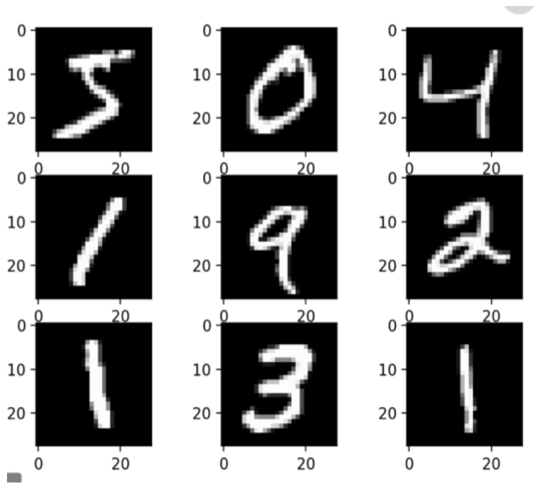
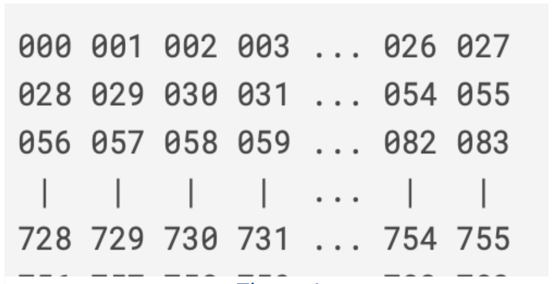
Handwritten Digit Recognition

Handwritten digits recognition is an area of machine learning, in which a machine is trained to identify handwritten digits. Humans recognize handwritten digits with ease, however, in order for machines to perform the recognition, it has to be trained so that the machine can recognize the digits based on the training given with the past data sets with the known digit labels. Machine learning models such as artificial neural network, convolutional neural network, decision tree classifier can be applied to this handwritten digits recognition problem. Handwritten digits recognition helps us to solve more complex problems and makes out tasks easier. For instance, it helps us in the processing of bank checks and zip code recognition. It is a research issue and there are different approaches and algorithms proposed by researchers, however, there is no perfect machine learning model in recognizing handwritten digits.

Datasets like Kaggle and MINST help simplify the handwritten digits recognition problem. These datasets contain a large number of sample images of handwritten digits that are already normalized to 28x28 pixels as you can see to the right. First, we will start using Kaggle dataset which contains 21,000 training and 21,000 testings handwritten digits. Each pixel has a grayscale value from 0 to 255. Since it's 28 by 28 pixels, this means each image consists of 784 pixels and we will be treating those pixels as our features. Therefore our Kaggle dataset would contain 42000 rows and 720 columns.





```
000 001 002 003 ... 026 027
028 029 030 031 ... 054 055
056 057 058 059 ... 082 083
 |   |   |   |  ...  |   |
728 729 730 731 ... 754 755
```

Figure 1

Decision Tree Classifier is a simple and widely used classification model. Decision Tree Classifier asks questions about the data's features. Every time it receives an answer, it asks a follow-up question until a conclusion about the class label of the data is reached. These questions and conditions are organized in a tree structure. In the decision tree, the root and internal nodes contain attribute test conditions to separate the digits that have different features.

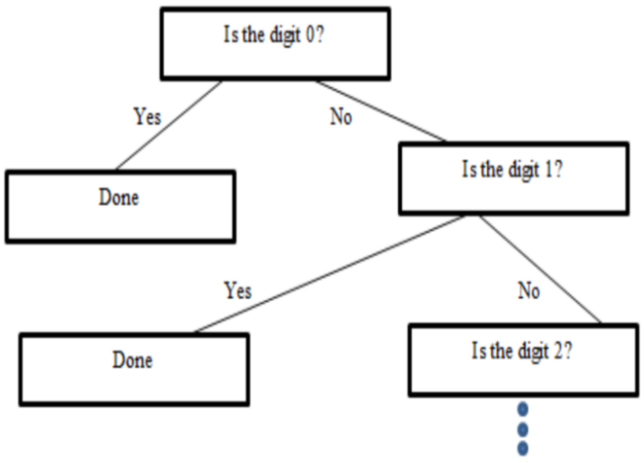
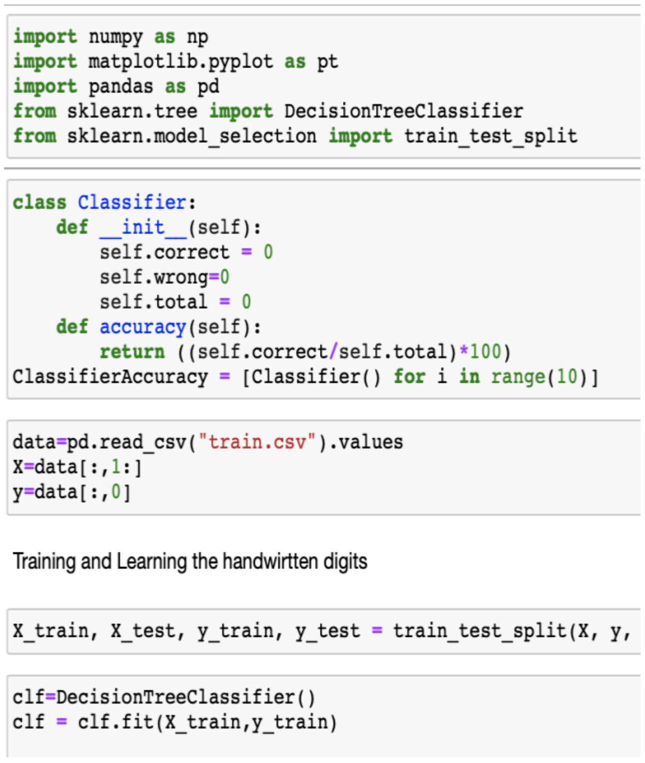


Figure 2

The goal here is to create a model that predicts and recognize the digit by learning simple decision rules inferred from the image pixels. We will build and optimize the Decision Tree Classifier using the Python Scikit-learn package. Pandas, Numpy, and Matplotlib libraries will be used as well for fast analysis, computing and visualization. We will be using this approach to recognize handwritten digits and evaluate the accuracy of the model against each digit from 0 to 9. First, we Load the Kaggle handwritten digits dataset for classification. Then we split the data sets into two sets, one for the training and the rest for testing. After that, we start learning and predicting and the recognizer is trained to predict the given an image of the handwritten digit. We will use the



```
import numpy as np
import matplotlib.pyplot as pt
import pandas as pd
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import train_test_split
```

```
class Classifier:
    def __init__(self):
        self.correct = 0
        self.wrong=0
        self.total = 0
    def accuracy(self):
        return ((self.correct/self.total)*100)
ClassifierAccuracy = [Classifier() for i in range(10)]
```

```
data=pd.read_csv("train.csv").values
X=data[:,1:]
y=data[:,0]
```

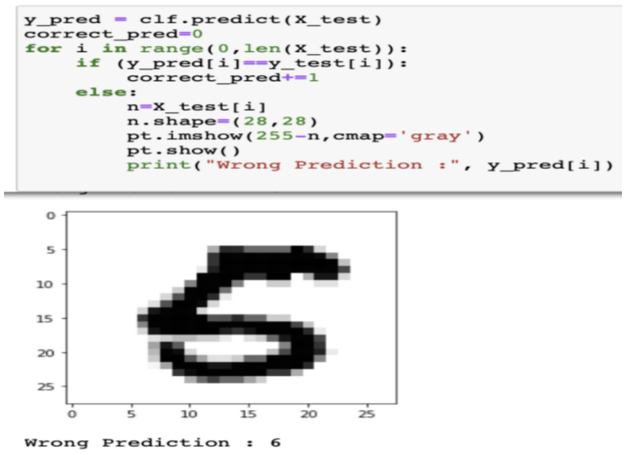
Training and Learning the handwirtten digits

```
X_train, X_test, y_train, y_test = train_test_split(X, y,
```

```
clf=DecisionTreeClassifier()
clf = clf.fit(X_train,y_train)
```

Figure 3

samples of each of the 10 possible classes of handwritten digits from Kaggle dataset (0-9) on which we fit an estimator to predict the classes to which unseen samples belong to.



```
y_pred = clf.predict(X_test)
correct_pred=0
for i in range(0,len(X_test)):
    if (y_pred[i]==y_test[i]):
        correct_pred+=1
    else:
        n=X_test[i]
        n.shape=(28,28)
        pt.imshow(255-n,cmap='gray')
        pt.show()
        print("Wrong Prediction :", y_pred[i])
```

```
Wrong Prediction : 6
```

Figure 4

In Figure 4, some of the digits that are not recognized by the decision tree classifier are displayed. For example, 5 were predicted as 6 and the reason behind this is the similarity of handwriting style between 5 and 6. The accuracy of the classifier in recognizing each digit was calculated and is shown in below Table 1.

Taking into consideration the size of the dataset (42000 rows of data), the system's overall performance is satisfactory with an accuracy of 84%.

Figure 5

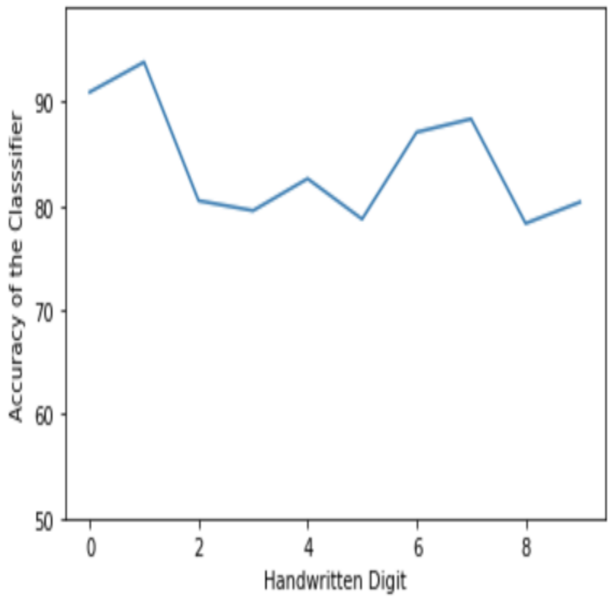
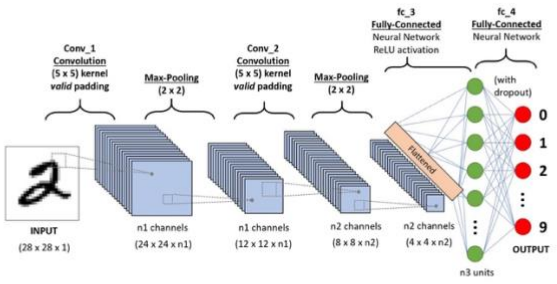


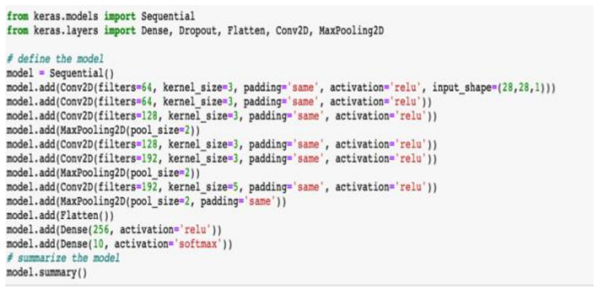
Table 1

HandWritten Digit	Total Tests	Accuracy of the Classifier
0	2101	90.91
1	2353	93.8
2	2081	80.49
3	2175	79.54
4	2006	82.6
5	1866	78.72
6	2095	87.06
7	2204	88.34
8	2047	78.31
9	2072	80.36
All Digits	21000	84.22

As you can see 3,8,5 had the lowest accuracy and the reason behind this is the similarity of handwriting style between the three numbers.

After using Decision Tree classifier, I decided to use Convolutional Neural Network because they are mainly used for image classification, so I expect better results. A neural network is a machine learning model that consists of connected layers of neurons. A neuron contains a number, that we call an activation. Connections have weights, which tells us the strength of the signal to the connected neuron. A Convolutional Neural Network is a Deep Learning algorithm which can take in an input image, assign learnable weights and biases to various parts in the image and be able to differentiate one from the other. While in the decision tree filters are hand-engineered, with enough training, CNN has the ability to learn these filters/characteristics because the pre-processing required is much lower.





```
from keras.models import Sequential
from keras.layers import Dense, Dropout, Flatten, Conv2D, MaxPooling2D

# define the model
model = Sequential()
model.add(Conv2D(filters=64, kernel_size=3, padding='same', activation='relu', input_shape=(28,28,1)))
model.add(Conv2D(filters=64, kernel_size=3, padding='same', activation='relu'))
model.add(Conv2D(filters=128, kernel_size=3, padding='same', activation='relu'))
model.add(MaxPooling2D(pool_size=2))
model.add(Conv2D(filters=128, kernel_size=3, padding='same', activation='relu'))
model.add(Conv2D(filters=192, kernel_size=3, padding='same', activation='relu'))
model.add(MaxPooling2D(pool_size=2))
model.add(Conv2D(filters=192, kernel_size=5, padding='same', activation='relu'))
model.add(MaxPooling2D(pool_size=2, padding='same'))
model.add(Flatten())
model.add(Dense(256, activation='relu'))
model.add(Dense(10, activation='softmax'))
# summarize the model
model.summary()
```

Figure 6

After Reshaping and Normalizing the Images. I used API Keras to import the Sequential Model. The functional API in Keras is an alternative way of creating models that offers a lot more flexibility and it makes creating deep learning models fast and easy. Keras works with deep learning frameworks like Tensorflow so we will setup our environment with Keras using Tensorflow as the backend. I added Convolutional, Max Pooling, Flatten, and Dense layers. As you can see in the model in Figure 6, input data is given into the first layer as raw pixel values , activating each input neuron to some extent. Based on the weights and an activation function the network determines which neurons from the next layer to activate and how strong the activation is going to be. This process is called feedforward and we repeat it until one of the output neurons is activated.

There are many different layers in a convolutional neural network. However, convolution, pooling, and fully connected layers are the main ones. Convolutional layer is the very first layer where we will be extracting the pixels from the digit images in our datasets. Convolution is the idea of filtering the image with a smaller pixel filter to decrease the size of the image without losing the relationship between those pixels. We are able to do this because pixels are only related with the adjacent and close pixels. For instance, if we apply convolution to a 5x5 image by using a 3x3 filter with one-pixel shift at each step, we will end up having a 3x3 simpler output. Then we have max pooling layer that selects the maximum element from the region of the feature map covered by the filter. Flatten layers flatten the two-dimensional arrays before building the fully connected layers. This allows us to have a feature map containing the most prominent features of the previous feature map. Both the Convolutional and Pooling layers reduce the time-space complexity significantly. This allows us to n construct a fully connected network in the end to classify our images.

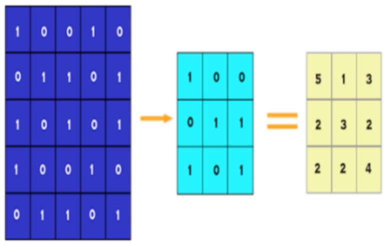


Figure 8 Convolutional Layers

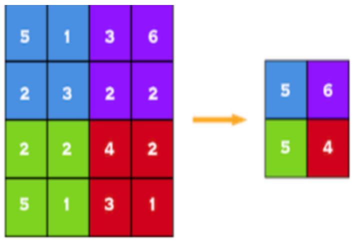


Figure 9 Max Pooling

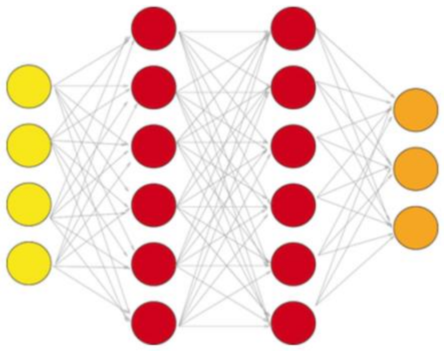
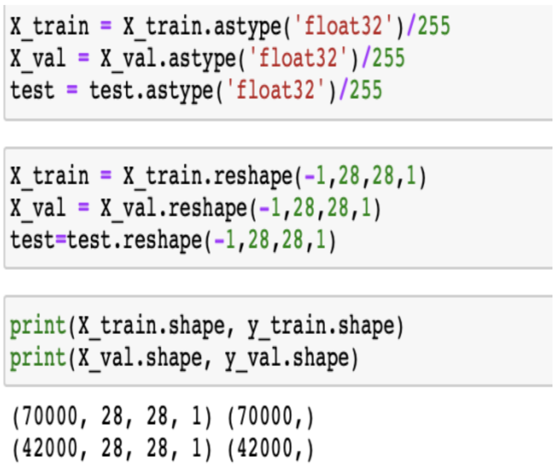


Figure 7 Fully Connected Layer

After building the Convolutional Neural Network and evaluating it. The results were satisfactory for such a simple model and I got an accuracy of 95%. However, I decided to use the

MNIST and see how much I can improve accuracy. MNIST is one of the most common datasets used for image classification and accessible from many different sources. It's so common that Tensorflow and Keras allow us to import and download it directly from their API. It contains 60,000 training images and 10,000 testing images but I used all 70000 images for training and the previous Kaggle dataset for validation. The MNIST was constructed from a number of scanned document dataset available from the National Institute of Standards and Technology (NIST) and that's where the name came from. To be able to use the dataset in Keras API, we need 4-dims NumPy arrays. This means we must normalize our data as it is always required in neural network models. We do this by dividing the RGB codes to 255.



```
X_train = X_train.astype('float32')/255
X_val = X_val.astype('float32')/255
test = test.astype('float32')/255
```

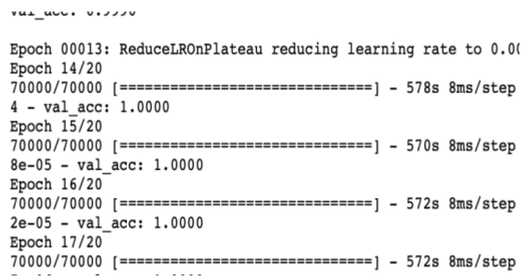
```
X_train = X_train.reshape(-1,28,28,1)
X_val = X_val.reshape(-1,28,28,1)
test=test.reshape(-1,28,28,1)
```

```
print(X_train.shape, y_train.shape)
print(X_val.shape, y_val.shape)
```

```
(70000, 28, 28, 1) (70000,)
(42000, 28, 28, 1) (42000,)
```

Figure 10

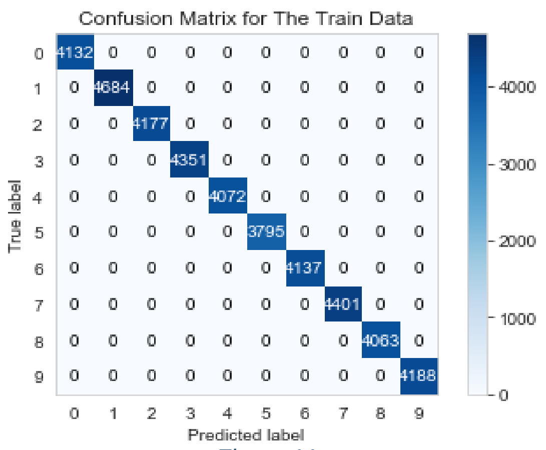
Using the MNIST dataset for training and Kaggle for validation, I was able to achieve "100% accuracy." It took a while but after just 14 epochs, we got 100% accuracy. The test performance is also shown in the confusion matrix in Figure 12. In the confusion matrix I put the predicted labels as the rows and the true labels as the columns where every predicted label will be compared with other digits, and we can easily see where the model predicted wrong and where it predicted correctly. By looking at the confusion matrix you could easily figure out what percent of predictions made by our classifier were correct and where it



```
val_acc: 0.9990

Epoch 00013: ReduceLROnPlateau reducing learning rate to 0.0(
Epoch 14/20
70000/70000 [==============================] - 578s 8ms/step
4 - val_acc: 1.0000
Epoch 15/20
70000/70000 [==============================] - 570s 8ms/step
8e-05 - val_acc: 1.0000
Epoch 16/20
70000/70000 [==============================] - 572s 8ms/step
2e-05 - val_acc: 1.0000
Epoch 17/20
70000/70000 [==============================] - 572s 8ms/step
```

Figure 12

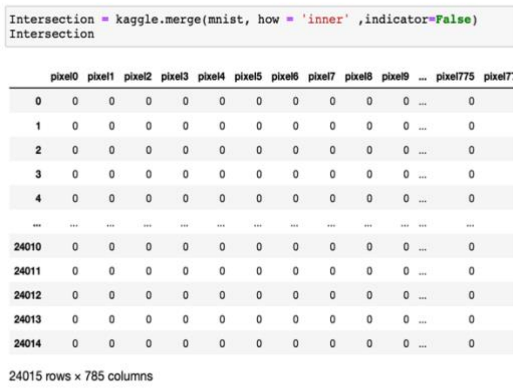


	0	1	2	3	4	5	6	7	8	9
0	4132	0	0	0	0	0	0	0	0	0
1	0	4684	0	0	0	0	0	0	0	0
2	0	0	4177	0	0	0	0	0	0	0
3	0	0	0	4351	0	0	0	0	0	0
4	0	0	0	0	4072	0	0	0	0	0
5	0	0	0	0	0	3795	0	0	0	0
6	0	0	0	0	0	0	4137	0	0	0
7	0	0	0	0	0	0	0	4401	0	0
8	0	0	0	0	0	0	0	0	4063	0
9	0	0	0	0	0	0	0	0	0	4188

Figure 11

was difficult for the classifier to predict the actual classification. However, since our model was "perfect" and got all 42000 tests correct, this confusion matrix doesn't say much.

After presenting these results to Erik K. Grimmelmann, Ph.D. He asked me something that made me question the accuracy of my model. He asked me whether Kaggle and MNIST were disjoint or not. This is something that should have crossed my mind before, but it didn't. I tried to do some research to find that out, but I didn't find anything saying the two datsets have common data. Thus, I decided to use python and the two datasets and find that out myself. Looking at Figure 13, one could see that more than half of Kaggle images are in MNIST. Therefore, the two datasets weren't disjoint, and this might have been the reason I got 100% accuracy.



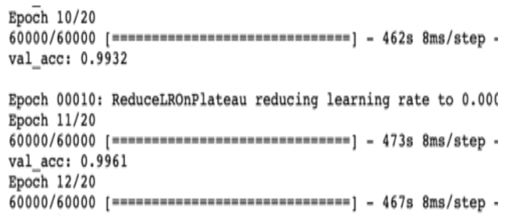
```
Intersection = kaggle.merge(mnist, how = 'inner' ,indicator=False)
Intersection
```

	pixel0	pixel1	pixel2	pixel3	pixel4	pixel5	pixel6	pixel7	pixel8	pixel9	...	pixel775	pixel7
0	0	0	0	0	0	0	0	0	0	0	...	0	
1	0	0	0	0	0	0	0	0	0	0	...	0	
2	0	0	0	0	0	0	0	0	0	0	...	0	
3	0	0	0	0	0	0	0	0	0	0	...	0	
4	0	0	0	0	0	0	0	0	0	0	...	0	
...	...	...	...	...	...	...	...	...	...	...	...	...	
24010	0	0	0	0	0	0	0	0	0	0	...	0	
24011	0	0	0	0	0	0	0	0	0	0	...	0	
24012	0	0	0	0	0	0	0	0	0	0	...	0	
24013	0	0	0	0	0	0	0	0	0	0	...	0	
24014	0	0	0	0	0	0	0	0	0	0	...	0	

24015 rows × 785 columns

Figure 13

To make the test more reliable I just used the MNIST dataset alone. The MNIST database contains 60,000 training images and 10,000 testings. After 10 epochs I was able to achieve a 99.6% accuracy. This means that even though the Kaggle and MNIST datasets are not disjoint, that didn't take much from the accuracy of our model. Out of the 1000 tastings, 32 of our predictions were wrong. From these 32 wrong predictions, 9 of them were because of the confusion between the digit 9 and 4. This is reasonable because of the similarity of handwriting style between the two numbers.



```
Epoch 10/20
60000/60000 [==============================] - 462s 8ms/step -
val_acc: 0.9932

Epoch 00010: ReduceLROnPlateau reducing learning rate to 0.00(
Epoch 11/20
60000/60000 [==============================] - 473s 8ms/step -
val_acc: 0.9961
Epoch 12/20
60000/60000 [==============================] - 467s 8ms/step -
```

Figure 14

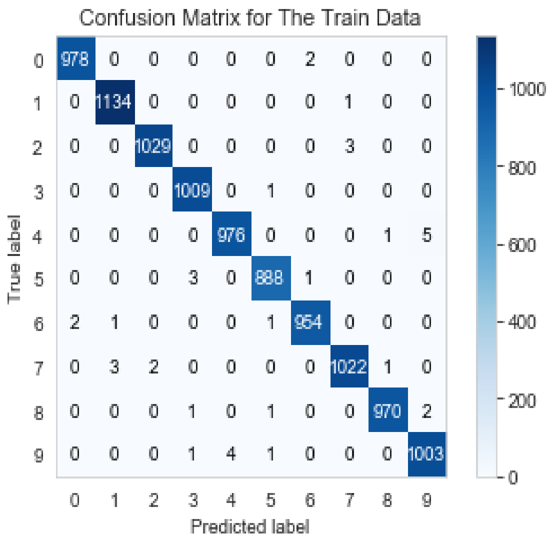


Figure 15

As using machine learning algorithms are used like Tree decision and Convolutional Neural Network, we see that Tree decision was less accurate but faster and simpler. Accuracy can depend on the splitting of training and testing data, and this can further be improved if the number of training and testing data is bigger. Both classifiers have their own accuracy and time consumption, but we realized that if the power of CPU changes to GPU, the classifier can perform better and give us more accurate results. Overall CNN using Deep learning performed better in recognizing these handwritten Digits.

Handwritten digits recognition helps us to solve complex problems and makes our tasks much simpler. It helps us in the processing of bank checks and zip code recognition but note that those problems wouldn't have segmented digits like we had in our datasets. In reality we will have multiple digits next to each other just like in Figure 16. We would need to change the image to greyscale and find the contours in the image to get the segmented digits. After having those segmented digits, we would need to reshape them to match them to be like or training data and make sure they are 28x28 pixels. This should make us appreciate the Human Visual System. Humans recognize handwritten digits with ease. In each of the hemisphere in the human brain, human use visual cortex, also known as V1, containing 140 million neurons which have tens of billions of connections between them. Humans don't just have V1 but many visual cortices doing more complex image processing.

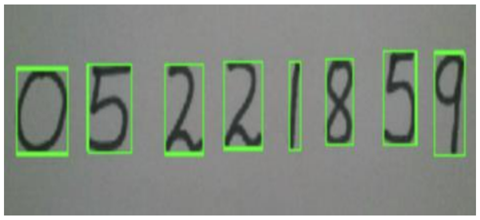


Figure 16

Nevertheless, both the decision tree classifier and Convolutional Neural Network are efficient in the recognition of handwritten digits. In the decision tree the machine was trained with a kaggle dataset which contains 42000 images and we were able to get 84% accuracy. We also were able to build a convolutional neural network to classify handwritten digits with TensorFlow's Keras API and achieve 99.6% accuracy. Overall, I was able to use what I have learned during the course and apply it to build these two simple models. I would like to take this and build more complex models that solves more image processing problems.

REFERENCES

Assegie, Tsehay & Nair, Pramod. (2019). Handwritten digits recognition with decision tree classification: a machine learning approach. International Journal of Electrical and Computer Engineering (IJECE). 9. 4446. 10.11591/ijece.v9i5.pp4446-4451.

Beniwal, Himanshu. “Handwritten Digit Recognition Using Machine Learning.” Medium, Medium, 27 Dec. 2018, medium.com/@himanshubeniwal/handwritten-digit-recognition-using-machine-learning-ad30562a9b64.

Nielsen, and Michael A. “Neural Networks and Deep Learning.” Neural Networks and Deep Learning, Determination Press, 1 Jan. 1970, neuralnetworksanddeeplearning.com/chap1.html.

Shengfeng Chen, Rabia Almamlook, Yuwen Gu, Dr. Lee wells, “*Offline Handwritten Digits Recognition Using Machine learning*,” *Proceedings of the International Conference on Industrial Engineering and Operations Management Washington DC*.

Yal, Orhan Gazi. “Image Classification in 10 Minutes with MNIST Dataset.” Medium, Towards Data Science, 22 Mar. 2020, towardsdatascience.com/image-classification-in-10-minutes-with-mnist-dataset-54c35b77a38d.