Machine Learning Engineer Nanodegree

Capstone Project

Christel Ilaka Gipangu

November 3rd, 2018

Automatic License Plate Recognition and Search System

I. Definition

Project Overview

Few years ago, I was amazed by the way agents on a CSI TV-show could track a car based on certain features of the car. At that time, it was all science fiction for me but things have changed after taking the Machine Learning Nanondegree at Udacity. Now I know that there is a field named “Computer Vision”; techniques and methods develop in this field allow computers to see and recognize objects that are present on an image or video. When these methods are combined with machine learning algorithms, amazing things happen like finding the license plate of a car on a given image or video and tracking it like in a sci-fi movie. Our capstone project is in the field of computer vision because we are dealing with images that must be processed to extract some information that will be used as input of the machine learning algorithm used here.

Problem Statement

In this project, we want to extract the license plate number (LPN) in plain text format from an image that will then be used to query a database. To get the LPN in plain text the algorithm must classify each character presents on the license plate; thus, we are dealing with a classification problem here. Classification is a supervised learning approach in which the computer program learns from the data input given to it and then uses this learning to classify new observations ([1]). This data set may be bi-class or it may be multi-class as it is for this project. There are many classification algorithms in machine learning such as support vector machines, decision trees, random forest, neural networks, logistic regression, and so on.

To solve our problem, we will use computer vision’s methods and techniques to describe the license plate image, in other words, we will take the image of a license plate, quantify it using the block-binary-pixel-sum methods and then return a feature vector that abstractly represent its content. This feature vector will then be used by the classification algorithm to learn the characteristics of each character so that in the future we could classify the unseen data. We will start by fitting 3 classifiers (a Neural Network, support vector machines, logistic regression) then we will choose the one that gives us a better result in terms of precision as our final model.

---

[1]

Metrics

For this problem, we want to focus more on precision, thus, we picked **f-beta score** as our metrics. The F-beta score is a F-measure defined as a harmonic mean of precision and recall ([2]).

$$F_\beta = (1 + \beta^2) \cdot \frac{precision \cdot recall}{(\beta^2 \cdot precision) + recall}$$

Beta (β) is a parameter that controls a balance between precision and recall. When $\beta = 1$, F_1 comes to be equivalent to the harmonic mean of precision and recall: $F = \frac{2 \times precision \times recall}{precision + recall}$. If β>1, F becomes more recall-oriented and if β<1, it becomes more precision-oriented, e.g., $F_0 = Precision$.

Precision answers the following question: out of the points we have predicted to be positive, how many are correct? While recall answers this question: out of the points labelled "positive", how many did we correctly predict?

$$Precision = \frac{True\ Positives}{True\ Positives + False\ Positives} \qquad Recall = \frac{True\ Positives}{True\ Positives + False\ Negatives}$$

Since we want to use the license plate's number in plain text to query the database, this extracted number must be precise otherwise we won't find the license plate number in our database. Now we know that when beta is close to zero we will get a score close to precision, thus, we will use beta equals to 0.5 in this project.

II. Analysis

Data Exploration

Creating an automatic license plate recognition and search system is quite challenging because each country, even cities in the world have their own license plate's design; thus, there is no one fit-all system that would work everywhere. For this this reason, we need to create our own system based on the Congolese license plate design patterns. However, there is no public dataset available that we could use to build our system. For this reason, we had to create our own dataset.

We received a lot of pictures from friends and families but only selected 110 images and took 20% of these images to create the final testing set. The other images were used to create the main dataset. We also used some fonts that we downloaded from the internet to add to the dataset. Down here we detail how we created the dataset:

[2] Yutaka Sasaki, (2007), *The truth of the F-measure*, School of Computer Science, University of

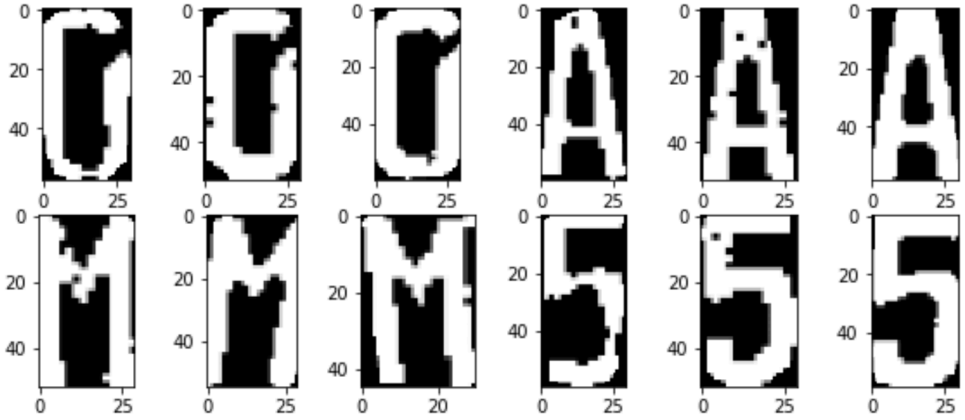
1. Run the *trainLP.py* file; what it does is looping over each image in the images folder. For each image, the license plate is found and each character on it is extracted and displayed on the screen, now we must press the key that correspond to the character. Once the key has been pressed, the script creates a folder with the name of the character and save the character's image in png format inside and continue. The figure down here shows this process.
2. The *trainFonts.py* does the same thing as the trainLP.py but here we use it for the fonts folder.

The final dataset is found in **char_Train** folder that contains 36 folders (one for each character) the images stored in these 36 folders have a dimension of *29xM pixels*. The *M* here means that the width of characters varies, in other words all characters have a height of 29 pixels but different width (for example we have 29x51, 29x52, 29x61, and so on).

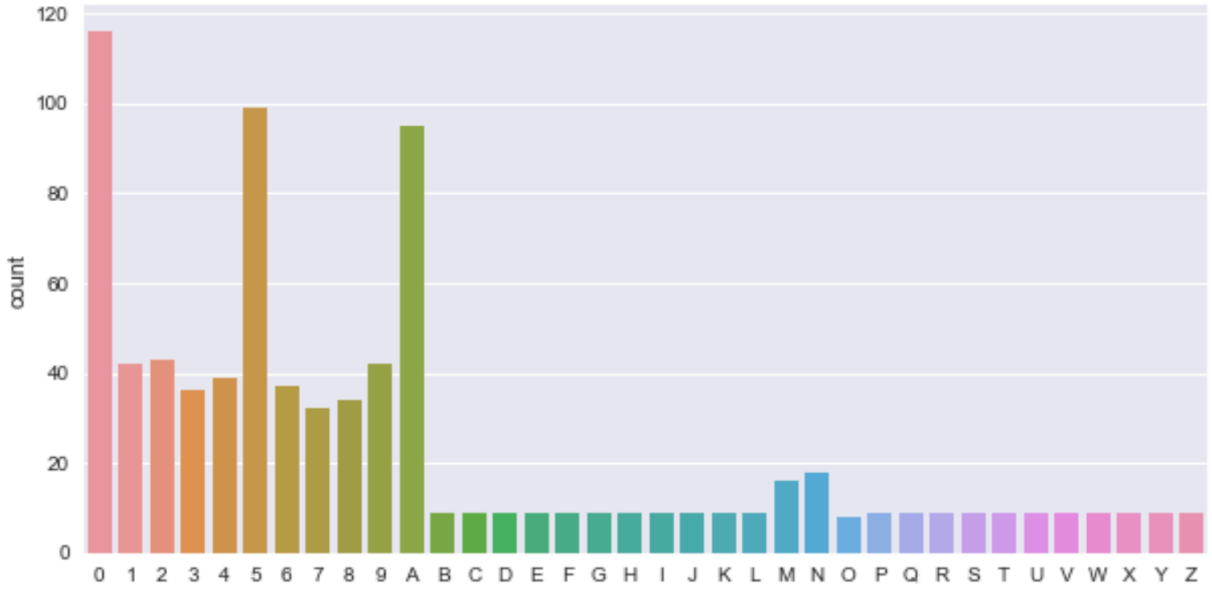
In total, we have **855 characters** in our dataset.

Visualizing some data

Let start by plotting some images of characters that are in the dataset. The figure down here plots the character 0, A, M, and 5.



Now let see how many images each category (character) has.

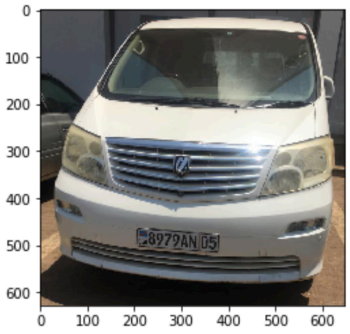


We can see that some characters have more data than other, this is the case for mostly the digits' characters and A, M, N. This makes a little bit sense because most of the license plate of the city of Lubumbashi end with 05 and contain A as the first letter. That is why when we were creating the dataset we encountered a lot of 0, 5, A, 2, 9 and so on as shown up here.

How about the actual car images? Down here we have 4 plots of cars randomly selected.

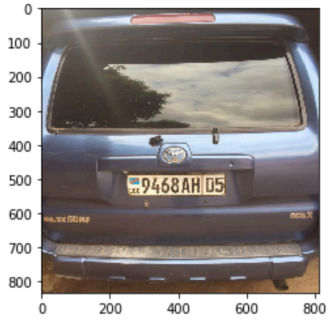
```
for i in np.random.choice(96, 4):
    image = cv2.imread(images_paths[i])
    title = images_paths[i][7:15]
    print(f"The shape of the car {title} is : {image.shape}")
    printImage(image)
```

```
The shape of the car 8979AN05 is : (628, 647, 3)
```

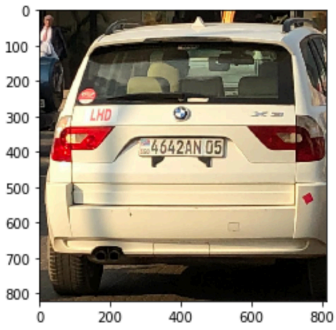



```
The shape of the car 2683AN05 is : (519, 688, 3)
```

The shape of the car 9468AH05 is : (833, 810, 3)



The shape of the car 4642AN05 is : (821, 810, 3)

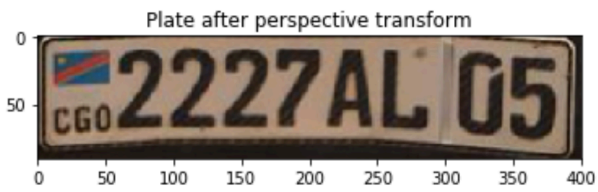


Visualizing these four images, we can see that each car has different angles and sizes. These abnormalities can affect the feature extraction step and thus the model during the prediction.

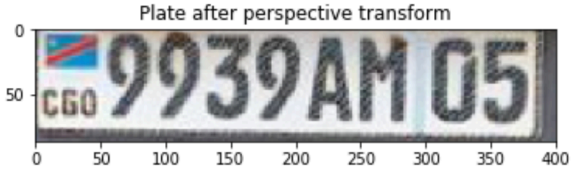
Before any feature extraction we will have to reshape the interested license plate regions to have a uniform width and angle like down here:

(79, 400, 3)

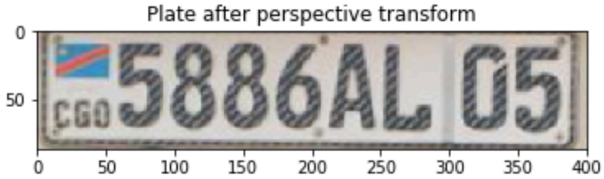
(90, 400, 3)



(86, 400, 3)



(86, 400, 3)



Algorithms and Techniques

The main problem here is to extract the license plate number in plain text. We can accomplish this in 3 steps:

1. Localizing the license plate in an image
2. Segmentation of characters from the license plate
3. Recognition (classification) of characters

Step 1: A license plate can be localized in an image by using either a machine learning method or a combination of morphological operations and contours techniques. We will use the second method because we don't have a large dataset to use the machine learning method like deep learning. Morphological operations are simple transformations applied to binary or grayscale images. These operations are applied to shapes and structures inside of images.

In our case, we apply morphological operations to reveal regions that could contain a license plate.

The contour properties are used to prune the license plate candidates, leaving only regions of the image that are highly likely to contain a license plate (more details on section 3.1.1. getRegions function).

Step 2: In this step we take the license plate regions that were found in step 1 and extract the foreground license plate characters from the background of the license plate. This is done by calling the ***charCandidates function***. To segment the license plate characters, we apply adaptive thresholding to each local 29 x 29 pixels region of the image. Thresholding is the process of converting a grayscale image to a binary image where the pixels are either 0 or 255. This technique works as follow: we select a threshold value T, and then set all pixel intensities less than T to zero, and all pixel values greater than T to 255. Once we have the thresholded license plate region, we apply a connected component analysis which determines the connectivity of "blob-like" regions in a binary image. Then we apply some contour properties to determine if the contoured region is a character candidate or not. The characters' candidates are cut from the binary, thresholded image using the ***scissor function***.

Step 3: To classification each character in a license plate, we start by describing them using the block-binary-pixel-sum method (more details on section 3.1.4. Feature extraction). Each character is then classify using a classification algorithm. The classification algorithms used here are: a logistic regression, a support vector machines, and multilayer perceptrons.

Logistic regression model uses the techniques of the linear regression model in the initial stages to calculate the logits (score) and in the later stages uses the estimated logits to train a classification model. It is used to predict an outcome variable that is categorical from predictor variables that are continuous and/or categorical (for our case, the predictor variables are continuous). For our case, the logistic function will be a softmax function because we have many classes:

$$\sigma(x_j) = \frac{e^{x_j}}{\sum_i e^{x_i}}$$

Support Vector Machine (SVM) is a very powerful and versatile machine learning model, capable of performing linear or nonlinear classification and regression. SVMs are particularly well suited for classification of complex but small- or medium-sized datasets such as the one used for this project. SVM uses a technique called the kernel trick to transform data and then based on these transformations it finds an optimal boundary between the possible outputs.

Multilayer perceptrons (MLPs) are the classical type of neural network. They are comprised of one or more layers of neurons. Data is fed to the input layer, there may be one or more hidden layers providing levels of abstraction, and predictions are made on the output layer, also called the visible layer. MLPs are suitable for classification prediction problems where inputs are assigned a class or label. They are very flexible and can be used generally to learn a mapping from inputs to outputs.

Benchmark

No work has been done on this dataset; thus, there is no result out there that we could use as benchmark. However, this will stop us to establish one. Working on a classification problem we can use a metric score as a benchmark, train multiple classifiers and compare their score to the benchmark score. For this reason, our benchmark for this project is the ***f-beta score*** of 0.85.

Moreover, it is a good practice to train two or more classifiers and then pick the one that outperforms that others on this dataset. Thus, we are training 3 classifiers here and expect that all of them will have a f-beta score higher than 0.85; finally, the classifier that will have the highest score will be used as the final model.

III. Methodology

3.1. Data Preprocessing

We identify two major abnormalities up here:

- The testing images have different shapes
- And they are taken in different angles.

We dealt with abnormalities by creating functions that preprocess each image. Down here are the role of each function:

3.1.1. getRegions function

The role of this function is to localize the regions of the image that can contain the license plate. To extract these regions, we applied the "blackhat" operation to reveal the dark regions against the light backgrounds, then we use the Scharr gradient on the blackhat image in the x-direction. We then apply a Gaussian blur on the gradient image and create a threshold that will be used to find the contours of candidates' regions. This function return a list of regions that are supposed to have the license plate.

3.1.2. charCandidates function

This function take as input a list of license plate's regions. To correct the image angle abnormalities, we apply a "four-point transformation" ([3]) technique on these regions and thus create a new image that gives

[3] https://www.pyimagesearch.com/2014/09/01/build-kick-ass-mobile-document-scanner-just-5-

us a top-down, bird's eye view which is more suitable for character segmentation. The first abnormality is also dealt here by resizing all the new images (license plate) to the same width of 400 pixels. This will help us to be consistent when applying the connected component analysis which is an algorithmic application of graph theory that is used to determine the connectivity of "blob"-like regions in a binary image.

This function return a tuple containing the license plate, the thresholded license plate, and the license plate characters.

3.1.3. scissor function

The role of this function is to cut each character of the license plate which will help us to extract the feature of the character. The function return a list of characters.

3.1.4. Feature extraction

At this point we are in good shape. In other words, we have a list that contains each character of the license plate, all we need is to extract the feature describing each character so that we could create our feature vector to be used by the classifier to recognize the license plate number. We use the Block-Binary-Pixel-Sum descriptor to extract the feature of each character.

The Block-Binary-Pixel-Sum descriptor functions by dividing an image into non-overlapping M x N pixel blocks and the ratio of foreground pixels to the total number of pixels in each block is calculated for each region. These ratios are collected and returned as the final feature vector. The class BlockBinaryPixelSum handle this process.

The figure down here shows the feature vector of an [A] image.

```
print(f"\nPrinting the feature vector of : {labels[580]}")
data[580]
```

```
Printing the feature vector of : A
array([0.28, 0.96, 0.32, 0.44, 1.  , 0.52, 0.6 , 0.44, 0.64, 0.76, 0.4 ,
       0.8 , 0.92, 0.8 , 0.96, 1.  , 0.  , 0.8 , 0.36, 0.98, 0.42, 0.68,
       0.42, 0.72, 0.96, 0.4 , 0.88, 0.62, 0.32, 0.72, 0.52, 0.52, 0.64,
       0.58, 0.8 , 0.86, 0.96, 0.5 , 0.8 , 0.67, 0.42, 0.55, 0.72, 0.68,
       0.88])
```

3.2. Implementation

3.2.1. Data preparation

We start here by creating an array [data] that contains feature vectors of all images in the dataset and another array [labels] that contains the labels corresponding to each feature vector. The following code takes care of this part:

```
# initialize
data = []
labels = []

for samplePath in sorted(glob.glob('char_Train/*')):

    sampleName = samplePath[samplePath.rfind("/")+1:]
    imagePaths = list(paths.list_images(samplePath))

    # Loop over all images
    for imagePath in imagePaths:
        char = cv2.imread(imagePath)
        char = cv2.cvtColor(char, cv2.COLOR_BGR2GRAY)
        char = preprocessChar(char)

        # describe the image of the character using BBPS
        features = desc.describe(char)

        data.append(features)
        labels.append(sampleName)
```

We have 36 categories here (A-Z and 0-9) meaning that the data in the [labels] array must be preprocess before being fed it into any classifier. We are using two type of preprocessed labels for this project and here are the reasons why:

- One-Hot encoding is used to preprocess labels that will then be used by the multilayer parceptrons algorithm. We are using Tensorflow and the intenger encoded labeled was not working that's why we had to create One-Hot encoded labels.
- Integer encoding is used to preprocess labels that is used for the logistic regression and the support vector machine model. For the same reason as stated up here, the one-hot encoded labels did not work with these algorithms.

```
from sklearn.preprocessing import LabelEncoder
from sklearn.preprocessing import OneHotEncoder

# integer encode
label_encoder = LabelEncoder()
integer_encoded = label_encoder.fit_transform(np.array(labels))
# One Hot encoding
onehot_encoder = OneHotEncoder(sparse=False)
integer_encoded = integer_encoded.reshape(len(integer_encoded), 1)
labels_encoded = onehot_encoder.fit_transform(integer_encoded)
```

```
print("===== Example of an integer encoded label ======\n")
print(integer_encoded[0])
print("\n==== Example of a One-Hot encoded label ======\n")
print(labels_encoded[0])
```

```
===== Example of an integer encoded label ======

[0]

==== Example of a One-Hot encoded label ======

[1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
```

3.2.2. Models Training

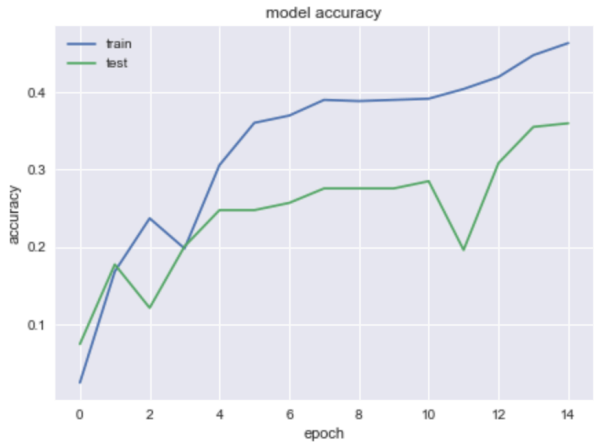
a. The Multilayer Perceptrons

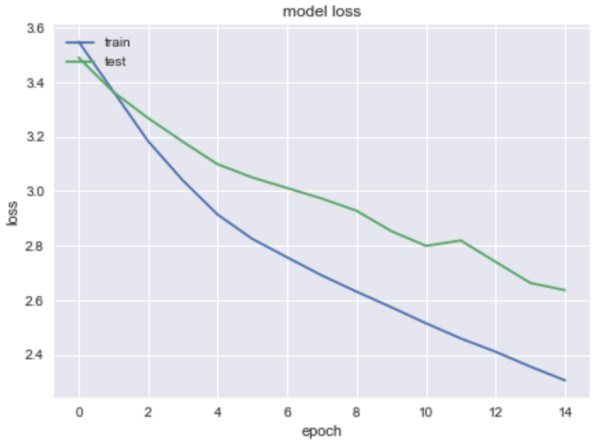
The first model trained was the multilayer perceptrons. The data and labels were split into training and testing set with a test size of 25% of the data.

Tensorflow was used to create the multilayer perceptrons and down here is the architecture of this neural network:

```
model1 = tf.keras.models.Sequential()
model1.add(tf.keras.layers.Dense(72, activation = tf.nn.relu))
model1.add(tf.keras.layers.Dense(144, activation = tf.nn.relu))
model1.add(tf.keras.layers.Dense(36, activation = tf.nn.softmax))
model1.compile(optimizer='sgd', loss = 'categorical_crossentropy',
               metrics = ['accuracy'])

history1 = model1.fit(xPercep_train, yPercep_train,
                      validation_data=(xPercep_test, yPercep_test),
                      epochs = 15, verbose=0)
```

This initial multilayer perceptrons has an input layer of 72 nodes, one hidden layer of 144 nodes and the output layer that has 36 nodes. This model is compile using a stochastic gradient descent as optimizer, and for the loss we used the “categorical crossentropy”. We used ReLu as the activation function for the input and hidden layers, and Softmax function for the output layer. The model was trained during 15 epochs. Down here we can visualize the result of this model:





This model performed poorly but it also indicates that improvements can be made and thus a robust model could be implemented.

b. The Support Vector Machine and Logistic Regression model

To train the logistic regression and Support Vector Machine model, the integer encoded data was used create the y_train and y_test variables. The test size is 25% of the data. Down here is the code and score of the initial implementation:

```
svm1 = LinearSVC()
logistic1 = LogisticRegression()

# Perform Cross Validation
scores_svm = cross_val_score(svm1, x_train, y_train, cv=3, scoring='f1_micro')
scores_logistic = cross_val_score(logistic1, x_train, y_train, cv=3, scoring='f1_micro')
print(f"SVM scores : {scores_svm} -- mean: {scores_svm.mean():.2}\n")
print(f"Logistic regression scores : {scores_logistic} -- mean: {scores_logistic.mean():.2}\n")
```

```
SVM scores : [0.95614035 0.95774648 0.97      ] -- mean: 0.96

Logistic regression scores : [0.86842105 0.91079812 0.93      ] -- mean: 0.9
```

These two models are doing well on their initial implementation with default parameters. As reported by the cross-validation score, the SVM model has a mean score of 0.96 while the logistic regression has 0.9.

3.3. Refinement

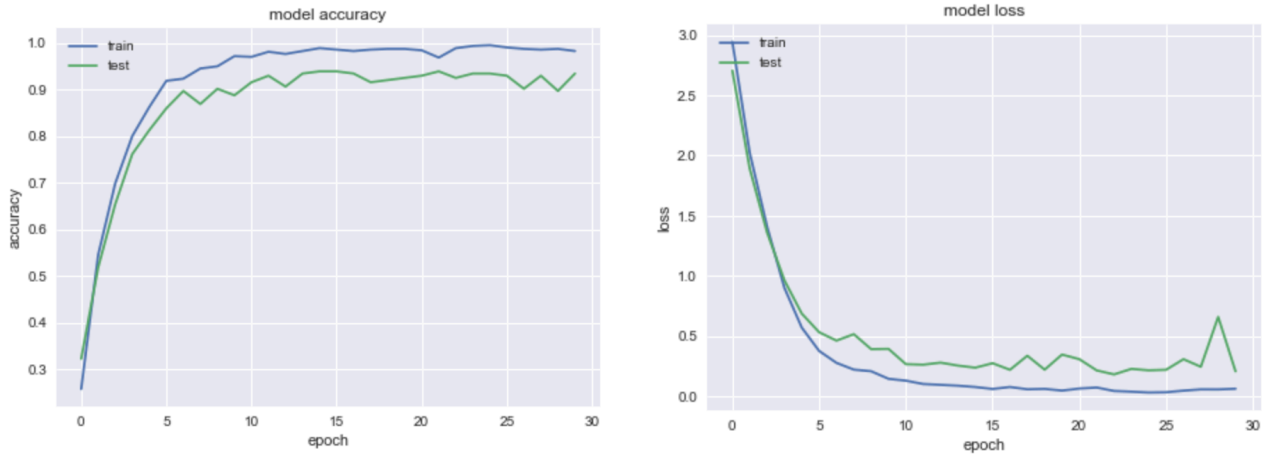
a. The Multilayer Perceptrons

The following refinements were made on this model:

- ✓ The architecture was restructured and now it has one input layer of 128 nodes, followed by two hidden layers with 200 and 240 notes respectively. We stacked Dropout layers between the input layer, the hidden layer, and the output layer.
- ✓ The optimizer was changed for the “adam”
- ✓ The model was trained during 30 epochs.

```
# Train the Multilayer Perceptron
model = tf.keras.models.Sequential()
model.add(tf.keras.layers.Dense(128, activation = tf.nn.relu))
model.add(tf.keras.layers.Dropout(0.2))
model.add(tf.keras.layers.Dense(200, activation = tf.nn.relu))
model.add(tf.keras.layers.Dropout(0.2))
model.add(tf.keras.layers.Dropout(0.2))
model.add(tf.keras.layers.Dense(240, activation = tf.nn.relu))
model.add(tf.keras.layers.Dropout(0.2))
model.add(tf.keras.layers.Dense(36, activation = tf.nn.softmax))

model.compile(optimizer='adam', loss = 'categorical_crossentropy', metrics = ['accuracy'])
history = model.fit(xPercep_train, yPercep_train, validation_data=(xPercep_test, yPercep_test),
                    epochs = 30, verbose=0)
```

The following graphics show the result of this new model



We can clearly see that this is an improved model. However, we will have to test it with car images and see how well it predict the license plate number of cars. We will test this model with the improved model of the SVM and logistic regression, then the final model will be picked.

b. The Support Vector Machine and Logistic Regression model

This are the changes made for the support vector machine model:

- ✓ We changed the solver to 'newton-cg' because it's handle multinomial loss
- ✓ Use L2 regularization as penalty is set to equal 'l2'
- ✓ We increased the number of maximum iteration from 100 to 150.

For the logistic regression, here are some changes:

- ✓ We also use L2 regularization
- ✓ The loss function was set to 'hinge'
- ✓ C = 2 and the maximum number of iteration is set to 150.

The cross validation is then perform using 5 sets. In other words, 4 sets are used for training and one set for validation. Here are the models' results:

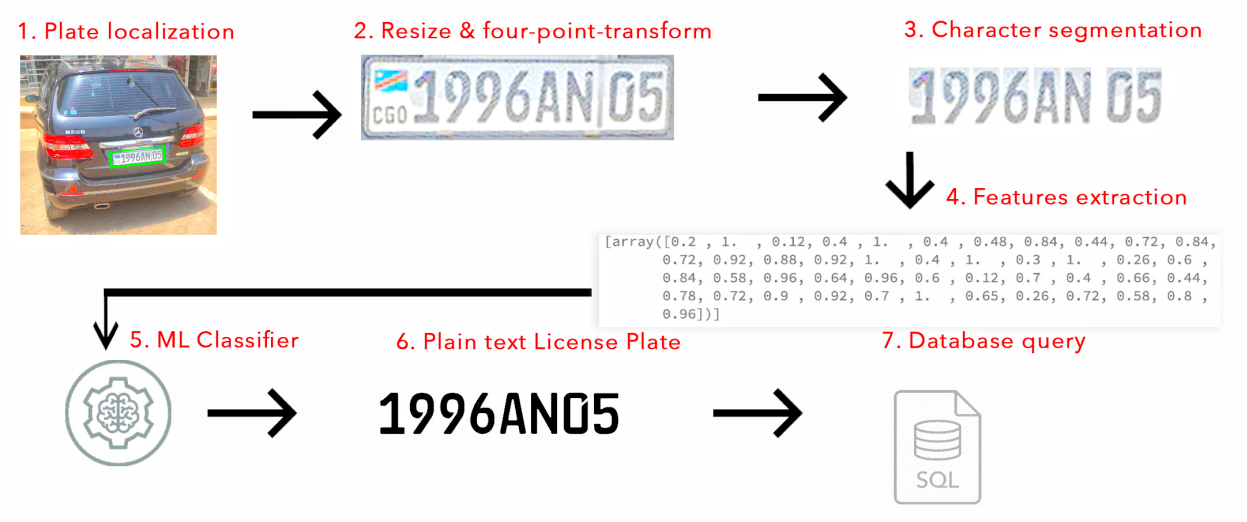
```
from sklearn.model_selection import cross_val_score
svm = LinearSVC(penalty='l2', loss='hinge', C=2, max_iter=150)
logistic = LogisticRegression(penalty='l2', C=2, solver='newton-cg', max_iter=150)

# Perform Cross Validation
scores_svm = cross_val_score(svm, x_train, y_train, cv=5, scoring='f1_micro')
scores_logistic = cross_val_score(logistic, x_train, y_train, cv=5, scoring='f1_micro')
print(f"SVM scores : {scores_svm} -- mean: {scores_svm.mean():.2}\n")
print(f"Logistic regression scores : {scores_logistic} -- mean: {scores_logistic.mean():.2}\n")
```

```
SVM scores : [0.93661972 0.94927536 0.94488189 0.95798319 0.97391304] -- mean: 0.95

Logistic regression scores : [0.92253521 0.93478261 0.90551181 0.95798319 0.96521739] -- mean: 0.94
```

We have an improvement for the logistic regression model with an average score of 0.94 while the SVM score is down from 0.96 to 0.95.

Prediction on car images and Model selection

It's time to use our 3 models in real world scenario by using car images to see how well each model perform and then pick the best model. Before that let see the pipeline of the solution:



1. The first operation is to localize the license plate on the input image. This is done by calling the *getRegions* function describe up here.

2. Once we have some candidate's regions that potentially contain the license plate, we then extract these regions from the original image and apply to it a "four-point-transformation" and then resize the extracted image to have a uniform width. It's all done by calling the *charCandidates* function.

3. The license plate image is then segmented so that each character could be describe and create a feature vector. The function *scissor* takes care of this operation

4. Things become more interesting here because we can now call the *describe* function to get the description of each character on the license plate image and create our feature vector.

5. In this step, we make prediction by classifying each character.

6. The classifiers produce a plain text license plate number as output. This is then use to query the database.

7. We created a database that contains the true license plate numbers of cars with random information on fees payment and date.

At this stage, we are going to stop at step 6 where we get the license plate number in plain text and then we will evaluate the model using f-beta score. Here is the result:

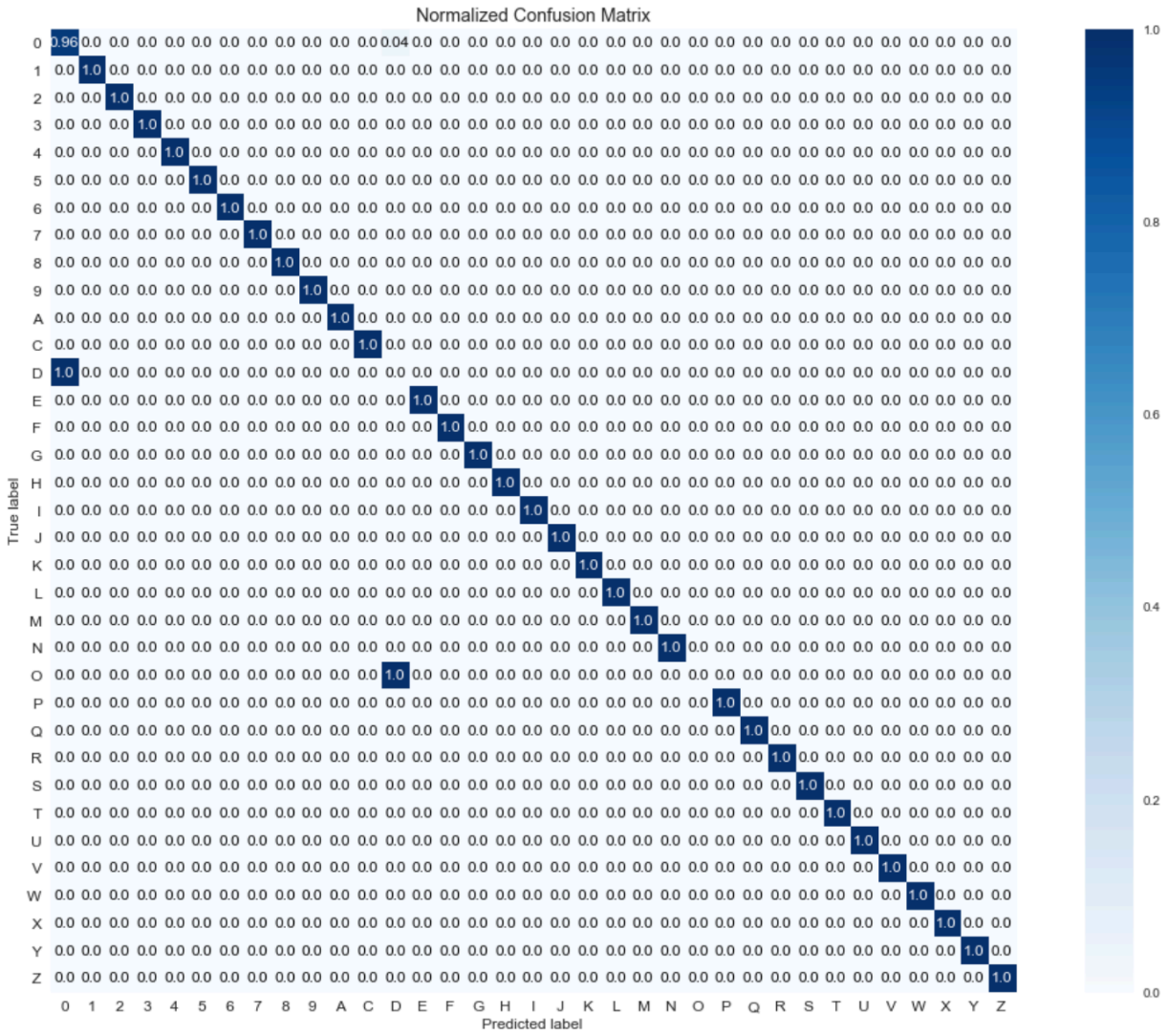
```
===== Evaluating the MLPs model ====

The f-beta score is for the MLPs model is: 0.9438202247191012
Total of True positive for the SVM model is : 84
89 license plate number were extracted


===== Evaluating the SVM model ====

The f-beta score is for the SVM model is: 0.9438202247191012
Total of True positive for the SVM model is : 84
89 license plate number were extracted


===== Evaluating the Logistic Regression model ====

The f-beta score is for the Logistic Regression model is: 0.9438202247191012
Total of True positive for the Logistic Regression model is : 84
89 license plate number were extracted
```

IV. Results

Model Evaluation and Validation

When we tested all the model on the real car images and we saw up here, they all had the same prediction with a ***f-beta score = 0.94***. Thus, we examined the confusion matrix of each model to choose the best one. The SVM model had the best result and for this reason we are going to use it as the final model for this project. Here is the SVM confusion matrix plots using the the ***Scikit-plot*** library:

```
# Ploting the SVM Confusion Matrix
y_test = [label_encoder.inverse_transform(y) for y in y_test]
y_pred_svm = svm.predict(x_test)
y_pred_svm = [label_encoder.inverse_transform(y) for y in y_pred_svm]
plt.rcParams['figure.figsize'] = (20, 14)
skplt.metrics.plot_confusion_matrix(y_test, y_pred_svm, normalize=True)
plt.show()
```



This confusion matrix shows that the model predicts well the characters that are mostly used in the Congolese license plate numbers. Although the character 'D' is misclassified and predicted as the digit '0' and the letter 'O' is predicted as 'D', the rest of characters are well classified. Therefore, as we noticed up here that the characters that are mostly used are mainly all digits, and alphabets like A, M, and N, picking this model to solve the actual problem make sense.

Justification

The main benchmark for this project was the metric that we choose to evaluate the models tested to solve the problem. Because there is no algorithm that was previously implemented using the actual data, we had to rely on the ***f-beta score*** of different classifiers that was tested on this project. We set ***f-beta score = 0.85*** as the benchmark for this project.

After doing cross-validation on test data and testing all 3 models on the real-world car images, we picked the Support Vector Model because it's cross-validation ***f-beta score*** is higher than the benchmark and its confusion matrix looks good.

The final setting of this model is with its default parameter, because the refined model performed a little bit poorly. We can see that in this two figures:

Initial training

```
from sklearn.model_selection import cross_val_score
```

```
svm1 = LinearSVC()
logistic1 = LogisticRegression()

# Perform Cross Validation
scores_svm = cross_val_score(svm1, x_train, y_train, cv=3, scoring='f1_micro')
scores_logistic = cross_val_score(logistic1, x_train, y_train, cv=3, scoring='f1_micro')
print(f"SVM scores : {scores_svm} -- mean: {scores_svm.mean():.2}\n")
print(f"Logistic regression scores : {scores_logistic} -- mean: {scores_logistic.mean():.2}\n")
```

```
SVM scores : [0.95614035 0.95774648 0.97      ] -- mean: 0.96

Logistic regression scores : [0.86842105 0.91079812 0.93      ] -- mean: 0.9
```

Refined models

```
from sklearn.model_selection import cross_val_score
svm = LinearSVC(penalty='l2', loss='hinge', C=2, max_iter=150)
logistic = LogisticRegression(penalty='l2', C=2, solver='newton-cg', max_iter=150)

# Perform Cross Validation
scores_svm = cross_val_score(svm, x_train, y_train, cv=5, scoring='f1_micro')
scores_logistic = cross_val_score(logistic, x_train, y_train, cv=5, scoring='f1_micro')
print(f"SVM scores : {scores_svm} -- mean: {scores_svm.mean():.2}\n")
print(f"Logistic regression scores : {scores_logistic} -- mean: {scores_logistic.mean():.2}\n")
```

```
SVM scores : [0.93661972 0.94927536 0.94488189 0.95798319 0.97391304] -- mean: 0.95

Logistic regression scores : [0.92253521 0.93478261 0.90551181 0.95798319 0.96521739] -- mean: 0.94
```

V. Conclusion

Free-Form Visualization

In this section, we are going to provide some elements that affected the prediction of the model. It's clear that the preprocessing part of the project plays an important role on the license plate number recognition because if one character is not segmented this lead the non-description of the character and thus we have some prediction with less than 8 characters.

As we can see in the left figure, the finale license plate (Char Threshold figure) that will be segmented and thus described has one last character left out. The zero has been left out because we set the number of characters to be 8; now the line between J and 1 is not a character. However, for this example due to the angle and light of the original picture, the line has been a character candidate and therefore, the last character has been cut out. This result to the model predicting this license plate number as – 0524AJ73 instead of 0524AJ10.

The figure on the right shows another problem where the pixels of '7' are not connected and thus the top part is cut off. The descriptor will describe the rest of the digit that looks like a 1.

To visualize these figures while predicting the license plate number, we must uncomment the "following lines on the charCandidates function:

```
printImage(imutils.resize(plate, width=400), title="Plate after
perspective transform")
printplt(thresh, title='LP Threshold')
printplt(charCandidates, title = "Pruned Candidates")
printplt(thresh, title = "Char Threshold")
```

Reflection

This project was about creating a License Plate Recognition and Search System for the Democratic Republic of Congo. We used a Support Vector Machine as our final model to classify the character found on a license plate. However, before any classification of characters, we had to preprocess the image that contains the car by localizing first the regions that could contain the license plate, afterward we extracted these regions and segmented the ones that had potential characters. To describe these characters and create a feature

vector, we use the block-binary-pixel-sum method which consist of dividing each segmented character into different sizes of blocks, then choose the sum of pixels each block as a member of features ([4]).

After classifying each character, we had the license plate number as a plain text variable, we use it to query a database that we created for simulation. The database return information's about the car whether the tag fee had been paid or not and on what date.

One interesting thing about this project is that I learned a lot about computer vision and image processing techniques than I imaged. Also, I realized how messy a dataset can be since all the images used here was received from friends and families; I received more than 200 images but only used 100 because the others were useless no matter what and it took me time to eliminate these images from the final dataset.

One other thing to note is that although deep learning algorithms are robust for this kind of problem, a simple algorithm like support vector machine used here can handle the problem in a situation, like ours, were we have few data.

Improvement

One improvement could be to train two classifiers: one for digits and the second for alphabets because as we can see, the Congolese license plate number follow a certain pattern of four digits, two alphabets, and end with two digits.

Another improvement would be the design of the license plate number. I realize that the bolts on the license plate were fixed differently from car to car. Other bolts were placed in a way that linked two characters and thus making it difficult to preprocess. For this reason, if funded, we will try to redesign and predispose places were the bolts should be fixed.

[4] YenChing Chang et al (2013), License Plate Character Recognition Using Block-Binary-Pixel-Sum

References

1. YenChing Chang et al (2013), License Plate Character Recognition Using Block-Binary-Pixel-Sum Features, https://www.atlantis-press.com/proceedings/iccnce-13/6483
2. Adrian Rosebrock, Pratical Python and OpenCV + Case Studies, 3rd Edition 2016. https://www.pyimagesearch.com/practical-python-opencv/
3. Baoguang Shi, (2017), Detecting and Recognizing Text in Natural Images. From https://bit.ly/2xWSaOA
4. Vishnu Sundaresan, Jasper Lin, Recognizing Handwritten Digits and Characters, http://cs231n.stanford.edu/reports/2015/pdfs/vishnu_final.pdf
5. Yutaka Sasaki, https://www.cs.odu.edu/~mukka/cs795sum10dm/Lecturenotes/Day3/F-measure-YS-26Oct07.pdf
6. Udacity, Machine Learning Engineer Nanodegree, www.udacity.com