American Dream

Martin Lahoumh
Brandon Tjandra
Jiazhou Zhang
Miguel Luna

Computer Science
CSc 59867 - Senior Design II
Yunhua Zhao

City College of New York
Grove School of Engineering

December 15, 2024

Abstract

The main goal of our project is to create a web application which takes an image supplied by the user that contains text of a specified language, identifies and extracts the text, and translates it to a user's preferred language (i.e. French, Spanish, Chinese, etc.). It will output the same image the user gives and overlay the translations on top of it.

The machine learning aspect of this project is both detecting where in an image text is located, and then extracting the text from the image into a code-friendly format, like a string: this is done in the form of an OCR pipeline. For this project specifically, we made use of EasyOCR, which is a full OCR pipeline implementation using multiple pretrained computer vision models, and googletrans, which is a Python wrapper for Google Translate. The application was built in React and uses a Flask backend for OCR incorporation.

While our final application does successfully invoke the OCR text extraction and perform the translation, it is still subject to a number of bugs relating to the text overlay color and size of the output image. If we are to revisit this project, we would like to address these bugs, as well as improve the model by customizing it to improve its accuracy when given images with more complex text transformations.

Introduction

One popular application for machine learning and artificial intelligence is using it to read text in images. A machine learning model used for this context is called an optical character recognition model, or OCR. Among the use cases of OCRs, one found to be among the most useful, innovative, important, and highly demanded was for image text translation. Up to this point, text translation could only be done with text strings, so if you wanted a translation from an image, you would have to transcribe the text manually.

The usage of OCRs for photo translation dates back to as early as 2010, with an augmented reality application for smartphones called “Word Lens”. Since then, the technology and company behind it would be acquired by Google in 2015 to be incorporated into their Google Translate application and services. With the shift to smartphones, most people have access to a digital camera at all times, and photo translation has become a popular and necessary component for any translation application following. In recent times as machine learning and artificial intelligence becomes more sophisticated and relevant in everyday purposes, photo translation apps have seen more and more improvements in the form of speed, accuracy, and additional features.

The United States of America is well known for being a country full of opportunity for immigrants and a diverse range of cultures that intermingle and form new ones, coining the famous “melting pot” analogy. With all of us being children of immigrant families and some of us being first-generation college students, that concept of the “American dream” is very important to us. One of the biggest issues that people face when travelling or immigrating to a foreign country is the language barrier: not only is it difficult to speak with local residents, but not being able to read any text of the local language can be a major hindrance to doing a lot of things in the area, sometimes even being for fundamental aspects of our lives, like transportation, government, or access to facilities. This is why us and many others think that quick and easy access to translation services are so important, and that the use of machine learning can not only solve this specific problem, but contribute to the wider idea of the American dream.

The main goal of our project within the course of this semester is to create a web application which takes an image supplied by the user that contains text of a specified language, identifies and extracts the text, and translates it to a user's preferred language (i.e. French, Spanish, Chinese, etc.). Like most existing applications nowadays, it will output the translations overlayed on top of the original image. The machine learning aspect of this project is both detecting where in an image text is located, and then extracting the text from the image into a code-friendly format, like a string: this is done in the form of the OCR pipeline. The OCR and translation functionality are the focus for the minimum viable project and the focus for the development time of this project; unless we have additional time and ideas and have a

satisfactory amount of work completed past this goal, we do not intend to add any additional features or technologies.

This report will proceed into a literature review that details two research papers relevant to the development of this project, summarizing them and explaining the key takeaways that were used towards the creation of our application. Next, we will detail the technologies and methods that were used in the programming technical development of the application, including a section of which is dedicated specifically to the OCR model. Lastly, we will discuss the result of the completed product, our findings and reflections, potential issues and/or improvements, and the broader implications of the application for wider use.

Literature Review

In terms of existing projects that are similar to ours, there are, of course, a multitude of various image translation applications that are available for public use, most notably Google Translate. However, the machine learning algorithms and models that Google and all other companies with similar services use are not public. During research, we found two research papers that have shared their approaches and difficulties with creating an OCR model designed for scene text detection and identification, both of which have essential components and ideas that were used towards the design of our model.

The first research paper we will cover is one that proposes a scene-text detection model called CRAFT, which is short for “Character Region Awareness for Text Detection”. CRAFT is a neural network that aims for improved text detection by focusing on character-level detections rather than whole word detections. This is because words in scene images have several different kinds of criteria that they can be detected by, such as color, space between characters, or meaning. Additionally, word segmentation boundaries that can break a word into individual characters cannot be strictly defined, so the model cannot create a significant annotation meaning for what it has identified.

The character level detection model they propose uses two types of bounding boxes: a character box that identifies the boundaries of a detected character, and an affinity box that identifies the space between characters in order to form a word. The model is trained to detect character region and affinity between characters using scene text images from datasets like ICDAR with rectangular ground truth bounding boxes, and datasets like MSRA-TD500 that use polygonal bounding boxes for curved text instances, and is evaluated using IoU scores compared with the ground truth. Their experimental results suggest that the CRAFT model compares favorably to other existing text detection models, and is able to perform well at multiple image scales without fine-tuning, but has difficulty identifying Arabic words because individual characters cannot be easily identified.

CRAFT is most relevant to this project because it is a component of the pretrained OCR pipeline that we will be using for this application. Judging by the evaluation metrics shown by the paper, we can assume that this will be the most effective text detection model available for public use that our app can harness, especially compared to that of our own attempts.

The second article, called “What Is Wrong With Scene Text Recognition Model Comparisons? Dataset and Model Analysis”, is one that analyzes the challenges and inconsistencies in comparing scene text recognition (STR) models due to variations in datasets and evaluation methodologies. Performance gaps of up to 0.8% can occur due to the inconsistencies in each dataset version, which makes it challenging to compare the performance of a model per use case. This study introduces a unified four-stage STR framework, consisting of transformation, feature extraction, sequence modeling, and prediction. This modular framework provides a consistent way to evaluate STR models and explores unexplored module combinations to improve accuracy, speed, and memory efficiency.

MJSynth and SynthText images were used to train the STR models. MJSynth consisted of 8.9M images and SynthText consisted of 5.5M images. In total, this gave the model 14.4M images to train on. The results were that The combination of the MJSynth and SynthText boosted the performance of the model by 84.1%. This led to the conclusion that a more diverse dataset is more significant to better results than simply a large quantity of datasets. This comes with its

tradeoffs, as models see lower speeds and more consumption of memory as extra modules were introduced.

The takeaway here for this project is that making our own scene text recognition model would pose challenges of inaccuracy and performance given that there is a lack of effective training datasets that would simplify the process. The authors here suggest that using synthetic text datasets would be the most effective solution when making a text recognition model, however we have had further difficulty either using these datasets to generate text or just use them given their large size which causes performance problems when preprocessing or training, as well as finding other suitable datasets that will be effective as training material, as the ones we could find are for more niche purposes, like EMNIST which only has data for handwritten characters and digits, or only serve to hinder the model's training when identifying text and cause overfitting, in the case of most scene text datasets like ICDAR.

It is clear that we would be running into the same problem if we were to attempt developing our own text recognition model, and this proved to be true in the two times we attempted to do so. Therefore, we came to the conclusion that the most effective model we could feasibly use (in terms of time and resources) for this project would be one that's already pre-trained and has satisfactory evaluation metrics, which we will discuss shortly.

Methodology

The main component of this project lies in the OCR. An OCR, which is short for optical character reader, is the process of converting text, be it printed, handwritten, physical or digital of origin, in an image to machine-encoded text, something the computer can encode. While it is common for OCR to be used for processing and preserving print media digitally as well as for other applications like text-to-speech and machine translation (something our project will also require), it is notable for being a field of research in artificial intelligence and computer vision, and this project is no different. We will be requiring an OCR for this project because it will be necessary for text detection: identifying the location of text in an image, and text recognition: deciphering the text that the image is known to contain.

It is referred to as an OCR pipeline because it actually incorporates several kinds of separate machine learning models into one: that being a text detection model, which locates text in an image and provides coordinates for it, and a text recognition model, which extracts text from an image into a format readable by a computer, such as a string. The text detection model identifies the location of text in an image and creates the bounding box coordinates, and then the text recognition model uses those coordinates to read the text within. Once the model is downloaded and included with the project files, EasyOCR can be initialized and used with only a few lines of Python code.

From our search for a suitable OCR model, we decided to go with the free and open-source tier of EasyOCR. EasyOCR contains a full OCR pipeline implementation. The process for extracting the text in the image is shown in Figure 1: first, the image is preprocessed and supplied to the text detection model, which is the aforementioned CRAFT model by default. CRAFT performs its inference, and produces bounding box coordinates on where it predicts the text to be in the image. Next, the data is processed again where it is passed into the text recognition model. By default, it uses ResNet (residual neural network) with LSTM and CTC and passes it into a greedy decoder, all of which can also be swapped out for different text recognition models and/or decoders. Lastly, the image is post-processed and the output is returned.

EasyOCR provides us with a few advantages that will help us specifically with the end goal of this project. One reason is that EasyOCR returns the bounding box coordinates of each of the text it recognizes in the image. This will help us overlay the text translation over the supplied image from the user. The second reason is that EasyOCR has been trained to recognize a variety of languages besides English when scanning an image. This gives us the capability to introduce translation from other base languages besides English, which we were not expecting to do. As of this project's completion, EasyOCR has support for over 80 languages, including those that don't contain alphabetical characters, such as Chinese, Korean, or Arabic.

Additionally, we used some datasets to test the effectiveness of EasyOCR. These datasets were ICDAR-2015, which contains 1,000 scene text images, and TextOCR, which contains around 25,000 images and 80,000 annotations on those images. Given that these datasets were

not built manually, but were instead created using OCR models themselves, we processed these datasets to check for entries that contained text that wasn't legible and/or valid English text according to an English corpus. We then passed every image through the model to get an inference, and for each inference we used intersection over union scores (IoU) to then evaluate the model using precision, recall, and F1 score.

For the translation aspect of this project, we were originally going to use a translation API that is called every time the model is invoked. However, we realized this wouldn't be possible because most translations we found, including our first choice, Google Cloud's translation API, required costs to use that would make this project impossible to use over a long term if we were to deploy.

Thankfully, we found an alternative in the form of a Python package called "googletrans". As the name suggests, it is built on top of Google Translate, but instead of using Google's API, it is simply a wrapper for the Google Translate website. In addition to being a fully functional implementation of Google Translate in Python that can perform translations in only a few lines, it provides a confidence score between 0 and 1 for each translation, which helps us gauge the accuracy of the translation. Of note is that it is a third-party package and not endorsed or licensed by Google to any official capacity, which may raise concerns of its usage and performance; however, we ourselves have not found any issues using it for this project.

Lastly, we used React for the application frontend interface and Flask for the backend interaction with the model and translation package. We used React because it's what we are familiar with for web development, and Flask because we need the website to interact with Python code.

Implementation

At the beginning, our team attempted to construct our own OCR model based on EasyOCR. EasyOCR has two models to get it to work: one is a detection model which detects where text in the image is located at, giving the bounding boxes for all appropriate text. The

second model was a recognition model that would be able to take in the text detected and identify each character in the text, giving the text as a result. Our team started off with the recognition part, as we thought it would be best to determine the course of our work. If the recognition wasn't good, then there would be no point in continuing with our original plan as that was the most important part of this project. As we were dealing with detection, it made sense to implement a CNN model.

For the dataset, we trained the model on the EMNIST dataset as it seemed to be the most popular amongst text detection. An issue occurred while training where the program kept crashing due to the large size of the dataset. It was decided to try cutting the dataset, so instead of using all english characters we will use from A-J. This proved to not be helpful as the dataset was still too big for the computer to handle. We then shrunk the dataset even more, this time taking samples from each character (A-J) out to make it more workable. This worked as the program wouldn't crash when training. However, it seemed to not give enough for the model to be trained effectively as the results after training did not meet our expectations. For starters, the accuracy behaved poorly coming at around 0.01. Sample images given to it, from EMNIST dataset and other images, all did poorly as the characters would either be guessed incorrectly or not guessed at all.

From here, we had to make a quick decision with the time we had left: one option was that we could take more time, with the 2 months we had left, working on this model to try to make it satisfactory. This would not only mean that we would continue working on the recognition part of the model, but afterwards we would need to start from the beginning and move on with the detection part of the model. The other option was to simply use EasyOCR, which was already trained and worked really well, and implement it into our web app. We decided, for the sake of time, that using EasyOCR was the best option. After being granted permission to use EasyOCR from our professor, we implemented it.

Our application runs on React (frontend) and Flask (backend). Most of the work is handled in the backend. For users to upload their own images into our application, we have a /upload route. It checks for the image that was sent to it, saves it onto a folder called ‘uploads’,

creates an image path for it and saves that image path, returning the image path to the front end. To retrieve this image on the frontend, a /uploads/pathname route was created, as we are dealing with images a lot so calling it simple seemed to be helpful. The final route was the /translate route that handles taking the image and actually running it through the model. It starts off by getting the form that the user sent in the front end (the image path, and the language wished to be translated to.) From the multiple tests we did with EasyOCR, we saw that it doesn't read a regular image path. Due to this, we had to clean it to its appropriate condition.

```
absolute_img_path = os.path.join(os.getcwd(), img_path.lstrip('/'))
```

We then ran the absolute path to the EasyOCR model and stored the results. The point of our app was to not just give the translation, but also give the original image with the translated text placed to its appropriate position. One original problem faced trying to do this was the type of image being used (such as JPEG) A fix was to check if the image has transparency (RGBA) and convert it to RGB, which fixed the error. The image is then darkened so that when the text is placed on it the user can read it. Since the results of the EasyOCR model is stored in an array, we cycled through all, storing the x and y coordinates for the bounding boxes and the translated text. When displaying the text on the image, there are a few factors to consider. We will need to know where exactly we should place the text, hence the use of the bounding box. We also will need to know how large the text has to be. Some signs have bigger text sizes than others, so adjusting the size as necessary is important. Finally, we will need to match the color with the original text color of the original image. To get the font size, we found the height of the bounding box.

```
box_height = max(y_coords) - min(y_coords)
font_size = int(box_height * 0.8)
```

Next, after taking the translated text, to get the color of what the text should be is by taking the RGB value of the center of the bounding box and getting the color of the pixel at the center. Originally, the way the font color was taken was by finding the average RGB value of the bounding box containing the letter, but this proved to be unreliable. Most of the time, the text

would be the same color as the sign since there would be more background than text. After the color, we placed the translation in the center of the bounding box and sent the image and translations back to the front end to be displayed.

```
text_bbox = draw.textbbox((0, 0), translated.text, font=font)
text_width, text_height = text_bbox[2] - text_bbox[0], text_bbox[3] - text_bbox[1]
text_position = (
    (top_left[0] + bottom_right[0]) / 2 - text_width / 2,
    (top_left[1] + bottom_right[1]) / 2 - text_height / 2,
)
draw.text(text_position, translated.text, fill=text_color, font=font)
```

Testing & Results

As previously mentioned, we first tried evaluating the model ourselves using IoU scores using the datasets ICDAR-2015 and TextOCR. The results of the inference testing is shown with Figures 2-5, where we can see that with ICDAR-2015, around 2,000 bounding boxes are created using a 500 image subset, and with TextOCR, around 14,000 bounding boxes are created using a 1,000 image subset. For the ICDAR-2015 testing, the average confidence score for the inferences came up to 0.29, and for the TextOCR testing, the average confidence score for the inferences came up to 0.26. This tells us that, for more complex scene text examples used in these datasets, EasyOCR doesn't do a particularly good job in finding text and/or accurately reading the text from the detected bounding box instances.

Next, we will discuss the results of the IoU scores, starting with the case of ICDAR-2015. In terms of the IoU scores, precision is the ratio of correctly drawn annotations to the total number of drawn annotations. With every predicted bounding box, the model has a precision score of about 55.75%, meaning it is correct in matching a truth bounding box by that percentage. Recall is the ratio of correctly drawn annotation to the total number of ground truth annotation. With every truth bounding box, the model is about 60% correct in predicting a matching bounding box. F1 score gives us a harmonic mean between precision and recall. The score is about 57.7%. As for TextOCR, we used a larger subset compared to ICDAR, which itself

is only a small fraction of the actual dataset. With more data, it performs much worse overall: Precision of 55.19%, recall of 35.43%, and F1 score of 21.58%. Precision remains about the same, but recall, and F1 score as a result, take a very big hit. There are a lot less matches when it comes to making bounding box predictions. While about a quarter of the predicted bounding boxes in ICDAR-2015 were incorrect, about half of the predictions are incorrect here.

The very low accuracy is likely due to the high difficulty of the datasets. The model does not perform well with image text that is transformed (rotated, warped, blurred, etc.) or uses an unorthodox font. Most images in the datasets are like this, hence the low average confidence and IoU scores. However, many existing translation apps aren't typically expected to work with such images, so we don't find it to be a major problem. Clearer datasets may be more indicative of the model's intended performance. EasyOCR does not have full support for detecting handwritten text as of now (which may explain the font issues), which may present problems when using the model in our application.

As for the resulting application, it mostly works as intended. However, there are a few bugs regarding translation and the output image from the inference that will be discussed here. The issue with the translation feature is that sometimes, the word just cannot be translated to another language. This is replicable and can be due to a multitude of issues, but it only really applies when the text is translated to certain languages on certain images: the results may vary depending on which language is requested. If this ever happens, we will provide the string along the lines of "[translation unavailable]" as a failsafe, otherwise the translation will crash the application. It is still currently unclear as to why this occurs with certain languages, and whether this is an issue with the model, translation package, backend script, or something else entirely, and as such is a problem that we may not be able to properly debug.

The way the output image is produced using text overlays also feature a few errors that prevent it from having visible translated text. The first is the way that the color of the overlaid text is chosen. We have it implemented such that it chooses the color of the very center pixel within the text's bounding box as the font color, however this doesn't work all the time and might instead choose the color of the background instead of the actual text, making the translation very difficult to see. Additionally, the font of the text being used to overlay the

translations on the images may not support all non-alphabetic fonts, leading to characters being represented as empty ("□"), even if the translation is correct in the outputs below the images. Lastly, the way the font size is chosen for the output image leads to cases where the overlaid text "spills" out of the border for the output image, becoming cut off and not showing the full translation. Unlike the previous problem with the translation, however, these issues with the image are much more within our control and are things that we have determined solutions to, and/or will consider if we are to continue working on improving the application.

Discussion

Within the allotted time for this project, we were at the very least able to accomplish our minimum goal of creating the application, with working functionality for picture upload and camera use, text detection, and text overlay output. To this extent, we are mostly satisfied with the end product and its functionality, aside from a few bugs mentioned previously. Perhaps the biggest disappointment, though, is the fact that we used a pretrained model rather than developing our own, which felt unfulfilling.

That being said, all of us in this group have had no experience with machine learning development until this class. Even if our application uses a model we did not train, it was a big learning curve for us. Given the time and resource constraints, we likely would still not have enough time to complete our application without using pretrained models, or at the very least accurate ones. If we could attempt this project with more time and the knowledge we have learned now, we may be able to implement a full OCR pipeline from scratch. If not, we can always just attempt to customize the pretrained models in EasyOCR by training them ourselves using data specifically for curved and/or transformed scene text, which it currently struggles with.

In terms of other considerations for this project's use, it is true that this application's functionality isn't exactly original. In fact, it may be considered a ripoff of existing translation applications, most notably Google Translate, which this project actually harnesses the technology of for translating text. Given the limited scope and the nature of the project given the context of

our team, we don't think that this will pose any problems of intellectual property theft, copyright infringement, or anything of the like. As for other ethical and social concerns in the same vein when developing this project, all sources and technologies that were used are verified to be open-source and free for public use.

Besides model customization, if we are to improve this project, our first steps would of course be to tackle the bugs that we discovered above. In the longer term, however, we would also like to add user account functionality and create a mobile version of this application. User login was planned at one point of this project, and would have been used to allow users to save previous translation inferences, but was dropped because of model prioritization given time constraints, being seen as unnecessary at the time but could be an interesting addition in the future. As for the mobile application, we thought about doing this as taking a picture would be more suitable for smartphones to do rather than from a computer. We also think that this may be easy to do because the application was built from React, so porting it to mobile through react-native shouldn't be challenging.

Conclusion

To conclude, we have completed what we set out to do, which is to make a photo translation application using an OCR machine learning model. While the application is indeed finished and we are satisfied with the effort we placed into researching, developing, and testing it, there is work to do not only in fixing bugs and improving the model, but also incorporating original work in the form of developing a model ourselves, which may not be nearly as accurate as EasyOCR, but would be even more accomplishing and fulfilling to considering this project our own, especially after a full year of preparation for this project.

We previously stated that this project is fairly small in scope and ambition given our current knowledge and resources, so this project most definitely won't have a greater impact on the world as a whole, especially given its inferiority to much more mainstream and popular translation apps out there. However, for us, this app marks a big turning point for us as we begin to really learn more about machine learning and artificial intelligence, so to that extent we hope it will be the start of something much greater.

References

- Baek, Y., Lee, B., Han, D., Yun, S., & Lee, H. (2019). Character Region Awareness for Text Detection. *CoRR*, *abs/1904.01941*. Retrieved from http://arxiv.org/abs/1904.01941.

- Baek, J., Kim, G., Lee, J., Park, S., Han, D., Yun, S., … Lee, H. (2019). What is wrong with scene text recognition model comparisons? dataset and model analysis. *CoRR*, *abs/1904.01906*. Retrieved from http://arxiv.org/abs/1904.01906.

- Jaided AI. (n.d.). Jaidedai/EasyOCR: Ready-to-use OCR with 80+ supported languages and all popular writing scripts including Latin, Chinese, Arabic, Devanagari, cyrillic and etc.. https://github.com/JaidedAI/EasyOCR/tree/master.

Appendix

This section contains images referred to by the main text.

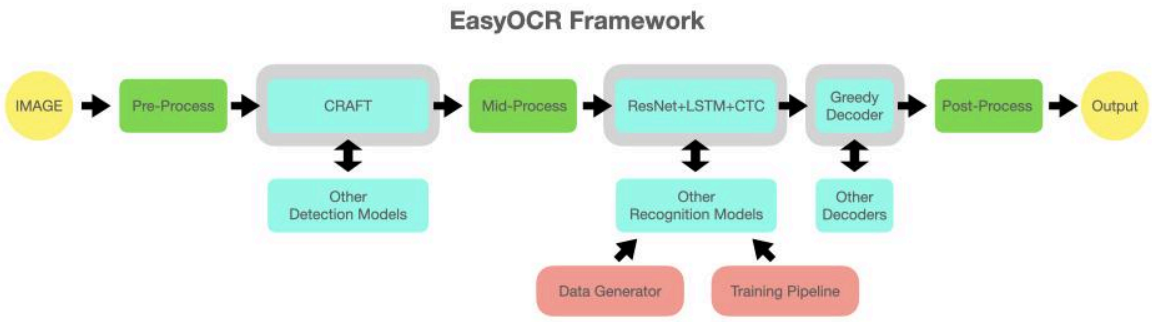

Figure 1: EasyOCR framework pipeline implementation.

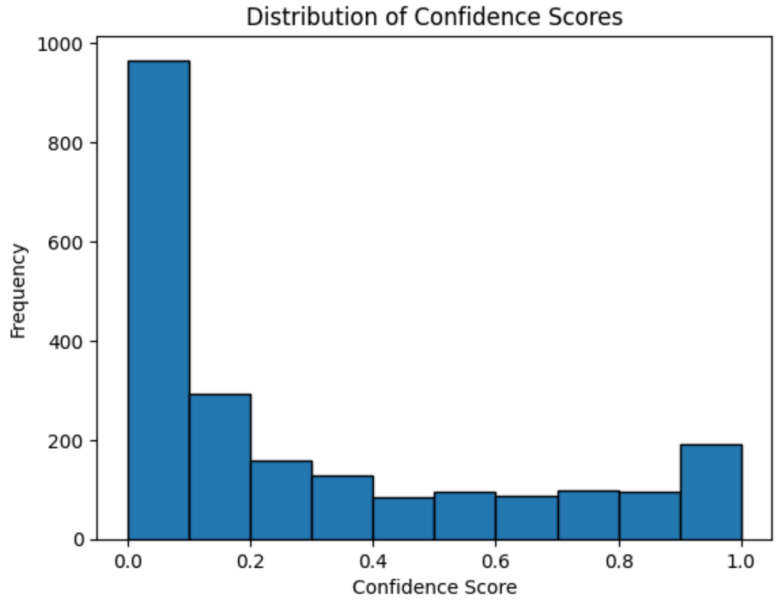

Figure 2: Distribution of confidence scores from EasyOCR using ICDAR-2015.

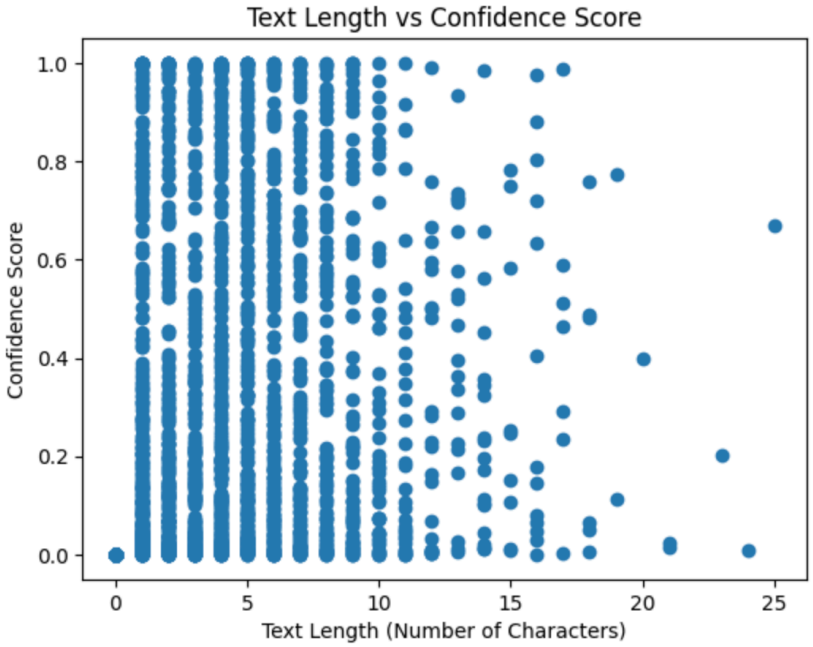

Figure 3: Text length vs. confidence score from EasyOCR using ICDAR-2015.

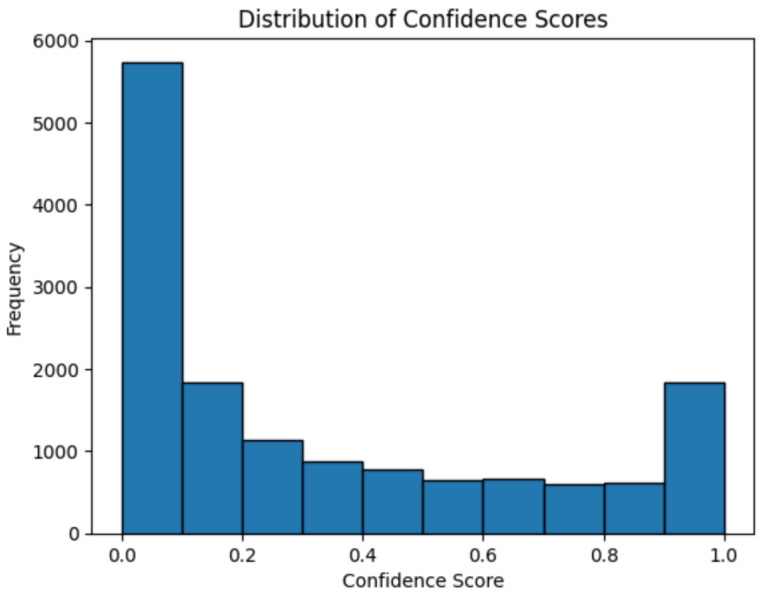

Figure 4: Distribution of confidence scores from EasyOCR using TextOCR.

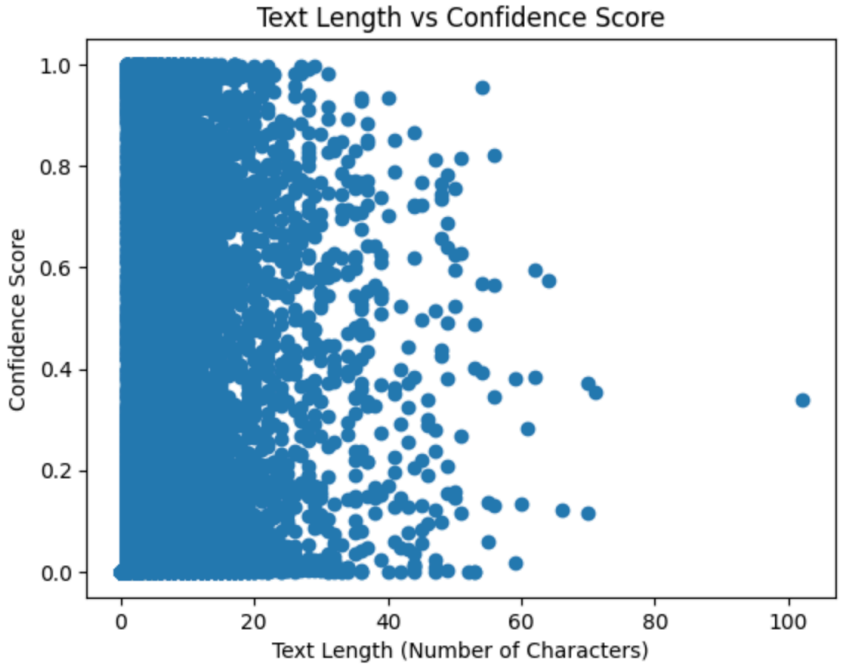

Figure 5: Text length vs. confidence score from EasyOCR using TextOCR.