GENERATIVE APPROACH TO THE AUTOMATION OF ARTIFICIAL INTELLIGENCE APPLICATIONS

Calvin Huang[1] and Yu Sun[2]

[1]University High School, 4771 Campus Dr, Irvine, CA 92612
[2]California State Polytechnic University,
Pomona, CA, 91768, Irvine, CA 92620

ABSTRACT

In order to use the full power of artificial intelligence, many are required to navigate through a complex process that involves reading and understanding code. Understanding this process can be especially intimidating to domain experts who wish to use A.I to develop a project, but have no former experience with programming. This paper develops an application to allow for any domain expert (or normal person) to gather data, assign labels, and train models automatically without the use of software to do so. Our application, through a server, allows the user to send HTTP API requests to train models, upload images to the database, add models/labels, and access models/labels.

KEYWORDS

Tensorflow Lite, Flask, Flutter, Google Colab.

1. INTRODUCTION

With the rise of popularity of artificial intelligence throughout the last few decades, the world has seen an interweaving between A.I and certain academic domains [1]. Artificial Intelligence and its many applications have been used throughout a variety of critical and far-reaching projects. From medical research in cancer detection to blind assistance systems, image-detection has been used in so many impactful projects [2][3]. It soon became imperative for certain domain experts who want to combine image detection to their projects to have a thorough understanding of programming, training models, gathering data, navigating through Integrated Development Environments, and concepts in Convolutional Neural Networks [4].

Image detection models are quickly becoming an extremely powerful tool for domain experts to use [5]. By requiring them to understand a different subject entirely when they are focused on another academic interest may be time-consuming and inefficient. For example, in order to actually train a model on a system such as Google Colab, they would need to go through a lengthy chunk of code, import files filled with their training data, and export the finished model [6]. The process is too time consuming for non-experts and even general programmers to use. Furthermore, they may need to coordinate and hire machine-learning engineers, which convolutes the process and makes the overall project more complex. Additionally, many domain experts may already be incredibly invested in their own field, which could deter them from taking the time to learn the concepts of machine learning [7]. However, if there was a way to introduce an abstraction that could allow them to train the models without any code, the process becomes significantly less challenging.

Some of the existing tools that have been used to make machine-learning more friendly and efficient for a non-experienced user are Google Colab Notebooks and an application called CreateML [8]. Google Colab, a hosted Jupyter notebook service developed by Google, gives anyone the ability to train a model simply by accessing a prewritten notebook on their site [9]. Without needing to install an IDE such as Jupyter, a user is theoretically able to add their own data-set into the notebook, and run the code in the prewritten notebook such that eventually the model is created. However, this method is not as efficient or user-friendly as our approach. Going through each section of code, and storing each image into a file one by one makes the process time-consuming and unappealing. In addition, some pieces of code may be confusing or unrecognizable to some users who want to use image-detection, which requires more programming knowledge.

Another tool that relates to this issue is CreateML, an application developed by Apple. CreateML allows the user to train an image-detection model without having to write a single line of code. However, this application is again quite limited as users do not have the ability to easily create their own datasets. Instead, they are forced to take potentially thousands of their own images, upload it all into its own folder, and then use their application to train the model. Thus, like Google Colab, the process of training is again very inefficient and time-consuming for a non-expert. Furthermore, CreateML, like many applications that attempt a Low-Code No-Code approach to image-detection, has a difficult User Interface for non-experts and therefore makes the process more difficult for them [10]. Although there are several existing approaches to making image-detection efficient for a non-expert, many of them are tedious, frustrating, and quite unfriendly for someone looking to utilize this tool but doesn’t have much experience with it.

In this paper, we propose a solution that allows a non-expert to train a model and utilize it in an intuitive and straightforward way. The application gives the user both the ability to train a model and use that exact same model. In the admin page, they can either select an existing model or create a new model with any name they want. After, they can give the model different label names, and for each label they can take pictures that correspond to that label. Using this method, non-experts will be able to gather a data-set without having to upload pictures onto a file and then utilize existing tools. After pictures are taken for each label, they can train the model.

The application also allows the user to use the image-detection model. They are given the option to load models, and by choosing an existing model that has already been trained, they can take a screenshot of what they want to compute. This directs them to a page that gives them the probability of the image being a certain label within the model that they choose. Therefore, we believe that our application is not only more friendly for a non-user, but also less time-consuming and more efficient for any tasks that need to implement an image-detection model.

In order to prove that our mobile application would be less tedious, more efficient, and include similar functionality as the standard approach, we conducted three experiments to compare the functionalities, compare the end-to-end processes, and evaluate the accuracy and confidence of the image-detection model used in our mobile application. First, by comparing the functionality of both approaches, we create a checklist to determine if our application succeeds in carrying out certain tasks that are instrumental to the process of machine learning. However, we also analyzed the benefits of using our application as well, and show that not only do we carry out such tasks, but we do it in a more intuitive manner. Furthermore, we compared the end-to-end processes, which allowed us to illustrate the strength of our application: our efficiency. Our results showed that while most of the methods in the typical approach required lines of code and the organization of images, our approach was able to simplify the process by only requiring users to make several clicks, type several words, and take shots with their phone camera. Finally, by evaluating our accuracy and confidence, we were able to show that the accuracy of our application would not differ from the accuracy of the typical approach. The goal of using these experiments in tandem is to demonstrate that our approach is not only the most efficient method for training and evaluating models, but it is also the most intuitive and user-friendly system for non-experts to use if they wish to create and organize a list of image-detection models.

The rest of the paper is organized as follows: Section 2 gives the details on the challenges that we met during the experiment and designing the sample; Section 3 focuses on the details of our solutions corresponding to the challenges that we mentioned in Section 2; Section 4 presents the relevant details about the experiment we did, following by presenting the related work in Section 5. Finally, Section 6 gives the conclusion remarks, as well as pointing out the future work of this project.

2. CHALLENGES

In order to build the project, a few challenges have been identified as follows.

2.1. How do we allow the user to create their own datasets in an efficient way

Perhaps the most tedious part of training an image-detection model is gathering the data. There are a couple of options. For instance, if a user prefers to create a model using existing datasets, then online platforms such as Kaggle can provide the user with thousands of images already organized in its specific labels [11]. There are certainly many websites that give users the ability to gather lots of data. However, if a user needed to customize their own data-set and execute a model based on their own images, they would need to take individual pictures for each label, export the images to a computer, store each image in its corresponding label file, and then export the file to be used to train the model on a notebook. This process is clearly very time-consuming and thus stops non-experts or other people without any experience in code to use image-detection in their projects or other initiatives.

2.2. How do we design a user-interface that is easy to navigate for a non-expert

Because the purpose of the application is to target those who do not have experience with machine learning, the user-interface must also be friendly and intuitive for them. However, because there are so many terms and concepts in Machine Learning, it is quite difficult to introduce a user-interface that doesn’t require the user to have at least a basic understanding of the training process in machine learning. For example, in order to understand the process of training a model, the user must be able to understand terms such as labels, training set, and test set. This issue is accelerated even further by the fact that understanding machine learning requires an understanding of coding concepts. If a domain-expert who wants to utilize image-detection does not even know how to code, they would be forced to either work with another engineer or learn by themselves, which is more time-consuming and less efficient.

2.3. How do we introduce an approach on a mobile application that trains the dataset

In order to allow users to use an application on their phones to build models, the app must include an approach to not just organize the dataset, but also train the model. The difficulty lies in the fact that training the model on a phone is too computationally expensive to train. The process would take more than one standard smartphone, which is unreliable and serves as poor user experience. Furthermore, the smartphone must also hold in potentially thousands of pictures to train the model. This is clearly not doable and thus the application must have some method in which a server is called in order to train the model.

3. Solution

Our application provides an approach to train image-detection models, gather training data, and compute accuracy of testing data on a mobile device. In order to customize the model, we used the Tensorflow Lite Model Maker, a library that reduces the training time and amount of training data, as a means of customizing each image detection model [12]. Our application has three main components - a front-end consisting of an admin and consumer page, a back-end, and a database.

The user is first greeted with a splash screen, and then interacts with the UI on the main menu. The frontend consists of text, buttons, and a list which holds in the different routing pages. From here, the user can choose to either customize their model by selecting "Model Admin", or test their model by selecting "Model Test". By selecting "Model Test", the UI will consist of a text field allowing the user to add a new model, and a list of past models that they can customize and train. Selecting a model will redirect the user to a new page where they can add labels to the model, or edit an existing label. By selecting a label, the user will need to capture images using their camera. The more images they take for each label, the more accurate the overall model. Once they take enough pictures for each label, they can select the "Training!" button to train the model.

If the admin chooses to go to the consumer page, the user will be greeted by a list of trained models. By selecting one, they will need to take a picture of whatever object they want. Once the picture is taken, the user will be shown a screen detailing the label that the image taken corresponds with and the likelihood of the model being correct.

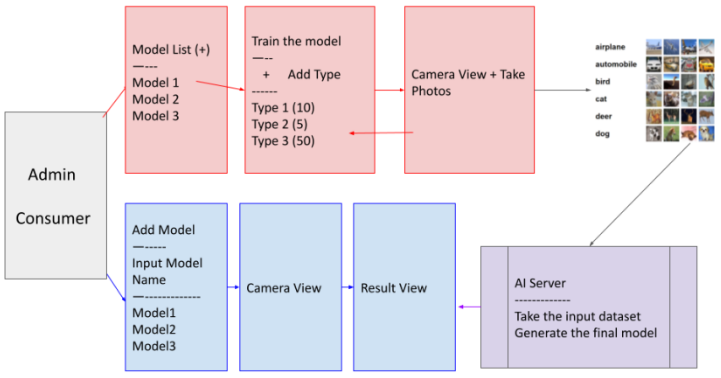


Figure 1. Overview of the solution

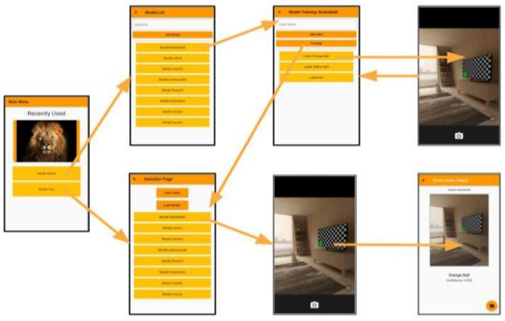
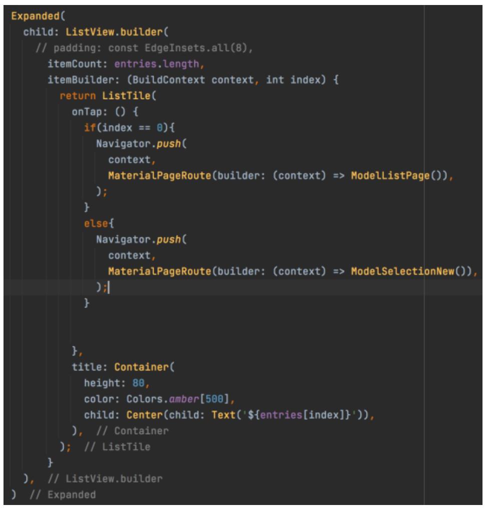


Figure 2. Screenshot of App process

The front-end of the application was developed using Flutter], a UI software development kit created by Google that supports both iOS and Android versions of the application [13]. The Main Menu page was built utilizing the ListView Class, which holds the Page Routers to either the Admin Page or the Consumer Page. Within the Admin portion, we used both the TextField Class to gather the names of any new Model IDs or labels inputted by the user, and the ListView Class to load any new Model IDs or labels. Within the Consumer portion, we also used the ListView Class to load any trained models for testing, and a button class that allowed the user to clear the cache. Once the user takes a picture on the Consumer side, a page is shown with an image of the label it computes to be most accurate, and a text displaying the accuracy.

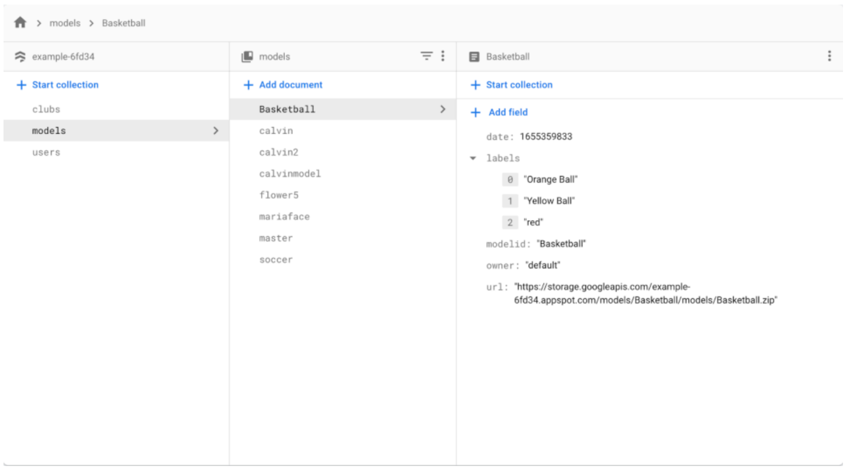


```
Expanded(
  child: ListView.builder(
    // padding: const EdgeInsets.all(8),
      itemCount: entries.length,
      itemBuilder: (BuildContext context, int index) {
        return ListTile(
          onTap: () {
            if(index == 0){
              Navigator.push(
                context,
                MaterialPageRoute(builder: (context) => ModelListPage()),
              );
            }
            else{
              Navigator.push(
                context,
                MaterialPageRoute(builder: (context) => ModelSelectionNew()),
              );
            }


          },
          title: Container(
            height: 80,
            color: Colors.amber[500],
            child: Center(child: Text('${entries[index]}')),
          ), // Container
        ); // ListTile
      }
  ), // ListView.builder
) // Expanded
```

Figure 3. Screenshot of code 1

There are two elements in the ListView: a page router to the admin section and a page router to the consumer section. Clicking on either element in the list will direct the user to that specific section.

The backend was made using a Python Flask server which holds 6 main HTTP APIs [14]. Flask is a web framework that allows for the routing of HTTP requests to the specified controller. The backend is connected to a Firebase database that stores the model names, model labels, and a url which consists of the labels text file and the tflite file of the trained model [15]. In addition, the database stores each image taken by the user for each label they select. By taking advantage of the HTTP APIs from the Flask server, we were able to access and edit the items within the Firebase database. Consequently, we were able to create changes on the front-end UI as well.

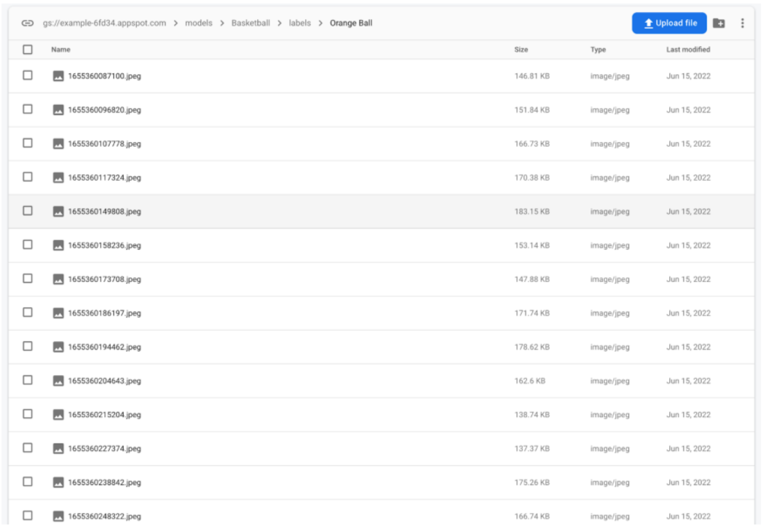


models > Basketball

example-6fd34	models	Basketball
+ Start collection	+ Add document	+ Start collection
clubs	Basketball	+ Add field
models	calvin	date: 1655359833
users	calvin2	labels
	calvinmodel	0 "Orange Ball"
	flower5	1 "Yellow Ball"
	mariaface	2 "red"
	master	modelid: "Basketball"
	soccer	owner: "default"
		url: "https://storage.googleapis.com/example-6fd34.appspot.com/models/Basketball/models/Basketball.zip"

Figure 4. Firestore Database

This image shows the Firestore Database, which holds a variety of models, with its branches having properties such as label names, model ids, and a url containing the tflite file and the label file. This structure allows us to utilize the HTTP APIs to access and edit the properties.



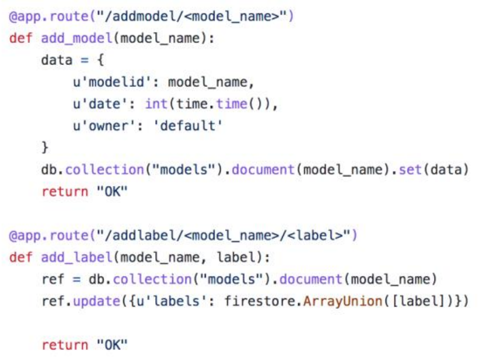
gs://example-6fd34.appspot.com > models > Basketball > labels > Orange Ball

Name	Size	Type	Last modified
1655360087100.jpeg	146.81 KB	image/jpeg	Jun 15, 2022
1655360096820.jpeg	151.84 KB	image/jpeg	Jun 15, 2022
1655360107778.jpeg	166.73 KB	image/jpeg	Jun 15, 2022
1655360117324.jpeg	170.38 KB	image/jpeg	Jun 15, 2022
1655360149808.jpeg	183.15 KB	image/jpeg	Jun 15, 2022
1655360158236.jpeg	153.14 KB	image/jpeg	Jun 15, 2022
1655360173708.jpeg	147.88 KB	image/jpeg	Jun 15, 2022
1655360186197.jpeg	171.74 KB	image/jpeg	Jun 15, 2022
1655360194462.jpeg	178.62 KB	image/jpeg	Jun 15, 2022
1655360204643.jpeg	162.6 KB	image/jpeg	Jun 15, 2022
1655360215204.jpeg	138.74 KB	image/jpeg	Jun 15, 2022
1655360227374.jpeg	137.37 KB	image/jpeg	Jun 15, 2022
1655360238842.jpeg	175.26 KB	image/jpeg	Jun 15, 2022
1655360248322.jpeg	166.74 KB	image/jpeg	Jun 15, 2022

Figure 5. The storage for all the images contained for each label

This image shows the storage for all the images contained for each label. For the example above, the label Orange Ball is selected for the model Basketball. The storage will contain a list of all the pictures that will be taken by the user in the model admin.

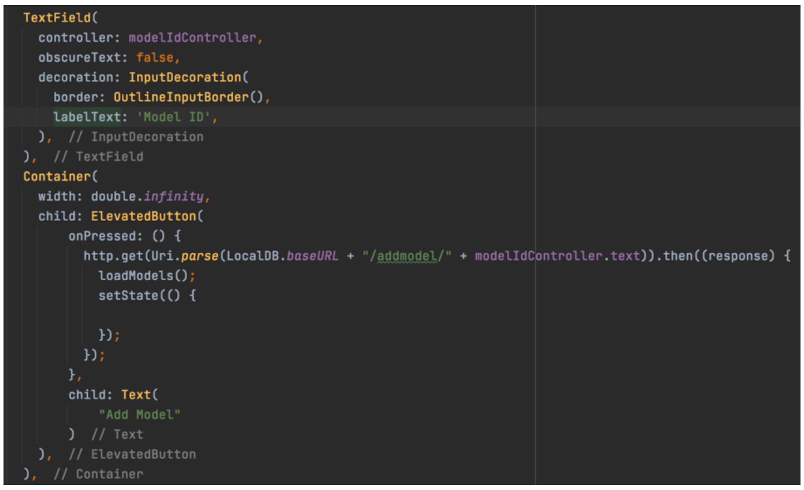
Our application uses an HTTP API named addmodel, which when given the name as a parameter, will add a new model branch in our Firebase. This allows users to create as many models with different names as they want. Similarly, we used another HTTP API named addlabel, which has two parameters: the name of an existing model and a new name for the label. By providing the name of the existing model, the user is able to attach this new label to the branch of that model as a new property.



```
@app.route("/addmodel/<model_name>")
def add_model(model_name):
    data = {
        u'modelid': model_name,
        u'date': int(time.time()),
        u'owner': 'default'
    }
    db.collection("models").document(model_name).set(data)
    return "OK"

@app.route("/addlabel/<model_name>/<label>")
def add_label(model_name, label):
    ref = db.collection("models").document(model_name)
    ref.update({u'labels': firestore.ArrayUnion([label])})

    return "OK"
```

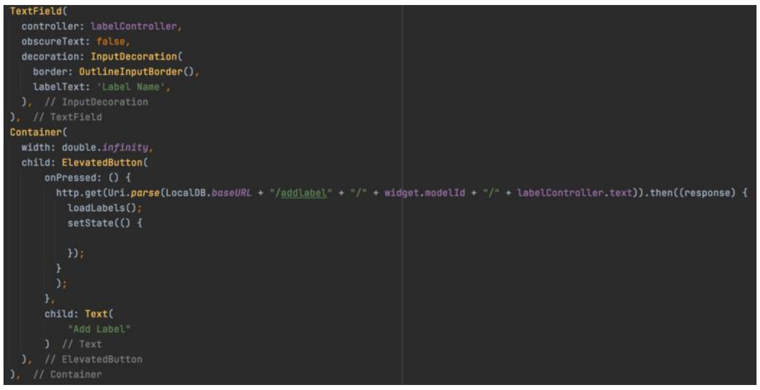
Figure 6. Screenshot of code 2

The Python Flask representation of the APIs for add_model and add_label. Both will access the Firestore Database and add specific values based on the user input.



```
TextField(
  controller: modelIdController,
  obscureText: false,
  decoration: InputDecoration(
    border: OutlineInputBorder(),
    labelText: 'Model ID',
  ),  // InputDecoration
),  // TextField
Container(
  width: double.infinity,
  child: ElevatedButton(
      onPressed: () {
        http.get(Uri.parse(LocalDB.baseURL + "/addmodel/" + modelIdController.text)).then((response) {
          loadModels();
          setState(() {

          });
        });
      },
      child: Text(
          "Add Model"
      )  // Text
  ),  // ElevatedButton
),  // Container
```

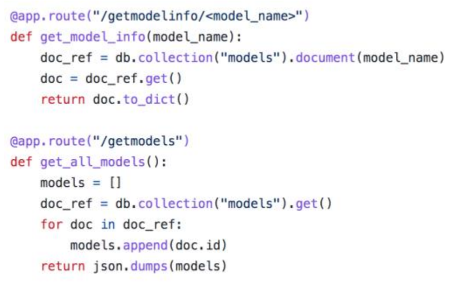
Figure 7. Screenshot of code 3



```
TextField(
  controller: labelController,
  obscureText: false,
  decoration: InputDecoration(
    border: OutlineInputBorder(),
    labelText: 'Label Name',
  ),  // InputDecoration
),  // TextField
Container(
  width: double.infinity,
  child: ElevatedButton(
      onPressed: () {
        http.get(Uri.parse(LocalDB.baseURL + "/addlabel" + "/" + widget.modelId + "/" + labelController.text)).then((response) {
          loadLabels();
          setState(() {

          });
        }
        );
      },
      child: Text(
          "Add Label"
      )  // Text
  ),  // ElevatedButton
),  // Container
```

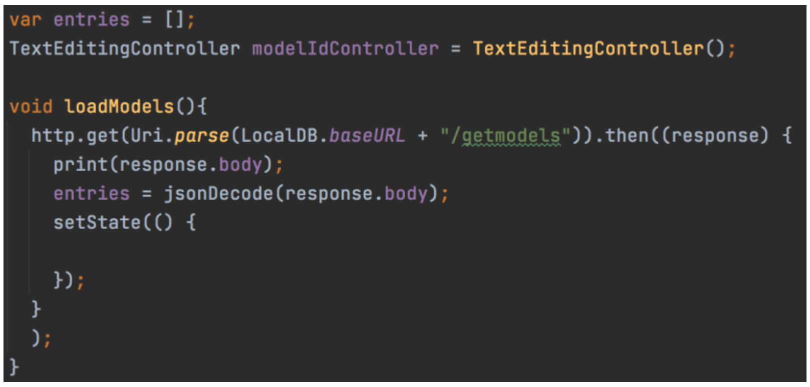
Figure 8. Screenshot of code 4

The app also uses two different HTTP APIs to gain access to all the model branches and labels for each particular model branch - get_all_models and get_model_info respectively. get_all_models, when executed by an HTTP request, returns a list of the name properties of all the models. This allows us to utilize the ListView class to linearly display each model with a text that holds the name property. get_model_info returns a Python dictionary that stores key-value pairs of objects. In order to gain access to the list of all labels, we set the key property to “labels”, allowing us again to display all the names of the labels as a ListView class.



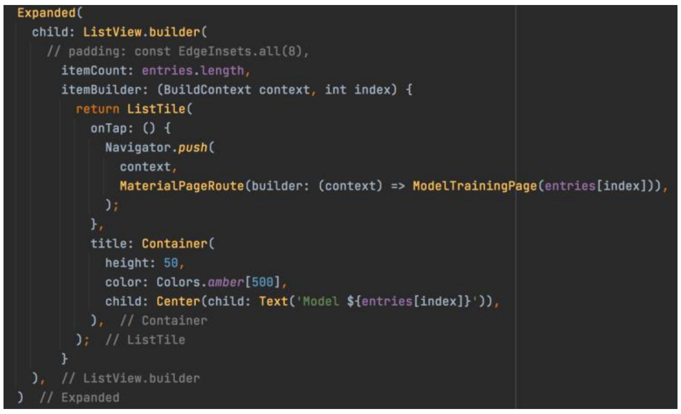
```python
@app.route("/getmodelinfo/<model_name>")
def get_model_info(model_name):
    doc_ref = db.collection("models").document(model_name)
    doc = doc_ref.get()
    return doc.to_dict()

@app.route("/getmodels")
def get_all_models():
    models = []
    doc_ref = db.collection("models").get()
    for doc in doc_ref:
        models.append(doc.id)
    return json.dumps(models)
```

Figure 9. Screenshot of code 5



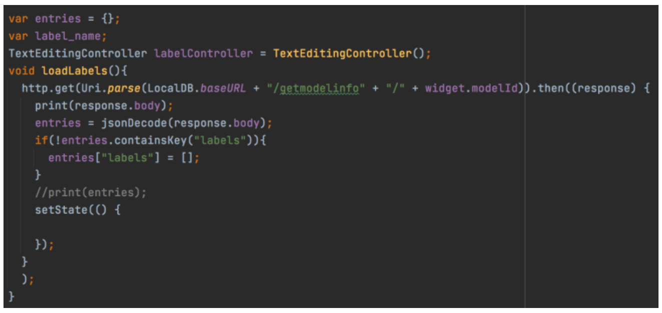
```dart
var entries = [];
TextEditingController modelIdController = TextEditingController();

void loadModels(){
  http.get(Uri.parse(LocalDB.baseURL + "/getmodels")).then((response) {
    print(response.body);
    entries = jsonDecode(response.body);
    setState(() {

    });
  }
  );
}
```

Figure 10. Screenshot of code 6



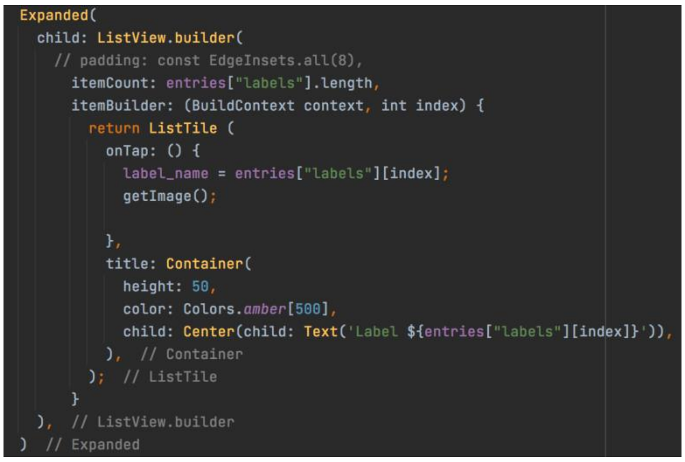
```dart
Expanded(
  child: ListView.builder(
    // padding: const EdgeInsets.all(8),
      itemCount: entries.length,
      itemBuilder: (BuildContext context, int index) {
        return ListTile(
          onTap: () {
            Navigator.push(
              context,
              MaterialPageRoute(builder: (context) => ModelTrainingPage(entries[index])),
            );
          },
          title: Container(
            height: 50,
            color: Colors.amber[500],
            child: Center(child: Text('Model ${entries[index]}')),
          ), // Container
        ); // ListTile
      }
  ), // ListView.builder
) // Expanded
```

Figure 11. Screenshot of code 7



```
var entries = {};
var label_name;
TextEditingController labelController = TextEditingController();
void loadLabels(){
  http.get(Uri.parse(LocalDB.baseURL + "/getmodelinfo" + "/" + widget.modelId)).then((response) {
    print(response.body);
    entries = jsonDecode(response.body);
    if(!entries.containsKey("labels")){
      entries["labels"] = [];
    }
    //print(entries);
    setState(() {

    });
  }
  );
}
```

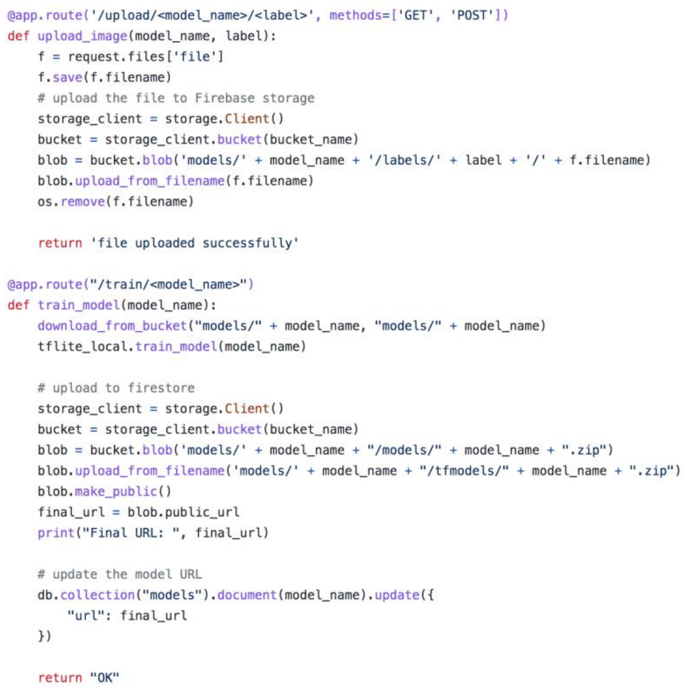
Figure 12. Screenshot of code 8



```
Expanded(
  child: ListView.builder(
    // padding: const EdgeInsets.all(8),
      itemCount: entries["labels"].length,
      itemBuilder: (BuildContext context, int index) {
        return ListTile (
          onTap: () {
            label_name = entries["labels"][index];
            getImage();

          },
          title: Container(
            height: 50,
            color: Colors.amber[500],
            child: Center(child: Text('Label ${entries["labels"][index]}')),
          ), // Container
        ); // ListTile
      }
  ), // ListView.builder
) // Expanded
```

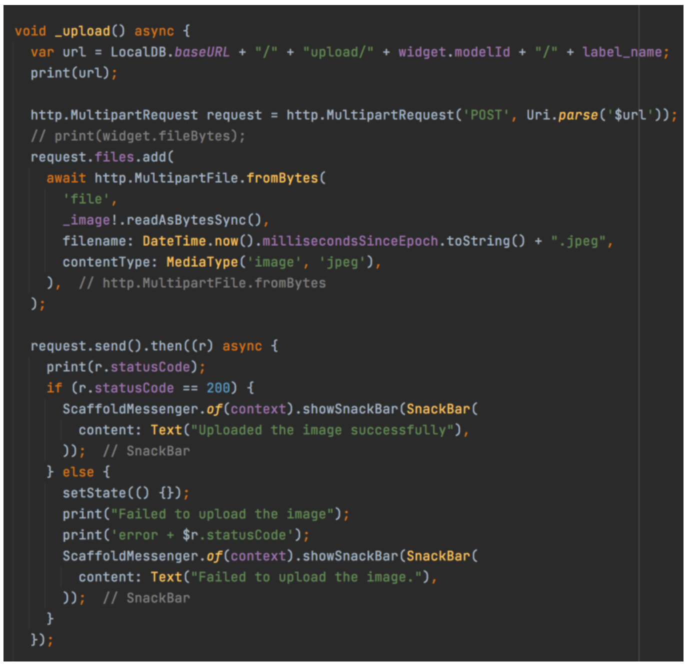
Figure 13. Screenshot of code 9

The fifth HTTP API we used was called train_model, which trains the model based on the labels and then uploads the model file into the Firebase. This allows us to call the get_model_info HTTP API on the consumer side, where we can set the key value of the dictionary to “url” to gain access to the model file for testing. The final HTTP API, upload_image, saves the picture taken by the user and stores the file to the Firebase as a property of each label. This in turn will allow the train_model API to gain access to these images and train the model.



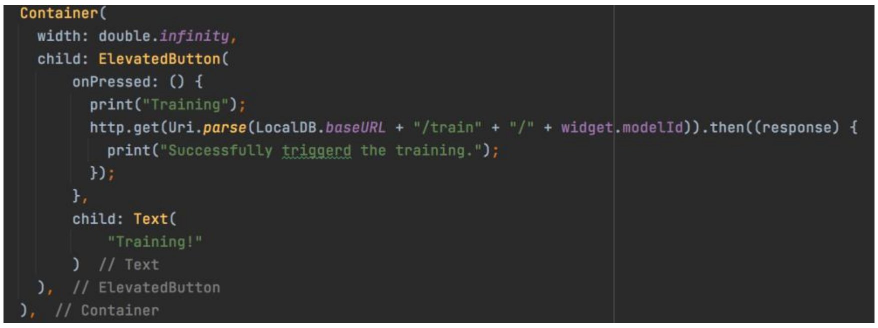
```python
@app.route('/upload/<model_name>/<label>', methods=['GET', 'POST'])
def upload_image(model_name, label):
    f = request.files['file']
    f.save(f.filename)
    # upload the file to Firebase storage
    storage_client = storage.Client()
    bucket = storage_client.bucket(bucket_name)
    blob = bucket.blob('models/' + model_name + '/labels/' + label + '/' + f.filename)
    blob.upload_from_filename(f.filename)
    os.remove(f.filename)

    return 'file uploaded successfully'

@app.route("/train/<model_name>")
def train_model(model_name):
    download_from_bucket("models/" + model_name, "models/" + model_name)
    tflite_local.train_model(model_name)

    # upload to firestore
    storage_client = storage.Client()
    bucket = storage_client.bucket(bucket_name)
    blob = bucket.blob('models/' + model_name + "/models/" + model_name + ".zip")
    blob.upload_from_filename('models/' + model_name + "/tfmodels/" + model_name + ".zip")
    blob.make_public()
    final_url = blob.public_url
    print("Final URL: ", final_url)

    # update the model URL
    db.collection("models").document(model_name).update({
        "url": final_url
    })

    return "OK"
```

Figure 14. Screenshot of code 10



```dart
void _upload() async {
  var url = LocalDB.baseURL + "/" + "upload/" + widget.modelId + "/" + label_name;
  print(url);

  http.MultipartRequest request = http.MultipartRequest('POST', Uri.parse('$url'));
  // print(widget.fileBytes);
  request.files.add(
    await http.MultipartFile.fromBytes(
      'file',
      _image!.readAsBytesSync(),
      filename: DateTime.now().millisecondsSinceEpoch.toString() + ".jpeg",
      contentType: MediaType('image', 'jpeg'),
    ),  // http.MultipartFile.fromBytes
  );

  request.send().then((r) async {
    print(r.statusCode);
    if (r.statusCode == 200) {
      ScaffoldMessenger.of(context).showSnackBar(SnackBar(
        content: Text("Uploaded the image successfully"),
      ));  // SnackBar
    } else {
      setState(() {});
      print("Failed to upload the image");
      print('error + $r.statusCode');
      ScaffoldMessenger.of(context).showSnackBar(SnackBar(
        content: Text("Failed to upload the image."),
      ));  // SnackBar
    }
  });
```

Figure 15. Screenshot of code 11

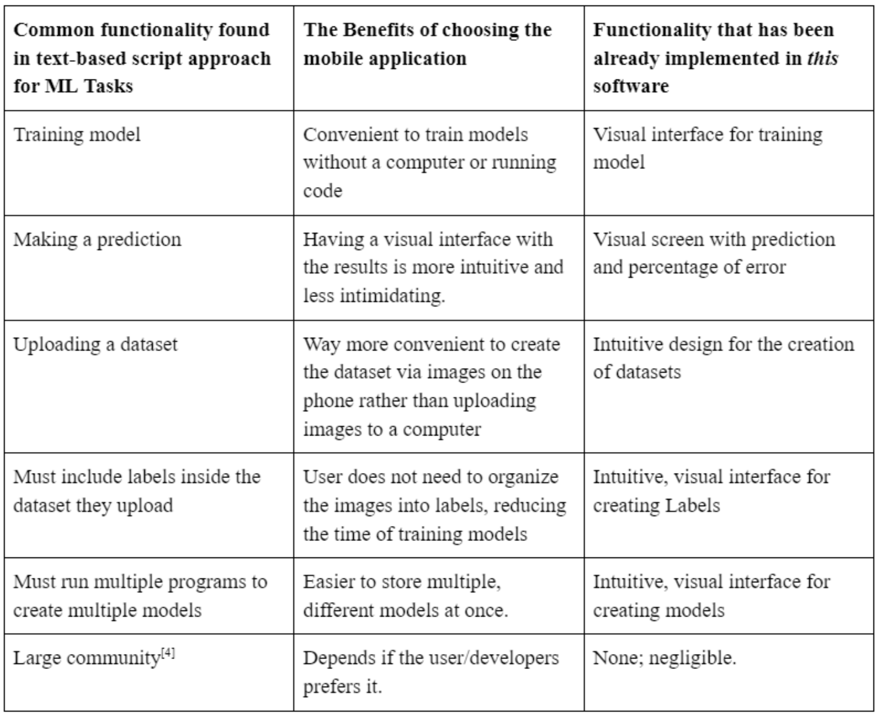


```
Container(
  width: double.infinity,
  child: ElevatedButton(
      onPressed: () {
        print("Training");
        http.get(Uri.parse(LocalDB.baseURL + "/train" + "/" + widget.modelId)).then((response) {
          print("Successfully triggerd the training.");
        });
      },
      child: Text(
          "Training!"
      )  // Text
  ),  // ElevatedButton
),  // Container
```

Figure 16. Screenshot of code 12

4. Experiment

4.1. Experiment 1



Common functionality found in text-based script approach for ML Tasks	**The Benefits of choosing the mobile application**	**Functionality that has been already implemented in *this* software**
Training model	Convenient to train models without a computer or running code	Visual interface for training model
Making a prediction	Having a visual interface with the results is more intuitive and less intimidating.	Visual screen with prediction and percentage of error
Uploading a dataset	Way more convenient to create the dataset via images on the phone rather than uploading images to a computer	Intuitive design for the creation of datasets
Must include labels inside the dataset they upload	User does not need to organize the images into labels, reducing the time of training models	Intuitive, visual interface for creating Labels
Must run multiple programs to create multiple models	Easier to store multiple, different models at once.	Intuitive, visual interface for creating models
Large community[4]	Depends if the user/developers prefers it.	None; negligible.

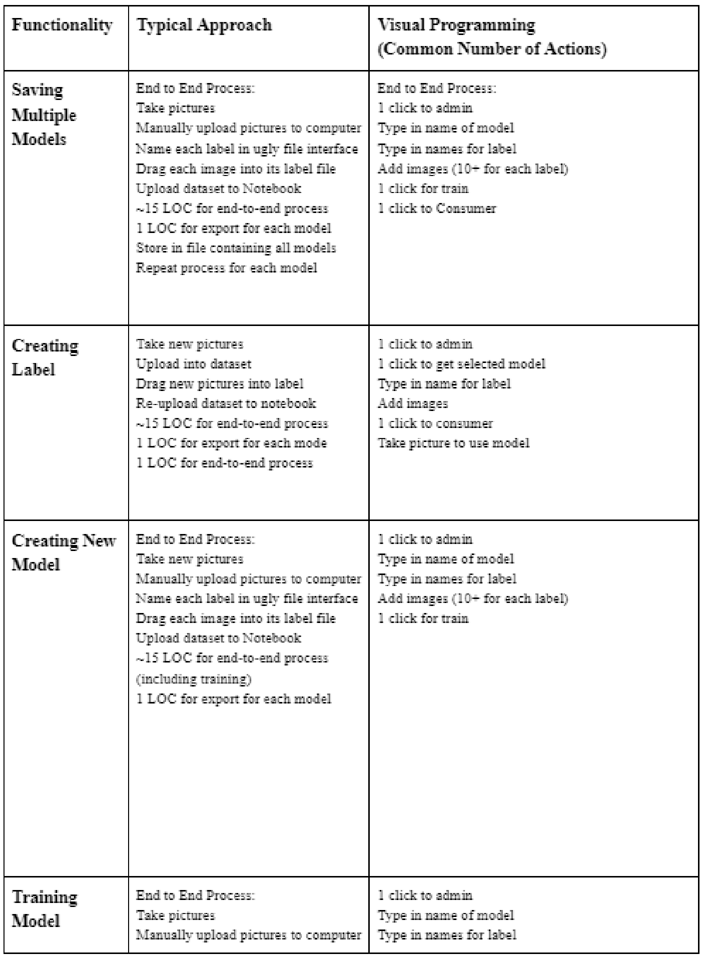
Figure 17. A qualitative test on common functionality

Figure 17 depicts a qualitative test on common functionality that is found in the typical script approach to generating image-detection models. We list such functionality and compare the differences between the typical approach and our mobile approach. While the approach can slightly differ, the purpose of the test is to ensure that we check the boxes in the standard functionality, including training, making a prediction, and utilizing data.

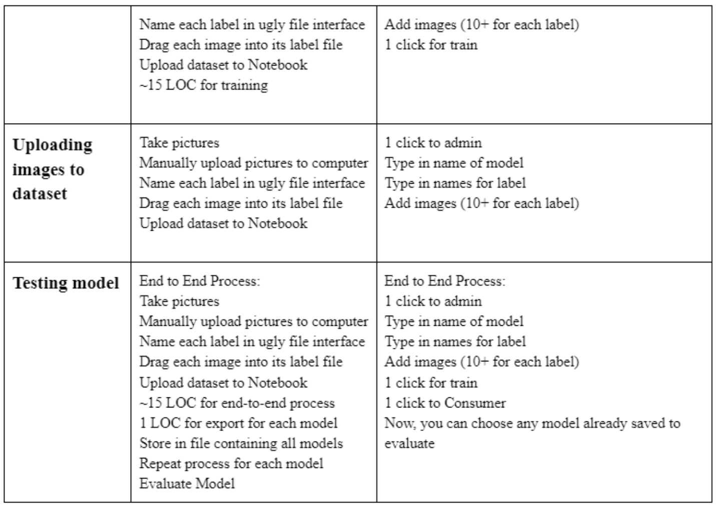
In order to be effective for domain experts to use, the application must include features and certain functionality that must be present in the typical approach . These include the abilities to train a model and predict the results based on the labels. However, our application also includes abilities that are generally not found in the standard text-based programming approach to machine learning, such as the ability to create custom datasets directly on the application. In this experiment, we attempt to compare our application's functionality with that found in a typical text-based script approach to image-detection. Here we can see that we have listed which functionalities are within both approaches and why our approach can be more beneficial for domain-experts with no experience in code. In every case, from training the model, making a prediction, uploading a dataset, and creating multiple models, our mobile application has a simple visual interface that makes the process significantly easier to navigate through.

4.2. Experiment 2

Figure 18 depicts a quantitative test comparing the lists of steps between the typical approach and the approach it takes to handle specific tasks within image-detection. In this experiment, we attempt to demonstrate that our approach is significantly less time-consuming, less tedious, and more intuitive. We will also remove any boilerplate code for the typical approach as implemented in the program is trivial.



Functionality	Typical Approach	Visual Programming (Common Number of Actions)
Saving Multiple Models	End to End Process: Take pictures Manually upload pictures to computer Name each label in ugly file interface Drag each image into its label file Upload dataset to Notebook ~15 LOC for end-to-end process 1 LOC for export for each model Store in file containing all models Repeat process for each model	End to End Process: 1 click to admin Type in name of model Type in names for label Add images (10+ for each label) 1 click for train 1 click to Consumer
Creating Label	Take new pictures Upload into dataset Drag new pictures into label Re-upload dataset to notebook ~15 LOC for end-to-end process 1 LOC for export for each mode 1 LOC for end-to-end process	1 click to admin 1 click to get selected model Type in name for label Add images 1 click to consumer Take picture to use model
Creating New Model	End to End Process: Take new pictures Manually upload pictures to computer Name each label in ugly file interface Drag each image into its label file Upload dataset to Notebook ~15 LOC for end-to-end process (including training) 1 LOC for export for each model	1 click to admin Type in name of model Type in names for label Add images (10+ for each label) 1 click for train
Training Model	End to End Process: Take pictures Manually upload pictures to computer	1 click to admin Type in name of model Type in names for label

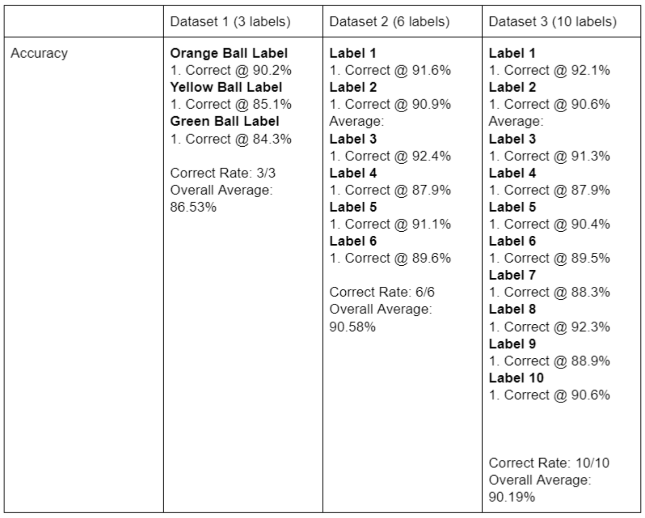


	Name each label in ugly file interface Drag each image into its label file Upload dataset to Notebook ~15 LOC for training	Add images (10+ for each label) 1 click for train
Uploading images to dataset	Take pictures Manually upload pictures to computer Name each label in ugly file interface Drag each image into its label file Upload dataset to Notebook	1 click to admin Type in name of model Type in names for label Add images (10+ for each label)
Testing model	End to End Process: Take pictures Manually upload pictures to computer Name each label in ugly file interface Drag each image into its label file Upload dataset to Notebook ~15 LOC for end-to-end process 1 LOC for export for each model Store in file containing all models Repeat process for each model Evaluate Model	End to End Process: 1 click to admin Type in name of model Type in names for label Add images (10+ for each label) 1 click for train 1 click to Consumer Now, you can choose any model already saved to evaluate

Figure 18. A quantitative test comparing the lists of steps

The results show that traditional text-based programming to accomplish any standard functionality is likely to be far more tedious, and requires a more technical understanding of both machine learning and programming. While in the typical approach we would need to use another piece of technology to gather images and then upload, we offer the approach of taking pictures on the mobile phone they are using to train and evaluate the model, making for a significantly less time-consuming process. Furthermore, important functionality such as evaluation and training requires utilizing code in the typical approach; we include the ability to do so with only a couple of clicks, typing, and taking pictures on the phone. This clearly reduces the knowledge threshold required to create image-detection models.

Figure 19 shows a quantitative test depicting the accuracy of the image-detection model used in our mobile application. We attempt to show that the difference between using the model in the text-based approach and our approach is negligible.



	Dataset 1 (3 labels)	Dataset 2 (6 labels)	Dataset 3 (10 labels)
Accuracy	**Orange Ball Label** 1. Correct @ 90.2% **Yellow Ball Label** 1. Correct @ 85.1% **Green Ball Label** 1. Correct @ 84.3% Correct Rate: 3/3 Overall Average: 86.53%	**Label 1** 1. Correct @ 91.6% **Label 2** 1. Correct @ 90.9% Average: **Label 3** 1. Correct @ 92.4% **Label 4** 1. Correct @ 87.9% **Label 5** 1. Correct @ 91.1% **Label 6** 1. Correct @ 89.6% Correct Rate: 6/6 Overall Average: 90.58%	**Label 1** 1. Correct @ 92.1% **Label 2** 1. Correct @ 90.6% Average: **Label 3** 1. Correct @ 91.3% **Label 4** 1. Correct @ 87.9% **Label 5** 1. Correct @ 90.4% **Label 6** 1. Correct @ 89.5% **Label 7** 1. Correct @ 88.3% **Label 8** 1. Correct @ 92.3% **Label 9** 1. Correct @ 88.9% **Label 10** 1. Correct @ 90.6% Correct Rate: 10/10 Overall Average: 90.19%

Figure 19. A quantitative test depicting the accuracy

The results show that overall, each prediction made by the model has been correct for models that have 3 labels, 6 labels and 10 labels. For 3 labels, we had an overall confidence of 86.53%. For 6 labels, we had an overall confidence of 90.58%. And for 10 labels, we had an overall confidence of 90.19%. The overall median of our results was above 90%, and increasing the number of labels did not decrease our model's accuracy. Our data can prove that the model works the same, and will display the correct result for an overwhelming majority of the time.

5. Related Work

kTrain is a low-code Python library that attempts to make the process of machine learning easier to program [16]. Using kTrain, tasks within the training that would normally require more lines of confusing code would be shortened using their libraries. Furthermore, the library makes each line of code more intuitive and allows the user to have an easier process when writing commands. kTrain, while simplifying the training, does not support users that don't know how to code. Our approach, on the other hand, gives the user the ability to train the model without having to write a single line of code. This gives an abstraction that opens up machine learning to everybody, not just those with a basic understanding of code.

Lobe AI is an application that allows users to gather testing data, train a model, and compute its results without having to write any code [17]. In addition, similar to our application, Lobe allows users to create their own dataset without having to export images. However, since Lobe AI is only supported on the computer, gathering such images using a webcam is not only inconvenient, but also limiting as some computers may not have a webcam that works. However, because our application is supported on smartphones, users can easily take pictures of the data using their phones and thus will have a better user experience.

Levity AI is a software that allows for the automation of images, text, and other documents [18]. By importing images or other pieces of training data, Levity is able to train a model based on such images. However, Levity is not only expensive, but it also does not give users the ability to create their own data-sets. This requires users to go through the time-consuming process of gathering images and exporting them. In contrast, our approach allows users to train models for free, increasing its usability and scope, and also allows users to create their own models based on the images they collect by their own phone camera.

6. Conclusions

In conclusion, my application allows the user to organize a list of models, train the models, create labels, and create datasets with a mobile phone. Using the images collected by the user, the application uses a server-side approach to train the model, allowing anyone to run tests using the models without having to train the model within the mobile app. Furthermore, we utilized HTTP APIs to add images to the database, get models/labels to load in our User Interface, and add models/labels to our database. We also designed a simple, easy-to-use UI that follows a simple procedure and isn't as convoluted as other similar applications. We conducted three experiments: one to evaluate the completeness of our approach, one to find the efficiency of our approach, and one to determine the accuracy of our approach. The results have shown that our application maintains the same accuracy and confidence as the typical text-based approach to train image-detection models. However, we have also concluded that we have a similar set of functionality with an easy-to-navigate UI and a dynamic structure that allows for easy modification of models. Similarly, we provide a simple approach to modify or add datasets so that the user can easily increase the amount of data used to train the model.

One limitation of our application is the time it would take to gather one image at a time. While taking photos on a phone and storing it to the dataset is already quite efficient, it will still be a tedious process to take pictures one at a time. Because an image-detection model generally requires hundreds of photos for it to be accurate, taking pictures would still be a time-consuming process. Furthermore, our approach only supports image-detection. As machine-learning encompasses other architectures such as data classification and object detection, our app can only be used to service a specific machine learning task.

We plan on creating a system that allows users to upload their own images to our app. Furthermore, we plan on introducing a video system that would allow the user to take pictures in batches at one time at high quantities. Both of these additions would allow the process of gathering data to be even more efficient and less tedious.

REFERENCES

[1] Flasiński, Mariusz. Introduction to artificial intelligence. Switzerland: Springer International Publishing, 2016.
[2] Bi, Wenya Linda, et al. "Artificial intelligence in cancer imaging: clinical challenges and applications." CA: a cancer journal for clinicians 69.2 (2019): 127-157.
[3] Kumar, Ashwani, and Ankush Chourasia. "Blind navigation system using artificial intelligence." International research journal of engineering and technology (IRJET) 5.3 (2018): 601-605.
[4] Albawi, Saad, Tareq Abed Mohammed, and Saad Al-Zawi. "Understanding of a convolutional neural network." 2017 international conference on engineering and technology (ICET). Ieee, 2017.
[5] Srivastava, Shrey, et al. "Comparative analysis of deep learning image detection algorithms." Journal of Big Data 8.1 (2021): 1-27.
[6] Carneiro, Tiago, et al. "Performance analysis of google colaboratory as a tool for accelerating deep learning applications." IEEE Access 6 (2018): 61677-61685.
[7] Jordan, Michael I., and Tom M. Mitchell. "Machine learning: Trends, perspectives, and prospects." Science 349.6245 (2015): 255-260.
[8] Thakkar, Mohit. "Custom core ML models using create ML." Beginning Machine Learning in iOS. Apress, Berkeley, CA, 2019. 95-138.
[9] Kluyver, Thomas, et al. Jupyter Notebooks-a publishing format for reproducible computational workflows. Vol. 2016. 2016.
[10] Sahay, Apurvanand, et al. "Supporting the understanding and comparison of low-code development platforms." 2020 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA). IEEE, 2020.
[11] Bojer, Casper Solheim, and Jens Peder Meldgaard. "Kaggle forecasting competitions: An overlooked learning opportunity." International Journal of Forecasting 37.2 (2021): 587-603.
[12] Louis, Marcia Sahaya, et al. "Towards deep learning using tensorflow lite on risc-v." Third Workshop on Computer Architecture Research with RISC-V (CARRV). Vol. 1. 2019.
[13] Windmill, Eric. Flutter in action. Simon and Schuster, 2020.
[14] Grinberg, Miguel. Flask web development: developing web applications with python. " O'Reilly Media, Inc.", 2018.
[15] Moroney, Laurence, Anglin Moroney, and Anglin. Definitive Guide to Firebase. California: Apress, 2017.
[16] Maiya, Arun S. "ktrain: A low-code library for augmented machine learning." (2020).
[17] García-Ortiz, Joselin, and Santiago Sánchez-Viteri. "Identification of the Factors That Influence University Learning with Low-Code/No-Code Artificial Intelligence Techniques." Electronics 10.10 (2021): 1192.
[18] Hughes, Larry W., and James B. Avey. "Transforming with levity: Humor, leadership, and follower attitudes." Leadership & Organization Development Journal (2009).

© 2022 By AIRCC Publishing Corporation. This article is published under the Creative Commons Attribution (CC BY) license.