, and then using the model to predict frames for the remaining 75%
6. Manual 75% / Blitz Label 25% + Correction: using the previous method, but then spending time to correct for mistakes
7. Manual 50% / Blitz Label 50% + Correction: using the previous method, but then spending time to correct for mistakes
8. Manual 25% / Blitz Label 75% + Correction: using the previous method, but then spending time to correct for mistakes

Results & Disucssion

Overall, the results show that the AIGVA tool using Blitz Label can significantly speed up the process of frame-by-frame labelling by as much as 8x. The results of the experiment are shown in Table 5.2. The first observation to note is the performance of the two manual labelling techniques, Manual #1 and Manual #2. For dataset 1, using the AIGVA tool without the label prediction features reduced the labelling time from 81min³ to 24min - a 3.3x performance improvement. For dataset 2, using the AIGVA tool without the label prediction features reduced the labelling time from 303min to 38min - a 8.0x performance improvement. The difference between datasets can be explained by the difference in dataset complexity. Dataset 2 had larger groupings of neighbouring labels (i.e. fewer label changes per video) which meant that using the AIGVA tool, where you can quickly label adjacent frames, was more efficient for this dataset. Overall, the speedup up 3.3-8.0x shows that even the AIGVA tool without the label prediction features can significantly speed up the labelling process.

³Manual #1 time spent is an estimation assuming it takes 1 second to label a frame and since either 4881 or 18179 frames needed to be labelled. Thus, data for Manual #1 is the same as data for Manual #2 since both approaches are human labelled and will produce the same result.

Labelling Technique	Time Taken (mins)				Labelling Accuracy		
	Labelling	Training/Prediction	Correction	Total	# Correct	# Incorrect	% Correct
Dataset #1: Game of Thrones							
Manual 100% #1 (Frame-by-frame)	81	0	0	81	4881	0	100.0%
Manual 100% #2 (Using AIGVA tool)	24	0	0	24	4881	0	100.0%
Manual 75% / Blitz Label 25%	18	5	0	23	4624	257	94.7%
Manual 50% / Blitz Label 50%	12	7	0	19	3921	960	80.3%
Manual 25% / Blitz Label 75%	6	8	0	14	3293	1588	67.5%
Manual 75% / Blitz Label 25% + Correction	18	5	2	25	4881	0	100.0%
Manual 50% / Blitz Label 50% + Correction	12	7	3	22	4881	0	100.0%
Manual 25% / Blitz Label 75% + Correction	6	8	6	20	4881	0	100.0%
CIFAR-10 3x Categories #2: YouTube Categories							
Manual 100% #1 (Frame-by-frame)	303	0	0	303	18179	0	100.0%
Manual 100% #2 (Using AIGVA tool)	38	0	0	38	18179	0	100.0%
Manual 75% / Blitz Label 25%	29	12	0	41	17558	621	96.6%
Manual 50% / Blitz Label 50%	19	17	0	36	16826	1353	92.6%
Manual 25% / Blitz Label 75%	10	21	0	31	15971	2208	87.9%
Manual 75% / Blitz Label 25% + Correction	29	12	2	43	18179	0	100.0%
Manual 50% / Blitz Label 50% + Correction	19	17	5	41	18179	0	100.0%
Manual 25% / Blitz Label 75% + Correction	10	21	6	37	18179	0	100.0%

Table 5.2: AIGVA Blitz Label Results

The second observation to note is the impact of using Blitz Label (without corrections). For both datasets examined there is a clear positive linear relationship between the amount of data you label manually and the percentage of labels that are correct. This is unsurprising as more training data generally produces more accurate models. For both datasets, the total time taken to label using any of the Blitz Label techniques is not much less than manually labelling all the data using the AIGVA tool - although this is still significantly faster than manually labelling videos frame-by-frame. For example, for dataset 1, using a 50/50 Manual/Blitz Label split reduces the time taken to label the data from 24min to 19 min but then mislabels ca. 20% of the data. Similarly, for dataset 2, using a 50/50 Manual/Blitz Label split reduces the time taken to label the data from 38 min to 36min but then mislabels ca. 7% of the data. Although these results do not appear too promising for the Blitz Label technique, it is worth noting that we have measured the total time taken, not the human time spent. For example, for dataset 2 using a 25/75 Manual/Blitz Label split reduces the human time taken to label the data from 38 min to 10min but then mislabels ca. 12% of the data.

The third observation to note is the impact of using Blitz Label with corrections. This works by running the Blitz Label model and then manually correcting the misplaced labels. These results bring the classification accuracy up to 100% in all cases

because they were manually corrected. This method is the fastest across all tested approaches. For example, for dataset 2 using a 25/75 Manual/Blitz Label split reduces the time taken from 303min using Manual #1 to only 37min, of which only 16min was human time. This represents a performance improvement of over 8.0x. These results are similar to those found in the previous Cold Label experiment where using a combination of Cold Label and corrections proved to be the fastest method. There is also room for significant improvements in training/prediction speed by for example either optimizing the Core Data API or running the server on a GPU.

Limitations

There are several limitations to this experiment. For similar reasons to the previous experiment, this experiment used limited toy datasets of either 8 or 12 videos. This is because of the laborious and time-consuming nature of the manual labelling tasks involved. Furthermore, the image classification task was constrained to only simple classes of only moderate visual difference: 2x TV show characters in one dataset, and 3x CIFAR-10 classes (cats, airplanes and automobiles) in the other. Again, despite these limitations, running the experiment took more than a full day. For the aforementioned reasons, we have to be careful when generalizing these results to both larger datasets and more difficult domains.

The "Blitz Label" method itself may have several shortcomings. First, the method may be prone to overfitting, as the extracted frames have temporal locality, although this may also be a benefit as the jittering of the videos can provide natural data augmentation (as discussed in Section 2.1.5). Second, if there is meaningful variation between the labelled videos used to train the classifier and the unlabelled videos that the classifier is used on, then the method may not generalize well. Third, similar to the potential drawbacks of the "Cold Label" technique, the complexity of the model used may be insufficient to capture the difference between classes.

5.2 Performance Testing

5.2.1 Web Application Performance

To test the performance and accessibility of the AIGVA application I use the open-source Lighthouse tool, which runs various automated audits and computes performance metrics for web applications. Modern web development is complicated since so many tools are used. In production, all of these development choices impact the user experience. Some developers use a checklist of various questions such as "how quickly does the first part of the DOM render?" and "what is the estimated input latency?". However, this approach is cumbersome. Instead most web developers recommend using a tool to automatically test your site against a long list of pre-built best practice "questions" (Sheppard, 2017). That is exactly what Lighthouse does. The Lighthouse tool runs a website against 41 different metrics and ultimately produces two scores: a performance score and an accessibility score.

The performance score (Lig, 2019) compares the speed of your website against others. The accessibility score is a weighted average of numerous accessibility measures such as the variation of CSS styles and the presence of "alt text" for images. Scores for both measurements are between 0 and 100. 100 is the best possible score which corresponds to the 98th percentile of all websites and a score of 50 represents the 75th percentile of all websites. See Appendix C.2 for a breakdown of the score weightings.

I ran the Lighthouse performance and accessibility tests twice for the 15 main pages of the AIGVA tool. The first time I ran the tests over WiFi to simulate the performance to the average user. The second time I ran the tests over an emulated 3G network, which significantly throttled the internet speed. The purpose of the second test was to see if the AIGVA tool relied too heavily on the high network speeds it was developed on to deliver its contents. All the results are shown in Figure 5.3

AIGVA Browser Performance & Accessibility Testing

Scores across web app sections

Score scale:

90-100

50-89

0-49

Section	Name	URL	WiFi Score ¹		3G Score ²	
			Performance	Accessibility	Performance	Accessibility
Project Management						
	Dashboard	/dashboard	100%	92%	93%	92%
	Add Project	/dashboard/add_project	100%	76%	98%	76%
	Edit Project	/dashboard/edit_project	100%	76%	98%	76%
Videos						
	Video Management	/videos/video_management	100%	94%	99%	94%
	Label Review	/videos/label_review	100%	94%	98%	94%
	Video Analysis	/videos/video_analysis	100%	75%	99%	75%
	Export	/videos/export	100%	93%	87%	93%
Label Prediction						
	Cold Label Home	/label_prediction/cold_predict	100%	94%	98%	94%
	Cold Label #1 (Extract)	/label_prediction/cold_create	100%	85%	98%	85%
	Cold Label #2 (Label)	/label_prediction/cold_review	100%	94%	99%	94%
	Cold Label #3 (Train)	/label_prediction/cold_train	100%	94%	98%	94%
	Blitz Label Home	/label_prediction/blitz_predict	100%	92%	92%	92%
	Blitz Label #1 (Extract)	/label_prediction/blitz_create	100%	85%	99%	85%
	Blitz Label #2 (Train)	/label_prediction/blitz_train	100%	94%	98%	94%
	Custom Model Home	/label_prediction/custom_model	100%	94%	98%	94%

Sources: ¹Imperial Library WiFi @ 216 Mbps, ²Applied 3G (4x CPU Slowdown),
Analysis created on 12/09/2019 using Google Lighthouse <https://developers.google.com/web/tools/lighthouse/>

Figure 5.3: AIGVA Performance and Accessibility Testing (Using Lighthouse)

The results of the performance and accessibility tests are good overall. There is no evidence that any part of the web application performs poorly. On the contrary, all performance scores when the application is run on WiFi are a perfect 100%, meaning that the all pages of the AIGVA tool load extremely fast when used over a good internet connection. This also suggests that there are not any inefficient event listeners in the DOM or memory leaks in the application. Similarly, the performance scores for the 3G connections are also very high, with the exception of one 87% score for the export page. I examined the performance of the export page further to try to understand the root cause of the slightly slower page time. Using a the built-in performance stack trace DevTool in Google Chrome, I could see all the DOM events occurring until the entire export page had loaded (see Figure C.1). I found that the performance penalty came from the label preview folder structure as it needed to make several API calls to estimate the number of labels per category. As

the performance penalty is still small enough to produce a score above 80% for the export page, I will not remove this feature. But one future extension could be to add the option to toggle the preview on/off in the settings page.

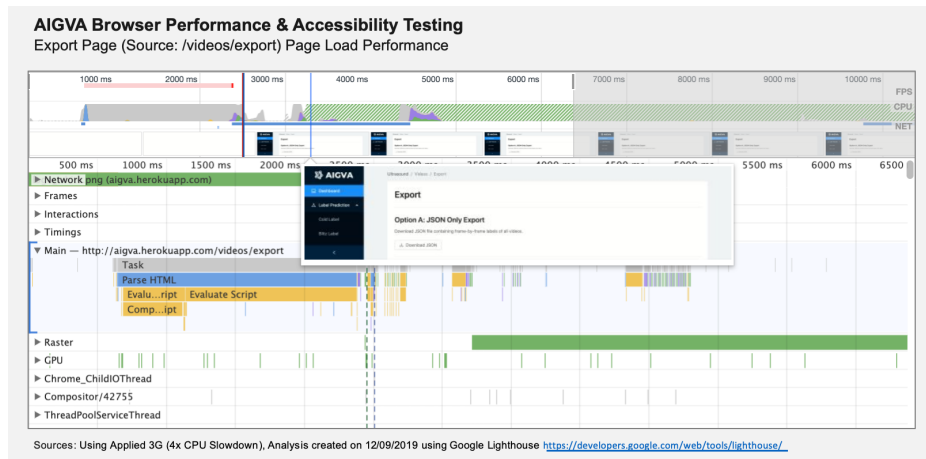


Figure 5.4: AIGVA Performance and Accessibility Testing - Export Page Analysis

The accessibility scores of the tests are equivalent for both WiFi and 3G connections. Overall, the accessibility scores are good, although there are number of scores below 90%. These lower scores all stem from the same cause: lacking "Accessible Rich Internet Applications" (ARIA) features on pages with any form of user input (such as dropdowns and input text fields). These ARIA features can "enhance the experience for users of assistive technology, like a screen reader" (Lig, 2019). However, since the usage scope of the AIGVA tool is limited to be used in a standard web browser, I have chosen not to address these ARIA features.

5.2.2 Core Data API Performance

There are several data intensive processes that run as part of the Core Data API. These include exports, label predictions and machine learning model training. To test the performance of these tools I have conducted several experiments measuring the time to run these jobs. All tests were done on a MacBook Pro (13-inch, 2016) with 2.9 GHz Intel Core i5.

Export Tool

The average export time depends on the number of frames needed to export. Therefore the export time can be measured in frames per second. The average export times for both methods are listed below:

- Export Option A (Labels Only): ~9900 FPS
- Export Option B (Full Dataset Download): ~55 FPS

Option A is clearly the fastest export method. This is by design as it only downloads the labels which are already kept in browser memory. Therefore the only action performed by the tool is to trigger the download. Option B downloads at 55FPS. This method is slower by design as it provides both labels and images organized in a zipped file. Recall, the full dataset download option triggers an API call to the Core Data API which then iterates over the videos to be exported and extracts a frames as images and places them in folders corresponding to their labels. Then the Core Data API uploads this folder as a zipped file to S3 before sending the link to the user.

Label Prediction Performance

The label prediction framerate for a single video is approximately 8 FPS. However, the AIGVA tool allows for concurrent predictions, which run as separate processes on a CPU. This pseudo-concurrency can bring the effective framerate up to nearly 11 FPS when running 4 concurrent predictions as shown in Table 5.3. Because of time constraints, I did not try deploying the Core Data API on a GPU. This could hypothetically lead to significant performance improvements. One researcher who also used a ResNet34 model saw improvements of 91x when running inference, as the time of a forward pass was reduced from 1530.01ms on a CPU to 16.71ms on a GTX 1080 Ti GPU ⁴. However, such claims would have to be tested as other factors such as frame extraction speed could prove to be bottlenecks.

⁴Data from <https://github.com/jcjohnson/cnn-benchmarks>

# Concurrent Predictions	Effective FPS
1	8.27
2	9.97
3	10.72
4	10.86

Table 5.3: Relationship Between Number of Concurrent Label Predictions and Prediction Framerate

Note that Effective FPS in this context refers to:

$$EffectiveFPS = \sum_{i=0}^j \frac{frames_j}{t} \quad (5.1)$$

Where j is the number of concurrent prediction jobs, $frames_j$ is the total frames predicted for job j in the time window t measured in seconds.

Model Training Performance

The time taken for a Blitz Label or Cold Label model to train is a factor of the number of training images used. To examine this relationship between the time taken for the model to train and the number of input images, I varied the number of input images exponentially and measured the time taken to train a model. For this experiment I trained a ResNet model using Blitz Label (note that Cold Label uses exactly the same model architecture and these results therefore also apply to that technique). I trained the Blitz Label model for three cycles which is usually sufficient to minimize the test set error. The results are shown in Table 5.4.

# Training Images	Model Training Time (secs)	Seconds / Image
111	58	0.52
222	118	0.53
445	247	0.56
888	404	0.45

Table 5.4: Relationship Between Training Images and Model Training Time (Using a ResNet Blitz Label Model Trained for 3 Cycles)

The model training performance scales almost linearly with the number of input

images. The results of experiment show that it takes 0.4-0.6 seconds per image to train a model for three cycles with the tested ranges of training images. There appears to be a convex relationship between the training images and the seconds / image to train a model. I suspect this is caused by other non-test processes disturbing the test results. Rather the takeaway from this experiment is that training time appears to be reasonable stable in the range 0.4-0.6 seconds per image.

5.3 User Interview & Feedback

To qualitatively assess the AIGVA tool, I conducted two rounds of interviews. First, I conducted four semi-structured interviews with machine learning researchers. Second, I conducted a single open-ended interview with a Clinical Sonographer that worked as part of the iFIND ultrasound project.

5.3.1 Machine Learning Researchers



Figure 5.5: AIGVA ML Researchers User Interviews: Selected Quotes

After giving four machine learning researchers from Imperial College London a demo of the tool, I asked several questions as part of a semi-structured survey - see Appendix B.1 for full survey transcripts. The first question asks researchers to rate their agreement with the statement "The AIGVA tool is useful for someone having to label video data" from Highly Disagree to Highly Agree. Figure 5.6 shows the results. Overall, the results are positive and all researchers agree that the tool is useful. Additional comments from users, as seen in Figure 5.5, show that reasons researchers find it useful include: the broad accessibility of the tool, the domain agnostic nature of the tool, and the ability to use it immediately in its current state.

The second question asks researcher to rate their agreement with the statement "AIGVA is simple to understand and easy to use" from Highly Disagree to Highly

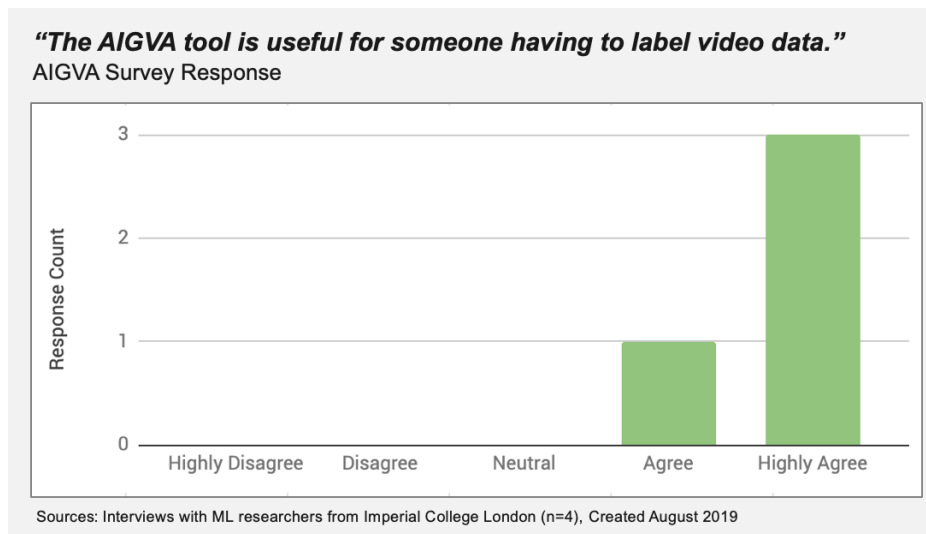


Figure 5.6: AIGVA ML Researcher Survey - Question 1 (n=4)

Agree. Figure 5.7 shows the results. All researchers highly agreed with the statement, which suggests that the tool accomplished its goals of being simple and easy to use. Further comments suggest that the reason for the positive feedback is because the researchers like the user interface and found it intuitive to use.

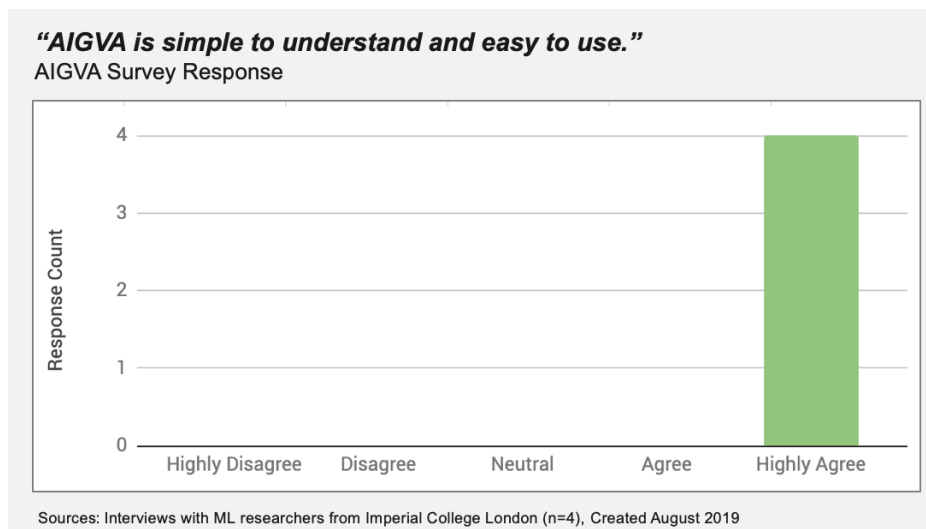


Figure 5.7: AIGVA ML Researcher Survey - Question 2 (n=4)

Additional questions in the survey dealt with any feedback, constructive criticism and feature requests the machine learning researchers had. As there were several commonalities across answers I have collated them in the following list of feature requests:

- Ability to do other types of labelling (i.e. segmentation, bounding boxes and pose estimation).
- Ability to use other types of data (i.e. image-level data, 3D volumetric data, and specific medical data such as fMRIs).
- Providing some form of label ownership to know which user or model produced the label. One researcher wanted this to discard poor labels in case he discovered that one of his labelers was performing poorly.
- GPU enabled prediction would be ideal to speed up labelling.
- A confidence bar for labels in the video analysis page that, similar to the label preview bar, would display a heat-map of the confidences across the time dimension. The idea being that you could quickly see regions where a model lacked confidence to focus review efforts there.
- A feature that could average/aggregate labels from several reviewers using the share feature. Then include some visual feature to capture label variability and display it.

5.3.2 Clinical Sonographer

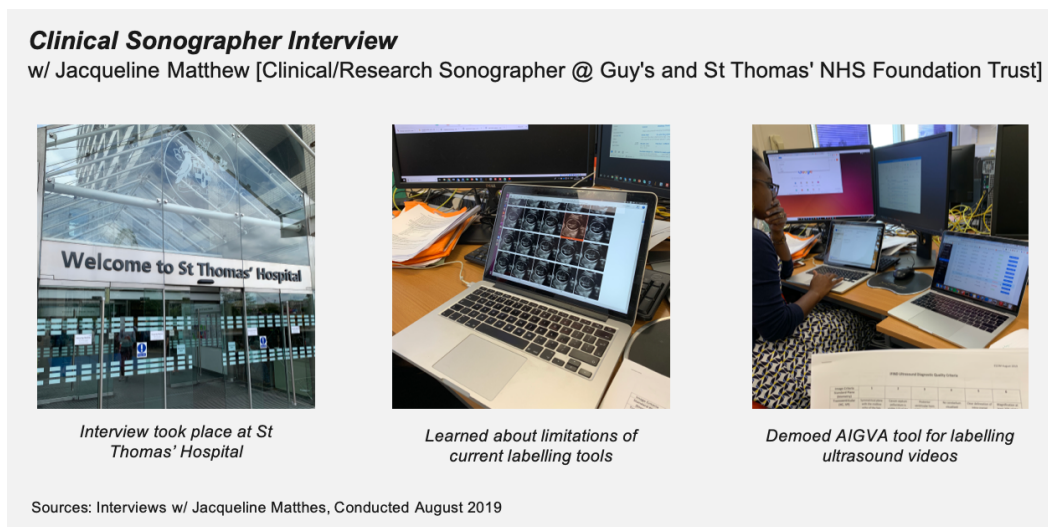


Figure 5.8: AIGVA Clinical Sonographer Interview

I was offered the opportunity to interview a clinical sonographer tool who works as part of the iFIND ultrasound project. Because the tool was still in development at the time of the interview, I used the opportunity to better understand the problems faced with current tools and therefore opted for an open-ended interview. I also presented an early prototype of the AIGVA tool to receive feedback. See Appendix B.2 for a full transcript.

After demoing an early prototype of the tool, we discussed potential use cases for sonographers and identified three key uses. First, the AIGVA tool could be useful for improving current labelling algorithms that are performing poorly. The poor performance of some current algorithms that attempt to label data is caused by the domain shift that often happens: either from (1) different sonographers labelling data or (2) different ultrasound imaging systems used than the ones the model was originally trained on. Second, the sonographer identified the opportunity to use the AIGVA tool to not only label ultrasound videos for the use of training machine learning algorithms but to catalogue ultrasound videos. Specifically, she explained how current videos are not saved and identified how videos could be stored on the AIGVA tool and automatically processed for standard planes. Having such a catalogue that could be very useful to keep an easily searchable database for both clinical and legal reasons. Third, the sonographer believed that the AIGVA tool could be useful for training sonographers. Newly trained sonographers could view different organs/regions across the length of the video using the label preview bar.

Chapter 6

Conclusion & Future Work

6.1 Summary of Achievements

This dissertation reviewed the literature on image classification and demonstrated that more labelled training data is an important factor in improving classification performance. The literature review found that videos could be an important source of such training data and subsequently examined the current landscape of video annotation tools available. The examination of existing tools found that all were lacking in label prediction and video playback features and that there was a clear opportunity to develop such a smart video annotation tool. To further motivate the development of a smart video annotation tool, the successes and challenges faced by the iFIND ultrasound research project were discussed.

I proceeded by building the foundation for a smart video annotation tool called AIGVA. I developed a set of novel label prediction techniques called Cold Label, Blitz Label and Custom Model Label. Then I presented a set of user stories to show how these techniques can be used as part of a web application to speed up the video annotation process. Then I crafted a set of design principles and functional requirements that would guide the development of this tool.

With the design principles and functional requirements defined, I presented the im-

plementation the AIGVA tool. I discuss the software architecture and technology stack that were selected and the trade-offs involved in the decision making process. The architecture was divided into three components: (1) the Core Data API responsible for video processing, machine learning and other data related tasks, (2) the Web Application responsible for providing the UI and browser tool that a user can interact with, and (3) the File Storage and Database layer responsible for hosting all video data and application related data. I proceed to discuss two of the most challenging feature implementations: the video management and label prediction features. For the video management tool I discussed how I built both an export tool and a frame-by-frame video player from scratch to enable frame-level labelling of videos via a web application. For the label prediction feature I presented how the label prediction engine was built to handle the heavy data processing created by the Cold, Blitz and Custom model labelling techniques. I discussed the implementation of the three labelling techniques and the challenges of their building such as the need to develop an algorithm to extract unique images from videos using perceptual hashing. I also demonstrate how the application has been tested and then deployed using Docker.

Finally I evaluated the tool using both quantitative and qualitative measures. I conducted an empirical study on the label prediction methods which suggested that the AIGVA tool could cut the time used to label videos by a factor of nearly 14-19x. I assessed the real-time performance of the AIGVA web app and the Core Data API using a variety of techniques that all suggest that the tool performs well. Finally I qualitatively assessed the tool by conducting a set of user interviews with machine learning researchers and clinical sonographers. Both rounds of interviews suggested that the tool could be useful in practical real-world settings with either minor or no modifications.

6.2 Future Work

There are multiple areas where the AIGVA tool could be extended to address its limitations. The application could be improved by adding user management features such as admin roles and reviewer roles. This would make it possible to develop label quality assessment tools where an admin can aggregate the labels from several reviewers. Furthermore, the tool is limited to image-level classification, meaning that only a single label can be associated with an image. Additional labelling techniques such as image segmentation, object detection and pose estimation could also be integrated to extend the usability of the tool.

Another limitation is that the tool only supports video data currently. Therefore, additional data types such as image-only data or 3D volumetric data could be supported to extend use cases further. Extending the upload tool functionality to handle multiple videos would improve the user experience as currently the upload functionality can only handle one video at a time. Another area that could benefit from further development is to improve label predictions by either: (1) using GPU inference instead of CPU inference to speed the process up, or (2) exploring new model types used for Blitz and Cold Label that could improve the classification rates and therefore increase prediction accuracy.

Overall the development of the AIGVA tool has been a rewarding and exciting experience. I hope that both technical and non-technical users will benefit from its development, which I plan to continue in the future.

Bibliography

- [1] (2019). Ant design - react ui library. <https://ant.design/docs/react/introduce>. Accessed: 2019-08-28. pages 40
- [2] (2019). Apexcharts - js charting library. <https://apexcharts.com>. Accessed: 2019-10-08. pages 67
- [3] (2019). Axios - promise based http client for the browser and node.js. <https://github.com/axios/axios>. Accessed: 2019-10-08. pages 66
- [4] (2019). Bizcharts - js charting library. <https://bizcharts.net>. Accessed: 2019-10-08. pages 67
- [5] (2019). Celery - python task queue. <https://docs.celeryproject.org/en/latest/index.html>. Accessed: 2019-08-08. pages 61
- [6] (2019). Createreactapp - officially supported way to create single-page react applications. <https://facebook.github.io/create-react-app/docs/getting-started>. Accessed: 2019-10-08. pages 65
- [7] (2019). Cvat - video annotation tool. <https://github.com/opencv/cvat>. Accessed: 2019-20-08. pages 23
- [8] (2019). Eu gdpr - what is personal data? https://ec.europa.eu/info/law/law-topic/data-protection/reform/what-personal-data_en. Accessed: 2019-28-08. pages 26
- [9] (2019). Filesaverjs - an html5 saveas() filesaver. <https://github.com/eligrey/FileSaver.js/>. Accessed: 2019-10-08. pages 68

-
- [10] (2019). Flask restless. <https://flask-restless.readthedocs.io>. Accessed: 2019-08-08. pages 59
- [11] (2019). Flask web framework. <https://github.com/pallets/flask>. Accessed: 2019-08-08. pages 58
- [12] (2019). Gunicorn - python wsgi http server for unix. <https://gunicorn.org/>. Accessed: 2019-08-08. pages 60
- [13] (2019a). Labelbox - video annotation tool. <https://github.com/Labelbox/Labelbox>. Accessed: 2019-20-08. pages 20
- [14] (2019b). Labelme - video annotation tool. <http://labelme.csail.mit.edu/Release3.0/>. Accessed: 2019-20-08. pages 21
- [15] (2019). Lighthouse web performance measurements. <https://developers.google.com/web/tools/lighthouse/>. Accessed: 2019-12-08. pages 111, 113
- [16] (2019). Moment.js - time management for js. <https://momentjs.com/>. Accessed: 2019-10-08. pages 68
- [17] (2019). Opencv - computer vision library. <https://opencv.org>. Accessed: 2019-08-08. pages 62, 83
- [18] (2019). phash - the open source perceptual hash library. <https://www.phash.org/>. Accessed: 2019-20-08. pages 83
- [19] (2019). Postgresql - the world's most advanced open source relational database. <https://www.postgresql.org/>. Accessed: 2019-10-08. pages 72
- [20] (2019). Preliminary report highway - nstb. <https://www.nts.gov/investigations/AccidentReports/Reports/HWY18MH010-prelim.pdf>. Accessed: 2019-30-08. pages 1
- [21] (2019a). React-router-dom - a dom binding of react router. <https://www.npmjs.com/package/react-router-dom>. Accessed: 2019-10-08. pages 69

- [22] (2019b). Reactjs - a javascript library for building user interfaces. <https://reactjs.org>. Accessed: 2019-10-08. pages 64
- [23] (2019). Rectlabel - video annotation tool. <https://rectlabel.com/>. Accessed: 2019-20-08. pages 21
- [24] (2019). Redis - real-time message broker and database. <https://redis.io/>. Accessed: 2019-08-08. pages 61
- [25] (2019). Vatic - video annotation tool. <http://www.cs.columbia.edu/~vondrick/vatic/>. Accessed: 2019-20-08. pages 22
- [26] (2019). Vott - video annotation tool. <https://github.com/microsoft/VoTT>. Accessed: 2019-20-08. pages 22
- [27] Baumgartner, C., Kamnitsas, K., Matthew, J., P. Fletcher, T., Smith, S., Koch, L., Kainz, B., and Rueckert, D. (2016a). Real-time detection and localisation of fetal standard scan planes in 2d freehand ultrasound. *IEEE Transactions on Medical Imaging*, 36. pages 24
- [28] Baumgartner, C. F., Kamnitsas, K., Matthew, J., Fletcher, T. P., Smith, S., Koch, L. M., Kainz, B., and Rueckert, D. (2016b). Real-time detection and localisation of fetal standard scan planes in 2d freehand ultrasound. *CoRR*, abs/1612.05601. pages 24, 25
- [29] Benyon, D. (2014). Designing interactive systems: A comprehensive guide to hci, ux and interaction design. pages 39
- [30] Cai, Y., Sharma, H., Chatelain, P., and Noble, J. (2018). *Multi-task SonoEyeNet: Detection of Fetal Standardized Planes Assisted by Generated Sonographer Attention Maps*, volume 11070, pages 871–879. pages 24
- [31] Chen, H., Ni, D., Qin, J., Li, S., Yang, X., Wang, T., and Ann Heng, P. (2015). Standard plane localization in fetal ultrasound via domain transferred deep neural networks. *IEEE journal of biomedical and health informatics*, 19. pages 24

-
- [32] Dean, J., Patterson, D., and Young, C. (2018). A new golden age in computer architecture: Empowering the machine-learning revolution. *IEEE Micro*, 38(2):21–29. pages 11
- [33] Deng, J., Dong, W., Socher, R., Li, L., Kai Li, and Li Fei-Fei (2009). Imagenet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*, pages 248–255. pages 14
- [34] Dyl, T. and Przeorski, K. (2017). *Mastering: Full-Stack React Web Development*. Packt Publishing. pages 70, 71
- [35] Fielding, R. T. (2000). *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis. AAI9980887. pages 58
- [36] Galitz, W. O. (2007). *The essential guide to user interface design: an introduction to GUI design principles and techniques*. John Wiley & Sons. pages 39, 40, 41, 42, 97
- [37] Ghahramani, Z. (2015). Probabilistic machine learning and artificial intelligence. *Nature*, 521(7553):452. pages 11
- [38] Gong, C., Tao, D., Maybank, S. J., Liu, W., Kang, G., and Yang, J. (2016). Multi-modal curriculum learning for semi-supervised image classification. *IEEE Transactions on Image Processing*, 25(7):3249–3260. pages 17
- [39] Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep Learning*. MIT Press. <http://www.deeplearningbook.org>. pages i, 12
- [40] Grinberg, M. (2014). *Flask Web Development: Developing Web Applications with Python*. O'Reilly Media, Inc., 1st edition. pages 72, 98
- [41] Guillaumin, M., Verbeek, J., and Schmid, C. (2010). Multimodal semi-supervised learning for image classification. pages 902–909. pages 17
- [42] He, K., Zhang, X., Ren, S., and Sun, J. (2015). Deep residual learning for image recognition. *CoRR*, abs/1512.03385. pages 10

- [43] Holzinger, A. (2016). Interactive machine learning for health informatics: when do we need the human-in-the-loop? *Brain Informatics*, 3(2):119–131. pages 13, 15
- [44] Huang, G., Liu, Z., and Weinberger, K. Q. (2016). Densely connected convolutional networks. *CoRR*, abs/1608.06993. pages 10
- [45] Huh, M., Agrawal, P., and Efros, A. A. (2016). What makes imagenet good for transfer learning? *CoRR*, abs/1608.08614. pages 12
- [46] Iandola, F. N., Moskewicz, M. W., Ashraf, K., Han, S., Dally, W. J., and Keutzer, K. (2016). Squeezenet: Alexnet-level accuracy with 50x fewer parameters and <1mb model size. *CoRR*, abs/1602.07360. pages 10
- [47] Jin, S., Roy Chowdhury, A., Jiang, H., Singh, A., Prasad, A., Chakraborty, D., and Learned-Miller, E. G. (2018). Unsupervised hard example mining from videos for improved object detection. *CoRR*, abs/1808.04285. pages 19
- [48] Joshi, A., Porikli, F., and Papanikolopoulos, N. (2009). Multi-class active learning for image classification. pages 2372–2379. pages 16
- [49] Karpathy, A., Toderici, G., Shetty, S., Leung, T., Sukthankar, R., and Fei-Fei, L. (2014). Large-scale video classification with convolutional neural networks. In *2014 IEEE Conference on Computer Vision and Pattern Recognition*, pages 1725–1732. pages 18
- [50] Kim, G., Debois, P., Willis, J., and Humble, J. (2016). *The DevOps Handbook: How to Create World-Class Agility, Reliability, and Security in Technology Organizations*. IT Revolution Press. pages 97
- [51] Kingma, D. P., Mohamed, S., Rezende, D. J., and Welling, M. (2014). Semi-supervised learning with deep generative models. In *Advances in neural information processing systems*, pages 3581–3589. pages 17

- [52] Kovashka, A., Vijayanarasimhan, S., and Grauman, K. (2011). Actively selecting annotations among objects and attributes. In *2011 International Conference on Computer Vision*, pages 1403–1410. pages 16
- [53] Krizhevsky, A., Hinton, G., et al. (2009). Learning multiple layers of features from tiny images. Technical report, Citeseer. pages 88
- [54] Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In Pereira, F., Burges, C. J. C., Bottou, L., and Weinberger, K. Q., editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc. pages 10
- [55] Lage, I., Ross, A., Gershman, S. J., Kim, B., and Doshi-Velez, F. (2018). Human-in-the-loop interpretability prior. In Bengio, S., Wallach, H., Larochelle, H., Grauman, K., Cesa-Bianchi, N., and Garnett, R., editors, *Advances in Neural Information Processing Systems 31*, pages 10159–10168. Curran Associates, Inc. pages 15
- [56] Ng, A. (2015). Advice for applying machine learning. <https://see.stanford.edu/materials/aimlcs229/ML-advice.pdf> [Accessed: 06-06-2019]. pages 12
- [57] NHS (2017). Ultrasound scans in pregnancy. <https://www.nhs.uk/conditions/pregnancy-and-baby/ultrasound-anomaly-baby-scans-pregnant/> [Accessed: 03-06-2019]. pages 2
- [58] Norouzi, M., Fleet, D. J., and Salakhutdinov, R. R. (2012). Hamming distance metric learning. In Pereira, F., Burges, C. J. C., Bottou, L., and Weinberger, K. Q., editors, *Advances in Neural Information Processing Systems 25*, pages 1061–1069. Curran Associates, Inc. pages 84
- [59] Perez, L. and Wang, J. (2017). The effectiveness of data augmentation in image classification using deep learning. *CoRR*, abs/1712.04621. pages 12
- [60] Real, E., Aggarwal, A., Huang, Y., and Le, Q. V. (2018). Regularized evolution for image classifier architecture search. *CoRR*, abs/1802.01548. pages 11

- [61] Research, G. (2016). Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *CoRR*, abs/1603.04467. pages 11
- [62] Roh, Y., Heo, G., and Whang, S. E. (2018). A survey on data collection for machine learning: a big data - AI integration perspective. *CoRR*, abs/1811.03402. pages 13, 14, 16, 17
- [63] Russakovsky, O., Li, L., and Fei-Fei, L. (2015). Best of both worlds: Human-machine collaboration for object annotation. In *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2121–2131. pages 15
- [64] Schlemper, J., Oktay, O., Schaap, M., Heinrich, M. P., Kainz, B., Glocker, B., and Rueckert, D. (2018a). Attention gated networks: Learning to leverage salient regions in medical images. *CoRR*, abs/1808.08114. pages 11
- [65] Schlemper, J., Oktay, O., Schaap, M., Heinrich, M. P., Kainz, B., Glocker, B., and Rueckert, D. (2018b). Attention gated networks: Learning to leverage salient regions in medical images. *CoRR*, abs/1808.08114. pages 24, 25
- [66] Sheppard, D. (2017). *Beginning Progressive Web App Development: Creating a Native App Experience on the Web*. pages 111
- [67] Shin, H., Roth, H. R., Gao, M., Lu, L., Xu, Z., Nogues, I., Yao, J., Mollura, D. J., and Summers, R. M. (2016). Deep convolutional neural networks for computer-aided detection: CNN architectures, dataset characteristics and transfer learning. *CoRR*, abs/1602.03409. pages 11
- [68] Simonyan, K. and Zisserman, A. (2014). Very deep convolutional networks for large-scale image recognition. *CoRR*, abs/1409.1556. pages 10
- [69] Sun, C., Shrivastava, A., Singh, S., and Gupta, A. (2017). Revisiting unreasonable effectiveness of data in deep learning era. *CoRR*, abs/1707.02968. pages 11, 12
- [70] Tajbakhsh, N., Shin, J. Y., Gurudu, S. R., Hurst, R. T., Kendall, C. B., Gotway, M. B., and Liang, J. (2016). Convolutional neural networks for medical image

- analysis: Full training or fine tuning? *IEEE Transactions on Medical Imaging*, 35(5):1299–1312. pages 88
- [71] Tan, M. and Le, Q. V. (2019). Efficientnet: Rethinking model scaling for convolutional neural networks. *CoRR*, abs/1905.11946. pages 10, 11
- [72] Toussaint, N., Khanal, B., Sinclair, M., Gómez, A., Skelton, E., Matthew, J., and Schnabel, J. A. (2018). Weakly supervised localisation for fetal ultrasound images. *CoRR*, abs/1808.00793. pages 24
- [73] Vinyals, O., Blundell, C., Lillicrap, T. P., Kavukcuoglu, K., and Wierstra, D. (2016). Matching networks for one shot learning. *CoRR*, abs/1606.04080. pages 17
- [74] Wang, F., Jiang, M., Qian, C., Yang, S., Li, C., Zhang, H., Wang, X., and Tang, X. (2017). Residual attention network for image classification. *CoRR*, abs/1704.06904. pages 11
- [75] Wang, Y. and Yao, Q. (2019). Few-shot learning: A survey. *CoRR*, abs/1904.05046. pages 17, 18
- [76] Yan, W., Yap, J., and Mori, G. (2015). Multi-task transfer methods to improve one-shot learning for multimedia event detection. In *Proceedings of the British Machine Vision Conference 2015, BMVC 2015, Swansea, UK, September 7-10, 2015*, pages 37.1–37.13. pages 19
- [77] Yang, Z., He, X., Gao, J., Deng, L., and Smola, A. J. (2015). Stacked attention networks for image question answering. *CoRR*, abs/1511.02274. pages 11
- [78] Yarowsky, D. (1995). Unsupervised word sense disambiguation rivaling supervised methods. In *Proceedings of the 33rd Annual Meeting on Association for Computational Linguistics, ACL '95*, pages 189–196, Stroudsburg, PA, USA. Association for Computational Linguistics. pages 17

- [79] Yu, F., Zhang, Y., Song, S., Seff, A., and Xiao, J. (2015). LSUN: construction of a large-scale image dataset using deep learning with humans in the loop. *CoRR*, abs/1506.03365. pages 15
- [80] Zang, S., Ding, M., Smith, D., Tyler, P., Rakotoarivelo, T., and Kaafar, M. A. (2019). The impact of adverse weather conditions on autonomous vehicles: How rain, snow, fog, and hail affect the performance of a self-driving car. *IEEE Vehicular Technology Magazine*, 14(2):103–111. pages 19
- [81] Zhang, Y., Li, K., Li, K., Wang, L., Zhong, B., and Fu, Y. (2018). Image super-resolution using very deep residual channel attention networks. *CoRR*, abs/1807.02758. pages 11
- [82] Zheng, S., Song, Y., Leung, T., and Goodfellow, I. J. (2016). Improving the robustness of deep neural networks via stability training. *CoRR*, abs/1604.04326. pages 18
- [83] Zhou, Z.-H., Zhan, D.-C., and Yang, Q. (2007). Semi-supervised learning with very few labeled training examples. In *Proceedings of the 22Nd National Conference on Artificial Intelligence - Volume 1, AAAI’07*, pages 675–680. AAAI Press. pages 17
- [84] Zoph, B., Vasudevan, V., Shlens, J., and Le, Q. V. (2017). Learning transferable architectures for scalable image recognition. *CoRR*, abs/1707.07012. pages 11

Appendices

Appendix A

User Guide

A.1 Installation

To install and run the AIGVA app you should start by cloning into the repo.

```
1 git clone https://github.com/Emil-Sorensen/AI-Generated-Video-  
   Anotation
```

The AIGVA app has three components that has to be setup: (1) the file storage and database, (2) the Core Data API, and (3) the web application. A lot of work has been done to ensure that total setup time is kept at a minimum (should take no more than 5-10mins).

File Storage

The file storage and database is both set up as part of the backend. First you need to create an AWS S3 bucket that you can use to store files in the cloud (i.e. videos, machine learning models etc.). Creating a file storage bucket can be done on the AWS website ¹. Once you have obtained your credentials, paste them into the backend config file shown below.

Listing A.1: File Storage Config (Located in: backend/app/config.py)

```
1 S3_KEY      =  
2 S3_SECRET   =
```

¹<https://docs.aws.amazon.com/AmazonS3/latest/dev/UsingBucket.html>

```
3 S3_BUCKET    =  
4 S3_HOST      =
```

To set up the database, you can use any PostgreSQL distribution. You can either host this locally or in the cloud. For cloud providers, I recommend the site Heroku ² for its quick setup. Once you have created a database, paste the credentials into the same file as before.

Listing A.2: Database Config (Located in: backend/app/config.py)

```
1 HOST          =  
2 DATABASE      =  
3 USER          =  
4 PORT          =  
5 PASSWORD      =  
6 URI           =
```

You have now set up all the config necessary to run the backend.

Core Data API

The Core Data API is Dockerized to make installation as simple as possible. Make sure you have Docker installed and running on your machine³.

With Docker running on your machine, run the following commands to install and run the AIGVA Core Data API.

```
1 cd backend  
2 docker-compose build  
3 docker-compose up
```

The Core Data API should now be up and running at port 5000 on your localhost.

Web Application

The web application runs on the React framework and was bootstrapped using create-react-app. Installing it and running it on your localhost can be done in a

²<https://heroku.com>

³See <https://docs.docker.com/get-started/> for help to install Docker

few commands. First you need to install the required packages by running the following:

```
1 cd frontend
2 npm install
```

After installing all the required packages you can now start the application with the following command:

```
1 npm start
```

The AIGVA web application should now launch in your default browser running on port 3000. Otherwise you can access the application on <http://localhost:3000/>

A.2 User Manual

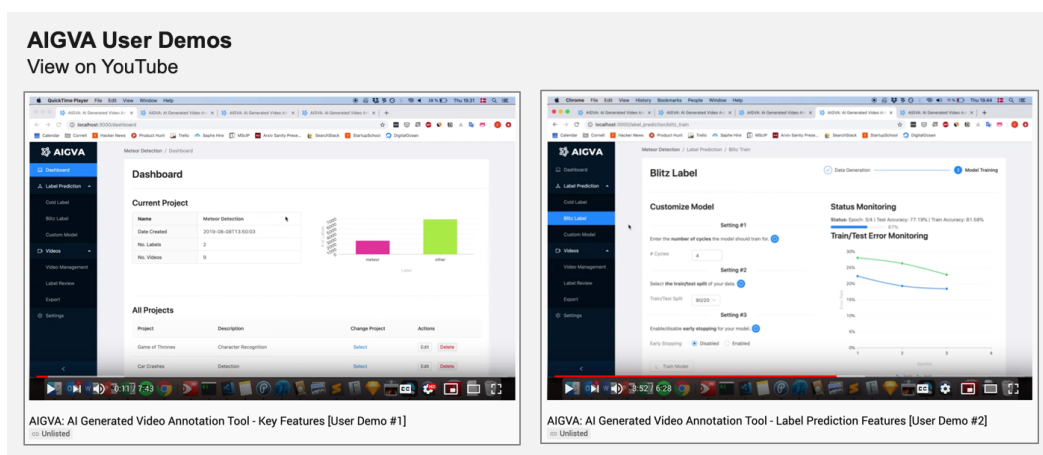


Figure A.1: User Manuals - Available on YouTube

A series of User manuals have been created and uploaded to YouTube.

- User Manual #1: Video/Project Management + Custom-built Video Player:
<https://www.youtube.com/watch?v=xSu6PxUhp88&feature=youtu.be>
- User Manual #2: Label Prediction Features (Cold + Blitz + Custom Model):
<https://www.youtube.com/watch?v=0wDyLw06SZI&feature=youtu.be>

Appendix B

Full Surveys

B.1 Machine Learning Researchers

User Interviews - ML Researcher 1 29/07/2019

1. Is it difficult to label video data / are the problems?

- Manual labelling is tedious and tiresome
- Can do smart things with ResNet/VGG but they are generally bad as they are trained on natural images so they cannot domain shift
- Believes the goal is to create something that is agnostic to domains
- Currently would drag labels from folder to folder

2. "The AIGVA tool is useful for someone having to label video data"? (Highly Disagree - Highly Agree)

- Highly Agree
- Calls it a "powerful tool"
- Likes that it has a Python interface for custom models
- Likes that you can "see" the labels on the video and modify them without dragging from folder to folder

- "I really like that it is domain agnostic"
3. "The AIGVA tool is simple to understand and easy to use"? (Highly Disagree - Highly Agree)
- Highly Agree
 - Difficult to say without tinkering with it for longer
 - Very "straightforward"
4. What features would you like to see?
- GPU enabled prediction
 - Tool between custom model and blitz learning
 - Online learning - ability to feed already reviewed labels into models and train classifier on that
 - Some kind of accuracy measure on a test set to understand performance
5. Any other feedback?
- Looks great! Very slick tool.
 - Suggested running experiment to show usability. Get a dataset like Caltech 101, which is image classification and craft a narrative. Do 80/20 test train split
 - Step 1: Train ResNet on training data and report classification scores
 - Step 2: "Muddle" i.e. mislabel half the labels of the training data and rescore it
 - Step 3: Use AIGVA to relabel data and show performance compared to model 1 and 2

User Interviews - ML Researcher 2 01/08/2019

1. Is it difficult to label video data / are there problems?

- Yeah. He's talked to a fetal cardiologist who has built his own tool.
 - The tool workflow was: manually label subset of data, then run inference, then displays 16 imgs and click the incorrect ones and then manually relabel
2. "The AIGVA tool is useful for someone having to label video data"? (Highly Disagree - Highly Agree)
- Agree
 - Don't always have to label data, but if they did it would be very useful.
3. "The AIGVA tool is simple to understand and easy to use"? (Highly Disagree - Highly Agree)
- Highly Agree
 - "Really nice user interface"
 - "AIGVA makes data labelling very accessible to a non-technical audience - you can do everything without writing a line of code"
 - Great even for the general public he worked at Siemens for a period where he knew they had their own in-house labelling team
4. What features would you like to see?
- Other annotation data: landmark and segmentation
5. Any other feedback?
- He really likes it and can see it be very very useful
 - Custom model implementation using python is great. I showed him the interface with docstring and he verified that he liked it.
 - Could also be useful for few-shot / meta-learning visualizing how it is doing. Has fast feedback loop so useful for research.

- Pretraining on Imagenet good for natural images but may suffer from domain specific

User Interviews - ML Researcher 3 01/08/2019

1. Is it difficult to label video data / are the problems?

- Yeah it's a pain. Especially bad in medicine since it's expensive to get doctors to label + they don't always agree.

2. "The AIGVA tool is useful for someone having to label video data"? (Highly Disagree - Highly Agree)

- Highly Agree
- Very useful. Any way to speed up the process of preparing data is hugely useful. Normally people use first 2 weeks of project just manipulating and curating data.

3. "The AIGVA tool is simple to understand and easy to use"? (Highly Disagree - Highly Agree)

- Highly Agree
- Very accessible and easy to use. "Much better than half the stuff postgrads make."

4. What features would you like to see?

- Other types of labelling: segmentation, bounding boxes, and maybe even pose estimation
- Other types of data: medical data such as segments of fMRIs.
- Worth experimenting with converting medical data into video then inputting into software
- Maybe a heatmap of the confidences similar to the label preview bar.

5. Any other feedback?

- Luca at BioMed has been working a lot with videos. He might be interesting to talk to.
- Really likes the share feature - would be cool if you could share it to many medical professionals and then average the results. It's a big thing in ML at the moment as medical labels are not always 100
- Possibly big commercial opportunity in capturing label variability and displaying it
- Check out the company Cloud Factory that works on similar labelling stuff
- Best applications are safety critical - doesn't matter that Netflix gets a recommendation wrong, but self-driving cars and medical diagnosis need to be much more correct
- Babylon Health interesting company that has GP (even with 70% accuracy where avg. doctor is 60% the NHS didn't approve them - holds them to higher standard so they went to Lagos instead)

User Interviews - ML Researcher 4 01/08/2019

1. Is it difficult to label video data / are there problems?

- Worked on a project where they had to label cardiac ultrasound and segment it frame-by-frame. Had built a matlab tool to guess what the next frame was but it performed quite poorly and had issues with prediction. Used a spline to smooth it.

2. "The AIGVA tool is useful for someone having to label video data"? (Highly Disagree - Highly Agree)

- Highly Agree
- "I have no doubt it could be used today in our research group."

- It would probably be most useful to people doing scan plane detection. Could also be useful for people doing adult cardiac ultrasound - it's hard to label these images because certain planes are difficult to find.
 - Could also be very useful for training Sonographers.
3. "The AIGVA tool is simple to understand and easy to use"? (Highly Disagree - Highly Agree)
- Highly Agree
 - "The interface is very slick"
 - "I don't see many tools like this that look this good"
4. What features would you like to see?
- Other annotation types: landmark + curve would be useful
 - Extension to 4D images: which are 3d images in time dimension
 - Also suggested a confidence bar for labels (like label preview bar)
 - Confidence for user generated labels should be 100%
5. Any other feedback?
- Label ownership: who authored the label. Was it the machine? Was it the user? If it was a user, who? He presented the example of having 4 students do the labelling. If 1 of the 4 was bad, then it would be nice to know which labels he did to be extra careful.
 - Current approach to labelling is dense (i.e. taking into account every frame) - could use a video software like approach with key-frame labelling
 - Really like the tool and could see it being incredibly useful - he loved the model training in browser
 - Ideas to scale to public: rights/security/privacy issues regarding labels + time spent labelling
 - Key idea: "empower manual labelling"

B.2 Clinical Sonographer

Clinical Sonographer Interview - August 2019

1. Is it difficult to label video data / are there problems?

- "Yes. It takes ages."
- They currently use a version of the SonoNet model to collect and label data. However, they recently switched machines from GE to Phillips, which has caused problems since the algorithms used to classify the standard planes were trained on GE machines. Specifically, the algorithm gets a lot of false positives and even false negatives.
- "There is a huge variation in quality of images" meaning not all standard planes are great.
- How it works today: a sonographer will take ca. 13 scans that are freeze-framed from a video. The video is not saved but only the 13 freeze frames.
- Only very few frames of an ultrasound are considered "good". You might have 300 frames of a kidney, but only 5-6 of them are worth saving and worthy of being a standard plane scan.

2. What are the use cases of a tool like AIGVA for sonographers?

- Use case 1: improving algorithms
- Currently the algorithms are not improving. This is a problem because of the domain shift that often happens: either from (1) different sonographers or (2) different imaging systems.
- "It is definitely a clinical problem"
- Use case 2: cataloging videos
- Current videos are not saved. If they could be stored on the AIGVA tool and automatically processed for standard planes, that could be very useful to keep an easily searchable database.

- Could also be useful for legal reasons. Suppose a baby is born with deficiency that the sonographer missed. Legal issues normally arise here, but since only the ca. 13 images are saved, it is normally hard to say whether or not a sonographer missed a plane. By saving the videos and making them easily searchable, you could go back and prove/disprove that sonographers did the right thing.
- Use case 3: training sonographers
- Could be very useful for new sonographers to view different organs/regions across the length of the video.

3. Any extensions?

- Extension 1: 3d volume data
- Requires less training and skill to perform a volumetric scan. But quality of the scan is currently worse. In the future, if scans are better it could cause a domain shift in the way ultrasound is done before. Less trained people perform the scans, leaving more time for sonographers to just analyze scans. Like CT scans today.
- Extension 2: quality control
- Quality control is hugely important but is often not performed around the world. Like any skill, there are good and bad sonographers. A once good sonographer may become out of touch with their skills/knowledge etc. over time.
- A tool that could automatically analyze videos of their ultrasounds and identify where they saved their scan planes and "rate" them, would be extremely useful and would be "adopted everywhere" according to Jacqui. Think of it like an audit of quality of service. Would be a great proactive tool able to detect poorly performing sonographers in advance.

Appendix C

Supporting Material

C.1 Folder Structure

Listing C.1: Web Application Component Structure (Located in: frontend/src)

```
1      components
2          App
3              index.js
4          Charts
5              LossChart.js
6          Dashboard
7              LabelChart.js
8              index.js
9          Export
10             index.js
11          LabelReview
12             index.js
13          MLBlitzLabel
14              BlitzCreate.js
15              BlitzTrain.js
16              index.js
17          MLColdLabel
18              ColdCreate.js
19              ColdReview.js
20              ColdTrain.js
21              index.js
22          MLCustom
23             index.js
24          Navigation
25             index.js
26          Projects
27              AddProject.js
28              EditProject.js
29          Settings
30             index.js
31          VideoAnalysis
32             index.js
33          VideoFramePlayer
34              LabelPreviewBar.js
35              index.js
36          VideoManagement
37              UploadVideo.js
38              index.js
39      constants
40          api.js
41          routes.js
42      index.js
```

C.2 Web Performance Testing

Figure C.1: AIGVA Performance and Accessibility Testing - Lighthouse Score Breakdown



C.3 Experimentation

C.3.1 List of Videos For AIGVA Experiment

Link	Name	Frame Count
https://www.youtube.com/watch?v=RWiQm9Quz0s	Cats And Dogs React To Farts Compilation	1752
https://www.youtube.com/watch?v=RN36RzSjWNw	Funny Cats And Kittens Meowing Compilation 2015	1305
https://www.youtube.com/watch?v=Bejo5TLEpg4	Cute Kittens and Funny Cats Compilation — More Yawns, More smiles	1784
https://www.youtube.com/watch?v=kMhw5MFYU0s	Dogs Who Fail At Being Dogs	1397
https://www.youtube.com/watch?v=q5chhdaNQ88	Funny Dogs Playing With Toys - Funny Dog Videos 2017	1227
https://www.youtube.com/watch?v=teQ299N-tj8	Funny Dogs Compilation 2018 - Dog is Sad About Going to The Vet	2010
https://www.youtube.com/watch?v=xk4LFFm1zAA	ULTIMATE Planes for children — jet planes, bi-planes, seaplanes, airplanes for kids	1630
https://www.youtube.com/watch?v=RokeSWzZAwA	The Wonderful World of Flying (HD) - Wolfe Air Reel by 3DF	1563
https://www.youtube.com/watch?v=028eol-V74g	Airplanes Take Off and Landing Compilation Video (Airplanes Takeoffs)	1431
https://www.youtube.com/watch?v=iK6bx2NMyTo	Top 10 Best Crossovers for the Money — Best Affordable CUVs	1474
https://www.youtube.com/watch?v=.BL1y6v974Y	The 7 Greatest Cars You Can Buy On A Seriously Tight Budget	1856
https://www.youtube.com/watch?v=uJdmjjs0hyc	Best small family cars for 2019/2020 — carwow Top 10	1314

Table C.1: List of YouTube Videos Used

C.3.2 Model Used For AIGVA Experiment

An image classification model was trained on the output of each labelling technique using the fast.ai PyTorch framework. The classifier used was a ResNet34 model pre-trained on ImageNet with the main layers frozen so it was possible to fine-tune it to the training data. The code for running the classifier and obtaining results can be found below.

Listing C.2: ResNet34 Model Used For Experimentation

```

1 from fastai import *
2 from fastai.vision import *
3
4 classes = ['airplane', 'automobile', 'cat']
5 path = Path('empirical_study/')
6
7 data = ImageDataBunch.from_folder(path, train="aigva_output", valid="
    cifar10_3classes",
8     ds_tfms=get_transforms(), size=224, num_workers=4).normalize(
    imagenet_stats)
9
10 learn = cnn_learner(data, models.resnet34, metrics=[accuracy,
    f1_score])

```

```

11
12 learn.fit_one_cycle(1)
13
14 interp = ClassificationInterpretation.from_learner(learn)
15 interp.plot_confusion_matrix()

```

C.3.3 Confusion Matrices for AIGVA Experimentation

Figure C.2: Confusion Matrix for Experiment 1

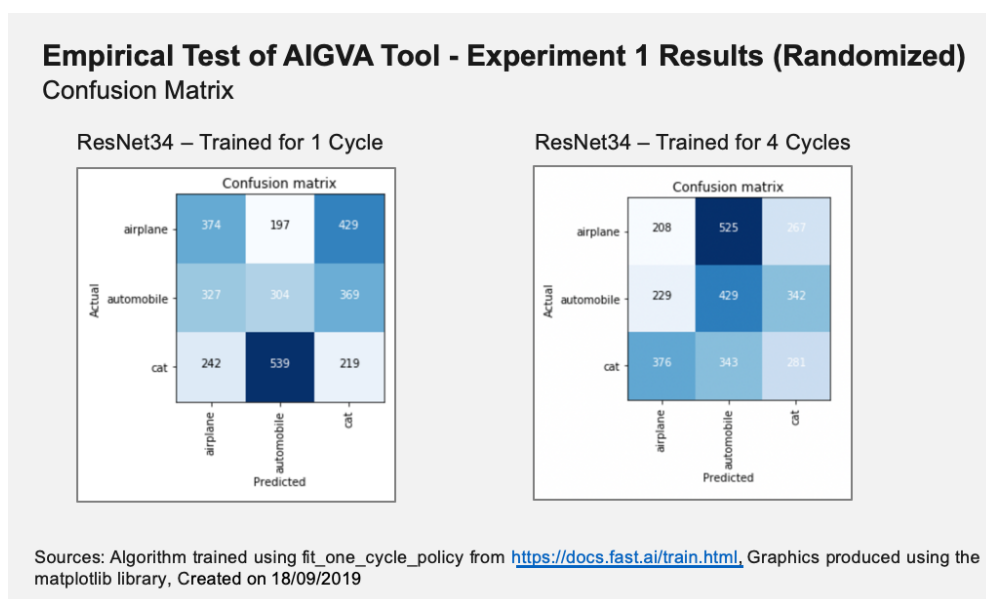


Figure C.3: Confusion Matrix for Experiment 2

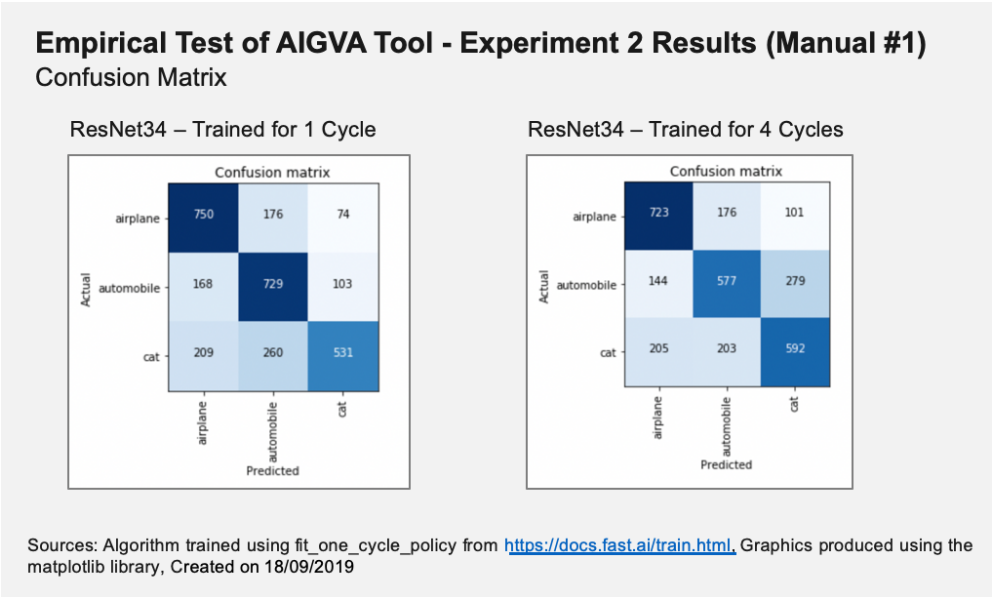


Figure C.4: Confusion Matrix for Experiment 3

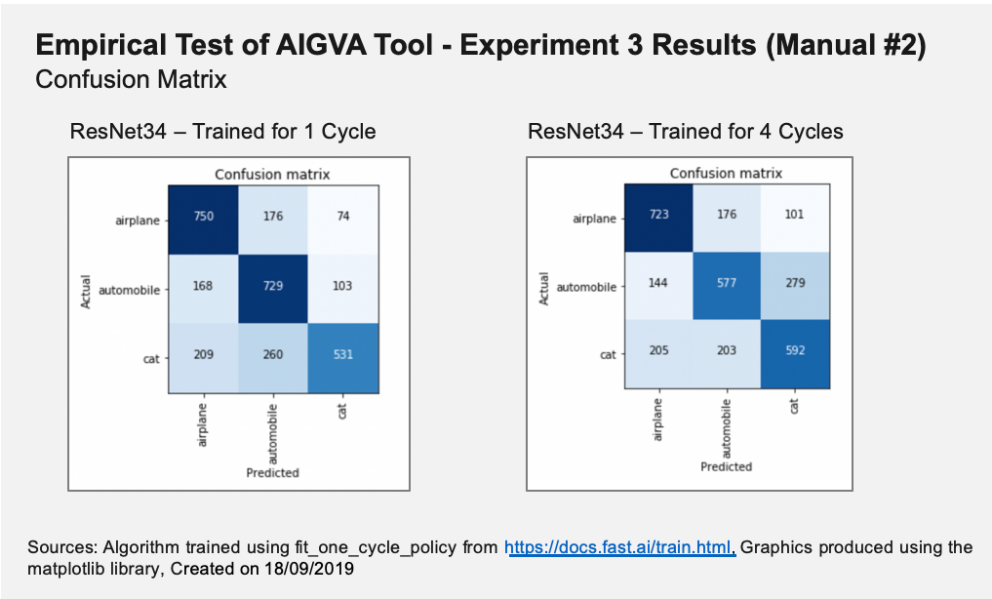


Figure C.5: Confusion Matrix for Experiment 4

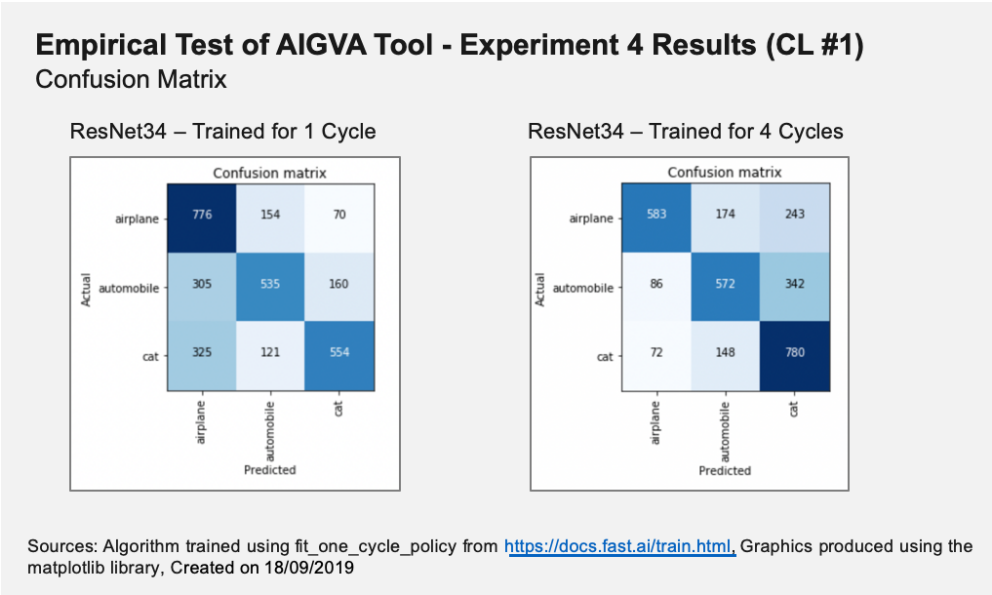


Figure C.6: Confusion Matrix for Experiment 5

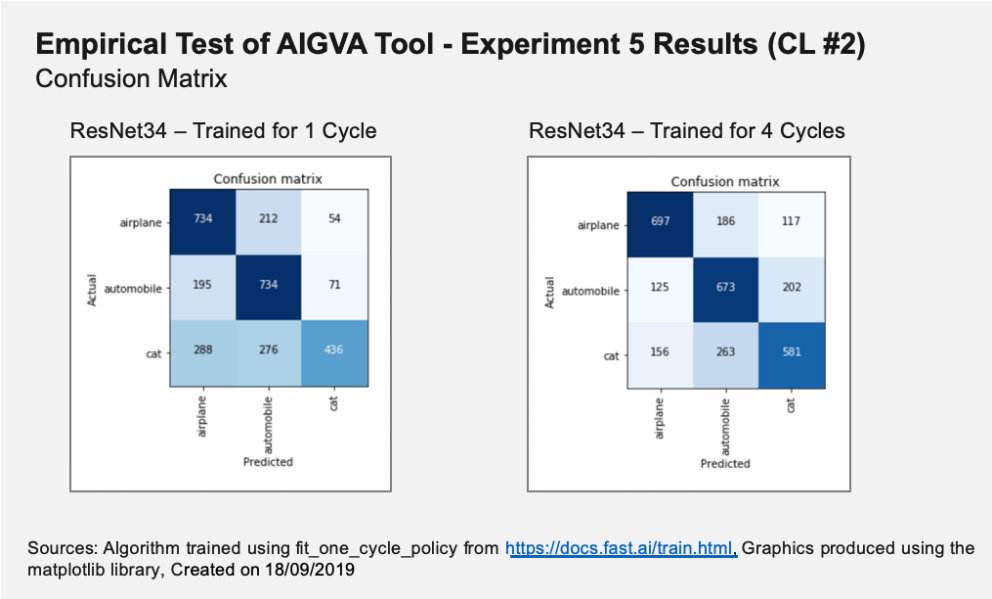


Figure C.7: Confusion Matrix for Experiment 6

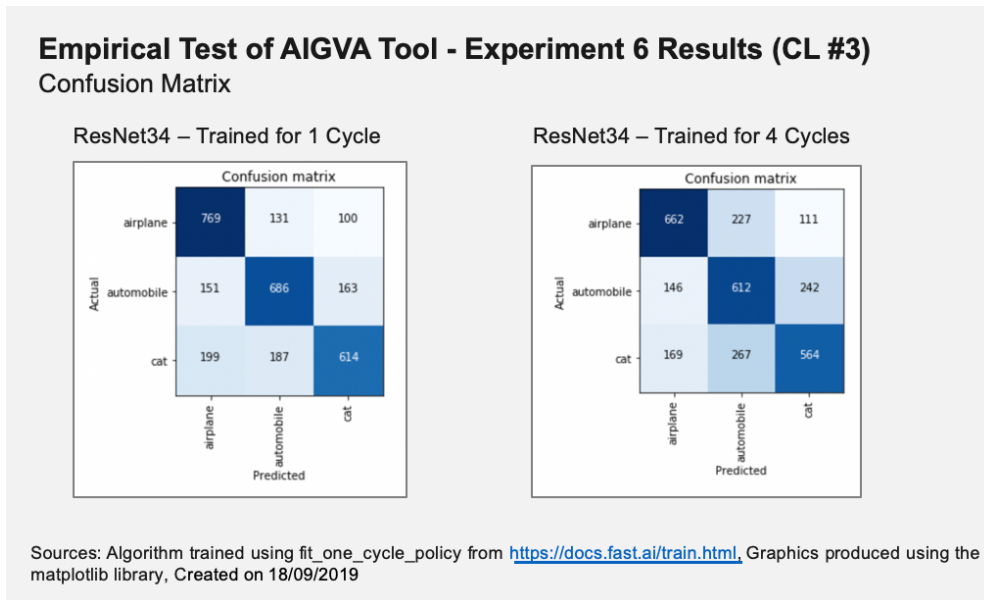


Figure C.8: Confusion Matrix for Experiment 7

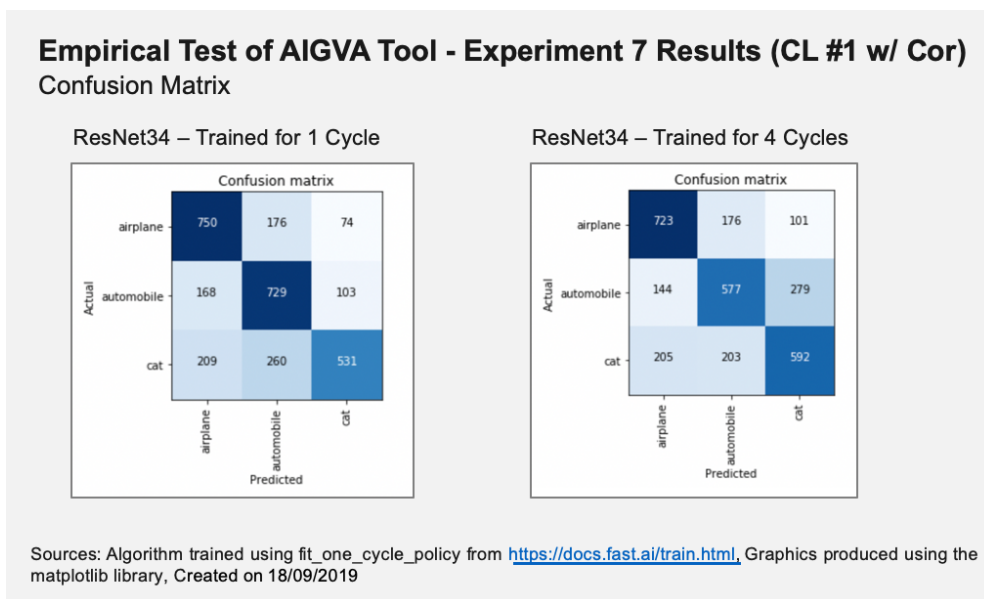


Figure C.9: Confusion Matrix for Experiment 8

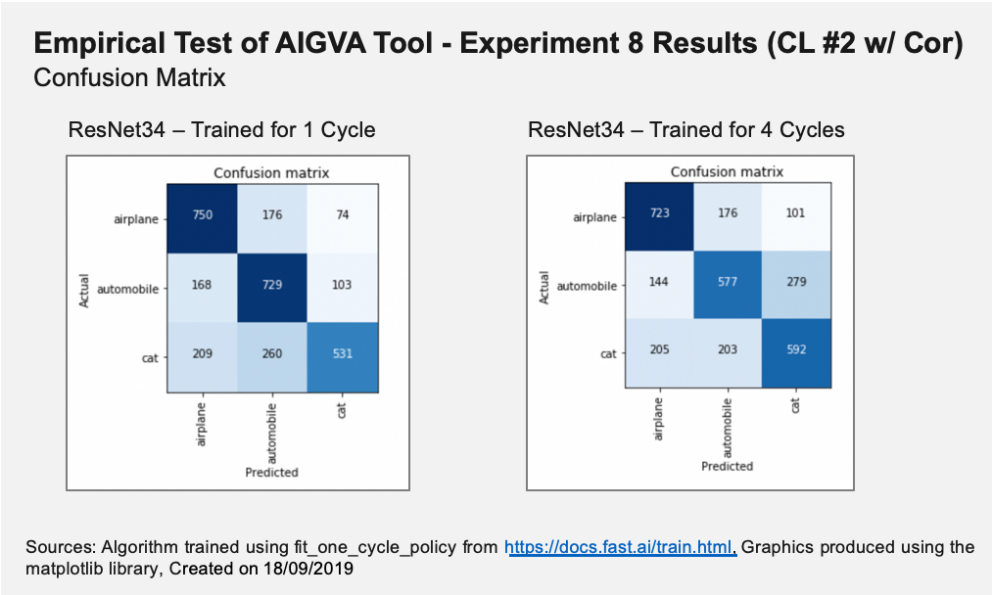
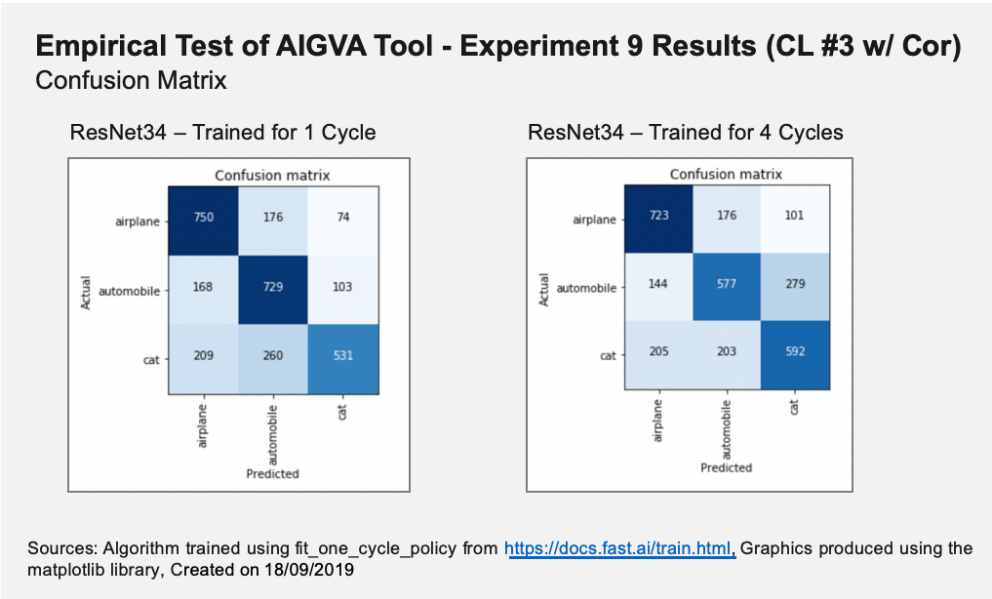


Figure C.10: Confusion Matrix for Experiment 9



Appendix D

Supporting Code

D.1 Web Application

D.1.1 Video Frame Player

React Video Component

Listing D.1: Custom-Built Video Frame Player Full (Located in: frontend/src/components/VideoFramePlayer/index.js)

```
1 class VideoFramePlayer extends Component {
2   constructor(props){
3     super(props)
4
5     this.vidRef = React.createRef();
6     this.state = {
7       video: this.props.source,
8       fps: this.props.fps,
9       duration: 0,
10      time: 0,
11      frame: 0,
12      progress: 0,
13      frames_to_skip: 1,
14      can_play: false,
15    }
16  }
```

```
17
18   startUpdatingFrames() {
19       // Check that no other intervals exist to prevent multiple
20       listeners
21       if (!this.timerID){
22           this.timerID = setInterval(() => this.updateFrames(),
23           INTERVAL_MS);
24       }
25   }
26
27   stopUpdatingFrames() {
28       clearInterval(this.timerID);
29   }
30
31   componentDidMount() {
32       // Initialize event listener to update UI on frame
33       this.updateFrames();
34       this.vidRef.current.addEventListener('canplay', this.
35       handleCanPlayEvent)
36   }
37
38   componentWillUnmount() {
39       // Kill event listener to prevent memory leaks on component
40       dismount
41       this.stopUpdatingFrames();
42       this.vidRef.current.removeEventListener('canplay', this.
43       handleCanPlayEvent)
44   }
45
46   handleCanPlayEvent = () => {
47       this.setState({can_play: true})
48   }
49
50   updateFrames() {
51       const current_frame = Math.floor(this.vidRef.current.
52       currentTime.toFixed(5) * this.state.fps);
53       const current_time = this.vidRef.current.currentTime
```

```
48     const current_progress = current_time/this.vidRef.current.  
duration  
49     this.setState({  
50         frame: current_frame,  
51         time: current_time,  
52         progress: current_progress,  
53     })  
54     this.props.sendFrames(current_frame);  
55 }  
56  
57 onPlay = () => {  
58     this.startUpdatingFrames();  
59     this.vidRef.current.play();  
60 }  
61  
62 onPause = () => {  
63     this.vidRef.current.pause();  
64     this.stopUpdatingFrames();  
65 }  
66  
67 onSliderChange = (value) => {  
68     this.setFrameToTimestamp(value)  
69     this.updateFrames();  
70 }  
71  
72 setFrameToTimestamp = (timestamp) => {  
73     this.vidRef.current.currentTime = parseFloat(timestamp)  
74 }  
75  
76 frameBackward = (frames) => {  
77     const time_minus_a_frame = this.vidRef.current.currentTime -  
(1/this.state.fps)*frames  
78     this.setFrameToTimestamp(time_minus_a_frame)  
79     this.updateFrames();  
80 }  
81  
82 frameForward = (frames) => {
```

```
83     const time_plus_a_frame = this.vidRef.current.currentTime +
      (1/this.state.fps)*frames
84     this.setFrameToTimestamp(time_plus_a_frame)
85     this.updateFrames();
86   }
87
88   labelBackward = () => {
89     const frame_of_prev_label = this.props.getFrameOfPrevLabel()
90     this.skipToFrameNumber(frame_of_prev_label)
91
92   }
93
94   labelForward = () => {
95     const frame_of_next_label = this.props.getFrameOfNextLabel()
96     this.skipToFrameNumber(frame_of_next_label)
97   }
98
99   secondsToMinutesSecondsString = (seconds) => {
100     const mins = Math.floor(seconds/60);
101     let secs = seconds%60;
102     if (secs < 10){
103       secs = '0' + secs
104     }
105     return mins + ':' + secs;
106   }
107
108   skipToFrameNumber = (frame_number) => {
109     const new_time = frame_number/this.state.fps;
110     this.setFrameToTimestamp(new_time)
111     this.updateFrames()
112   }
113
114   handleFrameSkipChange = (frames) => {
115     this.setState({frames_to_skip: frames})
116   }
117
118   handleKeyPress = key => {
```



```
119     if (key === 'right') {
120         this.frameForward(this.state.frames_to_skip)
121     }
122     if (key === 'left') {
123         this.frameBackward(this.state.frames_to_skip)
124     }
125     if (key === 'up') {
126         this.labelForward()
127     }
128     if (key === 'down') {
129         this.labelBackward()
130     }
131 }
132 render() {
133     // JSX TO RENDER PLAYER GOES HERE
134 }
135 }
136
137 export default VideoFramePlayer;
```

Core Data API Video Processing

Listing D.2: Video Processing API (Located in: backend/app/routes_videos.py)

```
1 @videos_api.route("/api/process_video/", methods=["GET"])
2 def process_video():
3     """
4     Processes a video and returns json dump with stats.
5     """
6
7     file_location = request.args.get("file_location")
8     default_label = request.args.get("default_label")
9
10    cap = cv2.VideoCapture(file_location)
11    fps = cap.get(cv2.CAP_PROP_FPS)
12    frame_count = int(cap.get(cv2.CAP_PROP_FRAME_COUNT))
13    cap.release()
14
```

```
15     frames = []
16     for i in range(frame_count):
17         frame = {
18             "Confidence": "0.00",
19             "Frame": 'frame' + str(i) + '.jpg',
20             "Label": default_label
21         }
22         frames.append(frame)
23
24     return json.dumps({
25         'fps': fps,
26         'frame_count': frame_count,
27         'labels': frames,
28     })
```

D.1.2 Export Tool

Listing D.3: Full Download API and Celery Task(Located in: backend/app/app.py)

```
1 @app.route("/api/full_download/", methods = ["GET"])
2 def full_download():
3     """
4     Fetches information from db and then initiates a celery full
5     download task
6     """
7     pid = request.args.get("pid")
8     skip_factor = request.args.get("frames_to_skip")
9
10    label_types = []
11    file_locations = []
12    label_jsons = []
13
14    # 1. Get label types from DB
15    query = Project.query.filter_by(id=pid).first().label_types
16    for label_type in query:
17        label_types.append(label_type['label'])
18    print('label_types fetched: ' + str(label_types))
19
20    # 2. Get file locations and label_jsons from DB
21    query = Label.query.filter_by(pid=pid)
22    for label in query:
23        file_locations.append(label.video.location)
24        label_jsons.append(label.json)
25    print('file_names and labels fetched: ' + str(file_locations))
26
27    # 3. Compute and store in folders
28    task = full_download_to_zip.apply_async(args=[file_locations,
29        label_jsons, label_types, skip_factor])
30
31    return jsonify({'Location': url_for('taskstatus', task_id=task.id)
32        })
```

```

31
32 @celery.task(bind=True)
33 def full_download_to_zip(self, video_links, label_jsons, classes,
34     skip_factor):
35     """
36     Downloads all videos into folders seperated by their labels.
37
38     Args:
39         video_links [str array]: array of all links to videos
40         label_jsons [json array]: array of all json arrays with
41         labels
42         classes [str array]: list of unique label classes
43         skip_factor [int]: number of frames to skip when exporting
44
45     Returns:
46         celery_status_obj [dict]: status of celery task
47     """
48     db.engine.dispose()
49
50     print('[Full Download] Starting full download for: ' + str(
51         video_links))
52
53     # 1: Remove any preexisting .zip files
54     zip_list = glob.glob('./*.zip')
55     for zip_path in zip_list:
56         os.remove(zip_path)
57
58     # 2: Make an export directory to store frames within
59     shutil.rmtree('exports', ignore_errors=True)
60     os.mkdir('exports')
61
62     print('[Full Download] Making directories for the following
63     classes: ' + str(classes))
64
65     for item in classes:
66         os.mkdir('exports/' + item)
67
68     for i in range(len(video_links)):
69         self.update_state(state='PROGRESS',
70             meta={'current': i + 1, 'total': len(

```

```
video_links) + 2,
64         'status': 'Extracing frames from: ' +
    str(os.path.basename(video_links[i])) + '...'})
65     export_video_to_folder(video_links[i], label_jsons[i], int(
skip_factor))
66
67     print('[Full Download] Succesfully downloaded and process files!'
)
68
69     # 3: Save frame and upload to S3
70     timestr = time.strftime("%Y%m%d-%H%M%S")
71     output_name = 'aigva_full_download_' + timestr
72     shutil.make_archive(output_name, 'zip', 'exports')
73     shutil.rmtree('exports', ignore_errors=True)
74     filename = output_name + '.zip'
75     size = humanize(os.path.getsize(filename))
76     self.update_state(state='PROGRESS',
77                       meta={'current': len(video_links) + 1, '
total': len(video_links) + 2,
78                             'status': 'Preparing ' + str(size) +
' to be downloaded...'})
79     s3_link = upload_file_to_s3(filename, 'exports')
80
81     return {'current': len(video_links) + 2, 'total': len(video_links
) + 2, 'status': 'Export completed!',
82            'result': s3_link}
```

D.2 Label Prediction

D.2.1 Cold Label

Listing D.4: Cold Label Image Extraction API (Located in: Backend/app/app.py)

```
1 @celery.task(bind=True)
2 def cold_generate_training_data_worker(self, pid, images_per_video,
3 smart_guess, HAMMING_THRESHOLD=8):
4     """
5     Extracts images_per_video number of images for all videos
6     associated with a project
7
8     Args:
9         pid [int]: project id
10        images_per_video [int]: number of images to extract per video
11
12    Returns:
13        celery_status_obj [dict]: status of celery task
14    """
15
16    db.engine.dispose()
17
18    print('cold_generate_training_data_worker')
19    self.update_state(state='PROGRESS', meta={'current': 0, 'total':
20 1, 'status': 'Fetching videos from database'})
21
22    # 1: Get associated videos from DB
23    video_links=[]
24    videos = Video.query.filter_by(pid=pid)
25    for video in videos:
26        video_links.append(video.location)
27
28    # 2: For each video extract images_per_video # of images and
29    upload to S3
30    s3_links = []
31    dirname = 'coldlabel' + str(pid)
```

```

28     shutil.rmtree(dirname, ignore_errors=True)
29     os.mkdir(dirname)
30     count = 0
31     if smart_guess:
32         for video_link in video_links:
33             cap = cv2.VideoCapture(video_link)
34             video_name = video_link.split("/")[-1].split(".")[0]
35             frame_count = int(cap.get(cv2.CAP_PROP_FRAME_COUNT))
36             fps = int(cap.get(cv2.CAP_PROP_FPS))
37
38             last_hash = None
39             while(cap.isOpened()):
40                 frameId = cap.get(1)
41                 self.update_state(state='PROGRESS', meta={'current':
count + 1*(frameId/frame_count), 'total': len(video_links), '
status': 'Video ' + str(count + 1) + '/' + str(len(video_links)) +
': Downloading and extracing images from ' + video_name})
42                 print("Hashing: " + str(frameId) + '/' + str(
frame_count))
43                 ret, frame = cap.read()
44                 if (ret != True):
45                     break
46                 cap.set(1, frameId + fps)
47
48                 #Perform hashing
49                 current_hash = cv2.img_hash.pHash(frame)
50
51                 # Base case
52                 if (int(frameId) == 0):
53                     last_img = frame
54                 else:
55                     hamming_distance = np.count_nonzero(current_hash
!=last_hash)
56                     if hamming_distance >= HAMMING_THRESHOLD:
57                         last_hash = current_hash
58                         loc = dirname + '/' + str(video_name) + "
_frame" + str(int(frameId)) + ".jpg"

```

```

59         print('Saving image: ' + str(loc))
60         cv2.imwrite(loc, frame)
61         s3_link = upload_file_to_s3(loc, '
coldlabel_training_images')
62         s3_links.append(s3_link)
63         print(s3_link)
64         os.remove(loc)
65     cap.release()
66     count += 1
67 else:
68     for video_link in video_links:
69         cap = cv2.VideoCapture(video_link)
70         video_name = video_link.split("/")[-1].split(".")[0]
71         frame_count = int(cap.get(cv2.CAP_PROP_FRAME_COUNT))
72         skip_factor = frame_count/(images_per_video-1)
73
74         while(cap.isOpened()):
75             frameId = cap.get(1)
76             self.update_state(state='PROGRESS', meta={'current':
count + 1*(frameId/frame_count), 'total': len(video_links), '
status': 'Video ' + str(count + 1) + '/' + str(len(video_links)) +
': Downloading and extracing images from ' + video_name})
77             print(str(frameId) + '/' + str(frame_count))
78             ret, frame = cap.read()
79             if (ret != True):
80                 break
81             cap.set(1, frameId + skip_factor)
82             loc = dirname + '/' + str(video_name) + "_frame" +
str(int(frameId)) + ".jpg"
83             cv2.imwrite(loc, frame)
84             s3_link = upload_file_to_s3(loc, '
coldlabel_training_images')
85             s3_links.append(s3_link)
86             print(s3_link)
87             os.remove(loc)
88
89         cap.release()

```



```

90         count += 1
91
92     # 3: Return array of S3 links as a result and remove dir
93     shutil.rmtree(dirname, ignore_errors=True)
94
95     return {'current': 100, 'total': 100, 'status': 'Predicted!',
96           'result': s3_links}

```

Listing D.5: Cold Label Model Training (Located in: Backend/app/app.py)

```

1  @celery.task(bind=True)
2  def cold_train_model_worker(self, pid, image_links, labels,
3      input_cycles, input_split, input_early_stopping=False):
4      """
5      Trains model given a set of .zip file of images organized by
6      their labels
7
8      Args:
9          pid [int]: project id
10         image_links [str array]: list of image links
11         labels [str array]: list of labels corresponding to image
12         links
13         input_cycles [int]: number of cycles to train for
14         input_split [float]: train test split as pct test
15         input_early_stopping [bool]: enable/disable early stopping
16
17     Returns:
18         celery_status_obj [dict]: status of celery task
19     """
20     db.engine.dispose()
21
22     # 1: Download images and put in folders seperated by their labels
23     self.update_state(state='PROGRESS', meta={'current': 0, 'total':
24     1, 'status': 'Preparing folder structure'})
25     dirname = 'coldlabel' + str(pid)
26     shutil.rmtree(dirname, ignore_errors=True)
27     os.mkdir(dirname)
28     for label in set(labels):

```

```

25         os.mkdir(dirname + '/' + label)
26
27     for i in range(len(image_links)):
28         filename = labels[i] + '_' + str(i) + '.jpg'
29         path = dirname + '/' + labels[i] + '/' + filename
30         print('Downloading ' + str(i + 1) + '/' + str(len(image_links
31             )) + ': ' + str(filename))
32         urllib.request.urlretrieve(image_links[i], path)
33         self.update_state(state='PROGRESS', meta={'current': (i + 1)
34             * 0.3, 'total': len(image_links), 'status': 'Downloading ' + str(i
35                 + 1) + '/' + str(len(image_links)) + ': ' + str(filename)})
36
37     num_images = sum([len(files) for r, d, files in os.walk(dirname)
38         ])
39     print('num_images' + str(num_images))
40
41     # 2: Define model hyperparams
42     MODEL = models.resnet18
43     MODEL_NAME = 'resnet18'
44     TRAIN_CYCLES = input_cycles
45     IMG_SIZE = 128
46     VALID_PCT = input_split
47     BATCH_SIZE = 16
48
49     self.update_state(state='PROGRESS', meta={'current': 0.06, 'total
50         ': 1, 'status': 'Initializing ' + MODEL_NAME})
51
52     # 3: Model hyperparam calculations
53     BATCH_SIZE = int((1 - VALID_PCT) * BATCH_SIZE)
54
55     #4: Initialize model
56     self.update_state(state='PROGRESS', meta={'current': 0.08, 'total
57         ': 1, 'status': 'Normalizing data using ImageNet stats'})
58     data = ImageDataBunch.from_folder(dirname, train=".", valid_pct=
59         VALID_PCT,
60         ds_tfms=get_transforms(), size=IMG_SIZE, num_workers=0, bs=
61         BATCH_SIZE).normalize(imagenet_stats)

```

```

54     list_files('./')
55     #os.mkdir('/nonexistent')
56     learner = cnn_learner(data, MODEL, metrics=[error_rate])
57
58     #5: Train model
59     self.update_state(state='PROGRESS', meta={'current': (0) * 0.9,
60 'total': TRAIN_CYCLES, 'status': "Epoch: 0" + '/' + str(
61 TRAIN_CYCLES) + " | Training..."}))
62     test_accuracy = 0
63     test accuracies = []
64     train_accuracy = 0
65     train accuracies = []
66     for epoch in range(TRAIN_CYCLES):
67         learner.fit_one_cycle(1)
68         self.update_state(state='PROGRESS', meta={'current': (epoch +
69 0.5 ) * 0.9, 'total': TRAIN_CYCLES, 'status': "Epoch: " + str(
70 epoch + 1) + '/' + str(TRAIN_CYCLES) + " | Validating..."}))
71         log_preds = learner.model.eval()
72         prob_test, actual_test = learner.get_preds(ds_type=
73 DatasetType.Valid)
74         prob_train, actual_train = learner.get_preds(ds_type=
75 DatasetType.Train)
76         prev_test_accuracy = test_accuracy
77         test_accuracy = round(1 - error_rate(prob_test, actual_test)
78 .item(),4)
79         train_accuracy = round(1 - error_rate(prob_train,
80 actual_train).item(),4)
81         test accuracies.append(test_accuracy)
82         train accuracies.append(train_accuracy)
83         # Break if early stopping is enabled and test accuracy is
84 lower than prev
85         if ((input_early_stopping==True) and (test_accuracy <
86 prev_test_accuracy)):
87             print('Early stopping')
88             self.update_state(state='PROGRESS', meta={'current': + (
89 epoch + 1 ) * 0.9, 'total': TRAIN_CYCLES, 'status': "[EARLY
90 STOPPING ACTIVATED] Epoch: " + str(epoch + 1) + '/' + str(

```

```

TRAIN_CYCLES) + " | Test Accuracy: " + str(test_accuracy*100) + '
%' + " | Train Accuracy: " + str(train_accuracy*100) + '%', '
accuracies': json.dumps({'test_accuracies': test_accuracies, '
train_accuracies': train_accuracies}))
79         break
80         print("Test Accuracy: " + str(test_accuracy*100) + '%')
81         self.update_state(state='PROGRESS', meta={'current': + (
epoch + 1 ) * 0.9, 'total': TRAIN_CYCLES, 'status': "Epoch: " +
str(epoch + 1) + '/' + str(TRAIN_CYCLES) + " | Test Accuracy: " +
str(round((test_accuracy*100),2)) + '%' + " | Train Accuracy: " +
str(round((train_accuracy*100),2)) + '%', 'accuracies': json.
dumps({'test_accuracies': test_accuracies, 'train_accuracies':
train_accuracies}))
82
83     # 6: Export model, upload to S3
84     model_name = str(MODEL_NAME) + '_' + str(TRAIN_CYCLES) + 'cycles_
' + str(IMG_SIZE) + 'imgsize_' + str(num_images) + 'noimgs_' + str
(VALID_PCT) + 'valpct' '.pkl'
85     self.update_state(state='PROGRESS', meta={'current': 0.95, 'total
': 1, 'status': 'Exporting trained model as ' + model_name, '
accuracies': json.dumps({'test_accuracies': test_accuracies, '
train_accuracies': train_accuracies}))
86     learner.export(model_name)
87     print('Uploading to S3')
88     s3_link = upload_file_to_s3(dirname + '/' + model_name, '
blitzlabel_models')
89     print('S3 Link: ' + str(s3_link))
90
91     #7: Add model to PostgresDB
92     model_obj = Model(date = datetime.datetime.now(), name =
model_name, location = s3_link, accuracy=float(test_accuracy*100),
num_images=num_images, type = 'cold' , pid = pid, file_size =
humanize(os.path.getsize(dirname + '/' + model_name)))
93     db.session.add(model_obj)
94     db.session.commit()
95
96     #8: Delete model

```

```
97     shutil.rmtree(dirname, ignore_errors=True)
98
99     return {'current': 1, 'total': 1, 'status': model_name + ' ready
    for prediction', 'result': 'Done!'}
```

D.2.2 Blitz Label

Listing D.6: Blitz Label Model Training (Located in: Backend/app/app.py)

```

1 @celery.task(bind=True)
2 def blitz_train_model_worker(self, pid, zip_link, input_cycles,
   input_split, input_early_stopping=False):
3     """
4     Trains model given a set of .zip file of images organized by
   their labels
5
6     Args:
7         pid [int]: project id
8         zip_link [str]: link to zip file with images organized by
   labels in subfolders
9         input_cycles [int]: number of cycles to train for
10        input_split [float]: train test split as pct test
11        input_early_stopping [bool]: enable/disable early stopping
12
13    Returns:
14        celery_status_obj [dict]: status of celery task
15    """
16    db.engine.dispose()
17
18    # 1: Download images and put in folders seperated by their labels
19    self.update_state(state='PROGRESS', meta={'current': 0, 'total':
20    1, 'status': 'Preparing folder structure'})
21    dirname = 'blitzlabel' + str(pid)
22    shutil.rmtree(dirname, ignore_errors=True)
23    os.mkdir(dirname)
24    print('Downloading ' + str(zip_link))
25    self.update_state(state='PROGRESS', meta={'current': 0.04, 'total
26    ': 1, 'status': 'Computing hyperparameters'})
27
28    urllib.request.urlretrieve(zip_link, dirname + '/data.zip')
29    shutil.unpack_archive(dirname + '/data.zip', dirname)
30    num_images = sum([len(files) for r, d, files in os.walk(dirname)

```

```

1)
29     print('num_images' + str(num_images))
30
31     # 2: Define model hyperparams
32     MODEL            = models.resnet18
33     MODEL_NAME       = 'resnet18'
34     TRAIN_CYCLES     = input_cycles
35     IMG_SIZE         = 128
36     VALID_PCT        = input_split
37     BATCH_SIZE       = 16
38
39     self.update_state(state='PROGRESS', meta={'current': 0.06, 'total
': 1, 'status': 'Initializing ' + MODEL_NAME})
40
41     # 3: Model hyperparam calculations
42     BATCH_SIZE = int((1 - VALID_PCT) * BATCH_SIZE)
43
44     #4: Initialize model
45     self.update_state(state='PROGRESS', meta={'current': 0.08, 'total
': 1, 'status': 'Normalizing data using ImageNet stats'})
46     data = ImageDataBunch.from_folder(dirname, train=".", valid_pct=
VALID_PCT,
47         ds_tfms=get_transforms(), size=IMG_SIZE, num_workers=0, bs=
BATCH_SIZE).normalize(imagenet_stats)
48     list_files('./')
49     #os.mkdir('/nonexistent')
50     learner = cnn_learner(data, MODEL, metrics=[error_rate])
51
52     #5: Train model
53     self.update_state(state='PROGRESS', meta={'current': (0) * 0.9,
'total': TRAIN_CYCLES, 'status': "Epoch: 0" + '/' + str(
TRAIN_CYCLES) + " | Training..."})
54     test_accuracy      = 0
55     test accuracies    = []
56     train_accuracy     = 0
57     train accuracies   = []
58     for epoch in range(TRAIN_CYCLES):

```

```

59     learner.fit_one_cycle(1)
60     self.update_state(state='PROGRESS', meta={'current': (epoch +
        0.5 ) * 0.9, 'total': TRAIN_CYCLES, 'status': "Epoch: " + str(
epoch + 1) + '/' + str(TRAIN_CYCLES) + " | Validating..."})
61     log_preds = learner.model.eval()
62     prob_test, actual_test = learner.get_preds(ds_type=
DatasetType.Valid)
63     prob_train, actual_train = learner.get_preds(ds_type=
DatasetType.Train)
64     prev_test_accuracy = test_accuracy
65     test_accuracy = round(1 - error_rate(prob_test, actual_test)
.item(),4)
66     train_accuracy = round(1 - error_rate(prob_train,
actual_train).item(),4)
67     test_accuracies.append(test_accuracy)
68     train_accuracies.append(train_accuracy)
69     # Break if early stopping is enabled and test accuracy is
lower than prev
70     if ((input_early_stopping==True) and (test_accuracy <
prev_test_accuracy)):
71         print('Early stopping')
72         self.update_state(state='PROGRESS', meta={'current': + (
epoch + 1 ) * 0.9, 'total': TRAIN_CYCLES, 'status': "[EARLY
STOPPING ACTIVATED] Epoch: " + str(epoch + 1) + '/' + str(
TRAIN_CYCLES) + " | Test Accuracy: " + str(test_accuracy*100) + '
%' + " | Train Accuracy: " + str(train_accuracy*100) + '%', '
accuracies': json.dumps({'test_accuracies': test_accuracies, '
train_accuracies': train_accuracies})})
73         break
74         print("Test Accuracy: " + str(test_accuracy*100) + '%')
75     self.update_state(state='PROGRESS', meta={'current': + (
epoch + 1 ) * 0.9, 'total': TRAIN_CYCLES, 'status': "Epoch: " +
str(epoch + 1) + '/' + str(TRAIN_CYCLES) + " | Test Accuracy: " +
str(round((test_accuracy*100),2)) + '%' + " | Train Accuracy: " +
str(round((train_accuracy*100),2)) + '%', 'accuracies': json.
dumps({'test_accuracies': test_accuracies, 'train_accuracies':
train_accuracies})})

```



```
76
77     # 6: Export model, upload to S3
78     model_name = str(MODEL_NAME) + '_' + str(TRAIN_CYCLES) + 'cycles_'
79     ' + str(IMG_SIZE) + 'imgsize_' + str(num_images) + 'noimgs_' + str
80     (VALID_PCT) + 'valpct' '.pkl'
81     self.update_state(state='PROGRESS', meta={'current': 0.95, 'total
82     ': 1, 'status': 'Exporting trained model as ' + model_name, '
83     accuracies': json.dumps({'test_accuracies': test_accuracies, '
84     train_accuracies': train_accuracies})})
85     learner.export(model_name)
86     print('Uploading to S3')
87     s3_link = upload_file_to_s3(dirname + '/' + model_name, '
88     blitzlabel_models')
89     print('S3 Link: ' + str(s3_link))
90
91     #7: Add model to PostgresDB
92     model_obj = Model(date = datetime.datetime.now(), name =
93     model_name, location = s3_link, accuracy=float(test_accuracy*100),
94     num_images=num_images, type = 'blitz' , pid = pid, file_size =
95     humanize(os.path.getsize(dirname + '/' + model_name)))
96     db.session.add(model_obj)
97     db.session.commit()
98
99     #8: Delete model
100     shutil.rmtree(dirname, ignore_errors=True)
101
102     return {'current': 1, 'total': 1, 'status': model_name + ' ready
103     for prediction', 'result': 'Done!'}
```

D.2.3 Custom Model

Listing D.7: Custom Model Template - Example (Located in: Backend/app/custom_model.py)

```
1 def custom_model_predict(img_path):
2     """
3     This is a custom interface for using your own custom models.
4
5     Called from custom_model_predict_label_and_update_db in app.py
6
7     Args:
8         img_path: filepath to the images that needs to be predicted.
9
10    Returns:
11        prediction [str] : label predicted by your model
12        confidence [float] : confidence that the label is correct in
13        range 0-1
14    """
15
16    # 1: Load learner
17    learner = load_learner(
18        './', './models/ultrasound_model.pkl')
19
20    # 2: Open image and predict
21    img = open_image(img_path)
22    pred = learner.predict(img)
23
24    # 3: Parse predictions into correct output format
25    prediction = str(pred[0])
26    confidence = torch.max(pred[2]).item()
27
28    return prediction, confidence
```

D.2.4 Prediction Engine

Listing D.8: Label Prediction API and Celery Task (Located in: Backend/app/app.py)

```

1 @celery.task(bind=True)
2 def downloaded_model_predict_label_and_update_db(self, lid,
3   skip_factor, video_link, model, MIN_FRAMES = 10):
4     """
5     Predicts all labels in video using downloaded model and updates
6     database.
7
8     Args:
9         lid [int]: label id
10        skip_factor [int]: number of frames to skip when predicting
11        video_link [str]: link to video
12        model [str]: link to model
13        MIN_FRAMES [int]: optimization param (min number of adjacent
14        frames a label can have)
15
16    Returns:
17        celery_status_obj [dict]: status of celery task
18    """
19    db.engine.dispose()
20
21    #1: Download and open video stream
22    self.update_state(state='PROGRESS', meta={'current': 0, 'total':
23    1, 'status': 'Downloading: ' + str(video_link)})
24    video_name = video_link.split("/")[-1].split(".")[0]
25    count = 0
26    cap = cv2.VideoCapture(video_link)
27
28    # 2: Download learner model
29    self.update_state(state='PROGRESS', meta={'current': 0, 'total':
30    1, 'status': 'Preparing ML model: ' + str(model['name'])})
31    downloaded_model = datetime.datetime.now().strftime("%Y%m%d-%H%M%
32    S%f") + model['name']
33    urllib.request.urlretrieve(model['location'], downloaded_model)

```

```
28 learner = load_learner('./', downloaded_model)
29
30 # 3: Prepare folder structure and get frame count
31 shutil.rmtree(video_name, ignore_errors=True)
32 os.mkdir(video_name)
33 frame_count = int(cap.get(cv2.CAP_PROP_FRAME_COUNT))
34 output = []
35
36 # 4: Loop through video frame by frame
37 while(cap.isOpened()):
38     frameId = cap.get(1)
39     ret, frame = cap.read()
40     # 4.1 Exit when video has no more frames
41     if (ret != True):
42         break
43     # 4.2 Only predict every skip_factor frames
44     if (frameId % math.floor(skip_factor) == 0):
45         self.update_state(state='PROGRESS',
46                             meta={'current': count, 'total':
frame_count,
47                                     'status': 'Predicting Frame ' + str(
count) + '/' + str(frame_count)})
48         # 4.2.1 Create image and open it
49         loc = "./" + video_name + "/frame.jpg"
50         cv2.imwrite(loc, frame)
51         img = open_image(loc)
52
53         # 4.2.2 Get prediction
54         pred = learner.predict(img)
55
56         # 4.2.3 Prepare write
57         tmp = []
58         filename = 'frame' + str(count) + '.jpg'
59         prediction = str(pred[0])
60         confidence = torch.max(pred[2]).item()
61         tmp.append(confidence)
62         tmp.append(filename)
```

```

63         tmp.append(prediction)
64         output.append(tmp)
65         print(str(count) + '/' + str(frame_count))
66
67         # 4.2.4 Remove image
68         os.remove(loc)
69         count += 1*skip_factor
70
71     cap.release()
72     shutil.rmtree(video_name, ignore_errors=True)
73     os.remove(downloaded_model)
74
75     self.update_state(state='PROGRESS',
76                      meta={'current': count, 'total':
77                           frame_count,
78                           'status': 'Uploading ' + str(
79                               frame_count) + ' labels to database' })
79
80     # 5. Run optimiser which only allows a label change if the new
81     # frame is consistently predicted for MIN_FRAMES
82     prev_label = output[0][2]
83     for i in range(len(output)):
84         current_label = output[i][2]
85         if current_label != prev_label:
86             next_change = -1
87             try:
88                 for j in range(1, MIN_FRAMES + 1):
89                     next_label = output[i+j][2]
90                     if next_label != current_label:
91                         next_change = j
92             except:
93                 if next_change != -1:
94                     output[i][2] = prev_label
95
96     # 6. Create frames array

```

```
107 frames = []
108 for i in range(len(output)):
109     frame = {
110         "Confidence": output[i][0],
111         "Frame": output[i][1],
112         "Label": output[i][2]
113     }
114     frames.append(frame)
115
116 # 7: Commit labels to database
117 query = Label.query.filter_by(id=lid).first()
118 query.json = frames
119 query.mid = model['id']
120 query.reviewed = False
121 query.date = datetime.datetime.now()
122 db.session.commit()
123
124 return {'current': 100, 'total': 100, 'status': 'Predicted!',
125         'result': 'Success!'}
126
127 @app.route("/api/predict_label/", methods = ["GET"])
128 def predict_label():
129     """
130     Initiates a label prediction celery task
131     """
132     lid = request.args.get("lid")
133     skip_factor = request.args.get("frames_to_skip")
134     video_link = request.args.get("video_link")
135
136     task = custom_model_predict_label_and_update_db.apply_async(args
137 = [lid, int(skip_factor), video_link])
138
139     return jsonify({'Location': url_for('taskstatus', task_id=task.id)
140 })
```

D.3 DevOps

D.3.1 Docker

Listing D.9: Dockerfile (Located in: Backend/)

```
1 FROM python:3.7
2
3 RUN apt-get update \
4     && apt-get install -y \
5         build-essential \
6         cmake \
7         git \
8         wget \
9         unzip \
10        yasm \
11        pkg-config \
12        libswscale-dev \
13        libtbb2 \
14        libtbb-dev \
15        libjpeg-dev \
16        libpng-dev \
17        libtiff-dev \
18        libavformat-dev \
19        libpq-dev \
20        && rm -rf /var/lib/apt/lists/*
21
22 RUN pip install numpy
23
24 WORKDIR /
25 ENV OPENCV_VERSION="4.1.0"
26 RUN wget https://github.com/opencv/opencv/archive/${OPENCV_VERSION}.
    zip \
27 && unzip ${OPENCV_VERSION}.zip \
28 && mkdir /opencv-${OPENCV_VERSION}/cmake_binary \
29 && cd /opencv-${OPENCV_VERSION}/cmake_binary \
30 && cmake -DBUILD_TIFF=ON \
```

```

31 -DBUILD_opencv_java=OFF \
32 -DWITH_CUDA=OFF \
33 -DWITH_OPENGL=ON \
34 -DWITH_OPENCL=ON \
35 -DWITH_IPP=ON \
36 -DWITH_TBB=ON \
37 -DWITH_EIGEN=ON \
38 -DWITH_V4L=ON \
39 -DWITH_FFMPEG=1 \
40 -DBUILD_TESTS=OFF \
41 -DBUILD_PERF_TESTS=OFF \
42 -DCMAKE_BUILD_TYPE=RELEASE \
43 -DCMAKE_INSTALL_PREFIX=$(python3.7 -c "import sys; print(sys.prefix
   )" ) \
44 -DPYTHON_EXECUTABLE=$(which python3.7) \
45 -DPYTHON_INCLUDE_DIR=$(python3.7 -c "from distutils.sysconfig
   import get_python_inc; print(get_python_inc())" ) \
46 -DPYTHON_PACKAGES_PATH=$(python3.7 -c "from distutils.sysconfig
   import get_python_lib; print(get_python_lib())" ) \
47 .. \
48 && make install \
49 && rm /${OPENCV_VERSION}.zip \
50 && rm -r /opencv-${OPENCV_VERSION}
51 RUN ln -s \
52 /usr/local/python/cv2/python-3.7/cv2.cpython-37m-x86_64-linux-gnu.
   so \
53 /usr/local/lib/python3.7/site-packages/cv2.so
54
55 EXPOSE 5000
56
57 ADD requirements.txt /app/requirements.txt
58 WORKDIR /app/
59
60 RUN pip install -r requirements.txt --no-cache-dir
61 RUN pip install gunicorn
62
63 RUN adduser --disabled-password --gecos '' app

```



```
64 RUN chmod 777 /app/  
65 RUN chmod -R 777 ../tmp/  
66 RUN mkdir /nonexistent  
67 RUN chmod -R 777 ../nonexistent/  
68 RUN ls
```

D.4 File Storage and Database

D.4.1 Database Schema

Listing D.10: Database Model (Located in: Backend/app/app.py)

```
1 class Project(db.Model):
2     id          = db.Column(db.Integer, primary_key=True)
3     name        = db.Column(db.Unicode)
4     desc        = db.Column(db.Unicode)
5     main_label  = db.Column(db.Unicode)
6     label_types = db.Column(db.JSON)
7     date        = db.Column(db.DateTime)
8     videos      = db.relationship("Video", backref=db.backref('
project'))
9     models      = db.relationship("Model", backref=db.backref('
project'))
10    labels      = db.relationship("Label", backref=db.backref('
project'))
11
12 class Video(db.Model):
13     id          = db.Column(db.Integer, primary_key=True)
14     name        = db.Column(db.Unicode)
15     location    = db.Column(db.Unicode)
16     fps         = db.Column(db.Float)
17     size        = db.Column(db.Integer)
18     pid         = db.Column(db.Integer, db.ForeignKey('project.id'))
19     labels      = db.relationship("Label", backref=db.backref('video'
))
20
21 class Model(db.Model):
22     id          = db.Column(db.Integer, primary_key=True)
23     name        = db.Column(db.Unicode)
24     date        = db.Column(db.DateTime)
25     location    = db.Column(db.Unicode)
26     type        = db.Column(db.Unicode)
27     file_size   = db.Column(db.Unicode, nullable=True)
```

```
28     accuracy      = db.Column(db.Float, nullable=True)
29     num_images     = db.Column(db.Integer, nullable=True)
30     pid            = db.Column(db.Integer, db.ForeignKey('project.id'))
31     labels         = db.relationship("Label", backref=db.backref('model'
32                                ))
33
34 class Label(db.Model):
35     id              = db.Column(db.Integer, primary_key=True)
36     date            = db.Column(db.DateTime)
37     json            = db.Column(db.JSON) #JSON
38     frame_count     = db.Column(db.Integer, nullable=True)
39     reviewed        = db.Column(db.Boolean)
40     vid             = db.Column(db.Integer, db.ForeignKey('video.id'))
41     mid             = db.Column(db.Integer, db.ForeignKey('model.id'),
42                                nullable=True)
43     pid             = db.Column(db.Integer, db.ForeignKey('project.id'))
```

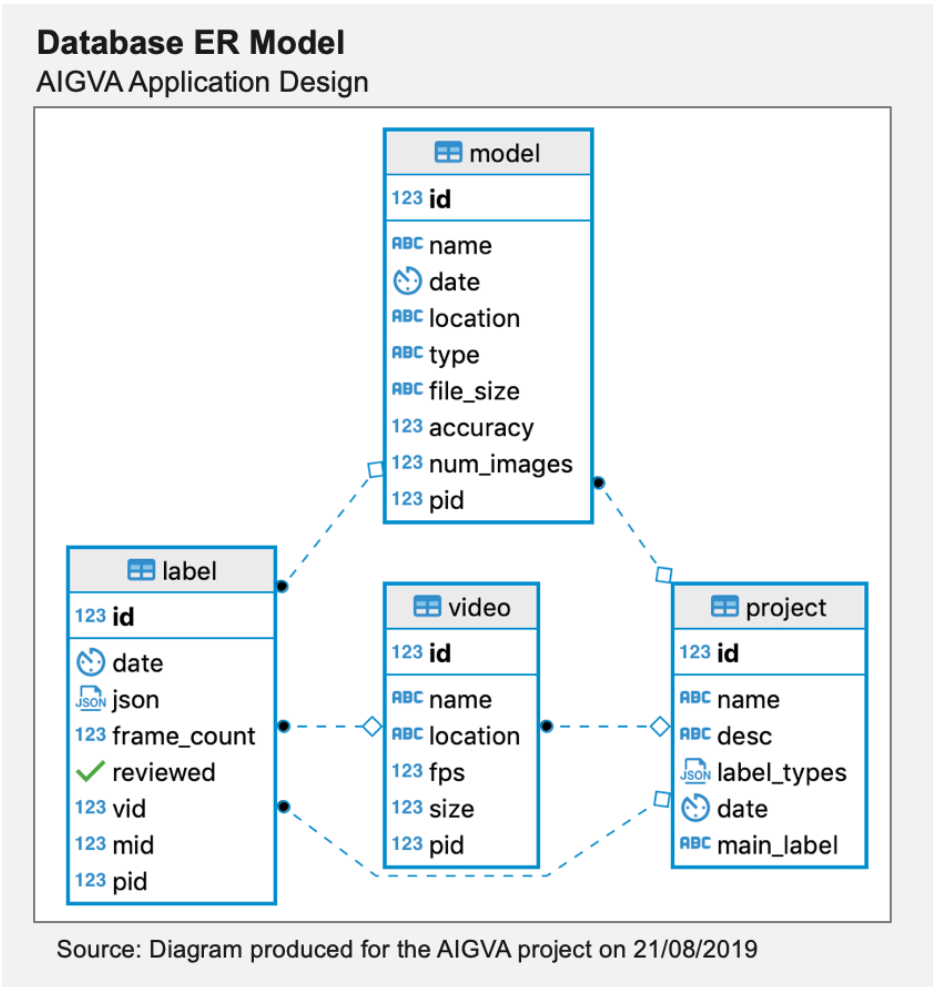


Figure D.1: AIGVA Database Model