UNIVERSITY OF
LINCOLN

Machine Learning

CMP9137M | Assessment Item 1

Peter Hart | 12421031

The following assignment encompasses the acquisition of the national institute of standard and technology (NIST) dataset, which is comprised of copious amounts of handwritten digit image examples with the digits values ranging between zero and nine. The supplied images are normalised bilevel images which contain grey levels because of an anti-aliasing technique used during normalisation, each image was centred within a 28 x 28 image field. As such, the fundamental task presented within this assignment was to solve the problem of recognising handwritten digits with various classification models using the datasets that were provided, such that the classification models would be capable of correctly classifying the value of a new handwritten digit image sample. As such, the following report will detail a total of three different classification models that were implemented and will provide discussion and critique between each of the models. Furthermore, the digits dataset was provided in a total of three subsets; namely 5000 training samples, 1000 validation samples and 500 test samples. The data was acquired from the three different datasets by importing all the image pixel intensity values into a matrix depicting the dataset features, whereas the digit labels were acquired by observing the digit number character on the filename for each input image. The features will be used for training and tuning the various classification models, whereas the labels will be used for comparing the predicted output with the expected output to establish an accuracy metric for each of the models.

Decision Trees

The first classification model implemented was a decision trees classifier, a non-parametric supervised learning classifier. The decision tree classifier largely operates in the form of a hierarchical top-down tree structure, whereby the dataset is gradually segregated into increasingly smaller subsets while an associated decision tree structure is continuously developed over time. As such, the final tree output depicts three fundamental components; namely the decision nodes, leaf nodes and the root node, where the decision nodes feature two or more branches in the tree whereas the leaf nodes are representative of a possible classification output.

As explained by Kulkarni and Shrestha (2017), every attribute pertaining to the tree is tested at the root node such that the attribute which justifiably classifies the data the best is selected, as such two fundamental components are computed to assess the overall qualities of the decision tree structure; namely entropy and information gain. The calculated entropy value can be considered as a measure of the amount of randomness that exists within the feature distribution of the dataset, which therefore can be considered as a measure of the impurity. For example, if the resulting entropy value is higher, it can be inferred that a node may contain more information. To this end, the information gain is estimated by evaluating the difference in entropy (Kulkarni and Shrestha (2017), where the information gain value was effectively used for ranking the importance of a given attribute from the vectors of features, which in turn is utilised when composing the order of attributes within each node of the developed hierarchical decision tree structure. Thus, the attribute which can be described as having the highest information gain value throughout the entire training dataset would be allocated as the root node for the decision tree structure, where every attribute is tested and the attribute which best classifies the data is selected. As such, this process is then repeated, where the next decision node is selected based on the attribute with the next highest information gain and continues to recurse until all the leaf nodes have been established and all attributes have been classified.

As advocated by Lu and Yang (2009), decision tree classification models advantageously present the opportunity for the development of an easy to use and efficient classification model, where this classifier can be argued to scale well with a larger dataset and still require a low training time with respect to alternative models. This is largely due to the tree size of the decision tree classifier being independent from the total size of the dataset, therefore the training time is proportional to the fixed height of the tree. Thus, the speed and effective performance presented with a decision tree classifier could be argued to challenge the efficiency of alternative models.

Support Vector Machine (SVM)

An additional classification model which was explored was an SVM classifier, a supervised linear classification learning model. The SVM model largely operates by attempting to localise the hyperplanes which can segregate the ten different digit classifications most effectively. However, before the SVM model can be associated with the digits datasets, each pixel value within each image sample must be transformed such that each pixel is represented within its own dimension. Furthermore, this would flatten the original two spatial dimensions of the original greyscale image matrices, for example each pixel contained within each of the 28 x 28 digit image samples

would be contained within its own dimension and therefore generating a total of 784 dimensions for 784 pixel intensities. As such, each pixel intensity contained within the 784 dimensions of each image can be considered as a train point for the SVM classifier, thus each image is subsequently being indexed within a new array where each row is indicative of the pixel intensities defining a different digit image sample. For example, the training dataset size evaluates to 5000 x 784, the validation dataset size evaluates to 1000 x 784 and the test dataset size evaluates to 500 x 784.

One of the key aspects of the SVM classifier is the estimation of the hyperplane, a parameter which denotes the decision boundary when the model attempts to segregate and distinguish between the ten possible digit classifications, furthermore deciding that the train points above the hyperplane should be associated with the digit classification, whereas the train points below the hyperplane should be classified otherwise. To this end, this initial SVM classification process is relatable to that of a binary classifier, merely classifying each train point between two different classes. The fundamental objective being accomplished with the SVM is to ensure that the margin between the hyperplane and the two classifications is maximised, this in turn will ensure that the estimated classification error is minimised and ensure that the classifier is capable of reliably correctly classifying each handwritten digit sample. However, as previously elaborated, the problem presented within this assignment is the successful recognition of multiple handwritten digits, thereby the recognition of multiple handwritten digit classifications. As such, a mere binary classification with a singular hyperplane is insufficient for multiclass classification, therefore a one vs one scheme (OVO) was applied.

$$OVO = \frac{N(N-1)}{2}$$

Formula 1. Formula for estimating the number of SVM classifiers required for one-vs-one (OVO).

The OVO scheme continues to train an SVM with a single hyperplane to produce binary classifications, however an imperative difference introduced with an OVO scheme is that multiple SVM classifiers are trained for each possible pair of digit classifications, for example a total of 45 SVM classifiers would be required to classify the 10 possible digit classifications involved within this assignment. As such, training each of the SVM classifiers entails only using the subset of instances from the training data which contains either of the classes pertaining to the current pair of class labels, whereas the other instances are ignored by this classifier. To surmise, the final output prediction is depicted through a voting strategy between each of the 45 different classifiers, where the class which received the most votes is selected as the final prediction.

As noted by Fedorovici and Dragan (2011), while alternative approaches such as a neural network architecture offer the opportunity to perform better overall, an SVM classifier presented the advantage of being able to train the model in a shorter amount of time in comparison while still achieving a high performance. For example, in a comparative study conducted by Fedorovici and Dragan (2011), a neural network architecture required a total of two weeks to fully train and fine-tune the model with their training dataset, whereas the SVM classifier was capable of training on the same amount of data in under thirty minutes. Though the training of the CNN architecture involved in this assignment fortunately did not require two weeks of training, it can be observed that the CNN architecture required a considerably larger amount of time to fully train and refine the model with the training dataset involved within this assignment.

Convolutional Neural Network (CNN)

However, one final approach which was implemented was a convolutional neural network (CNN) with a softmax function. A CNN is a feedforward network which delegates a series of logistic regression models stacked on top of each other, with the final layer being either a logistic or linear regression model (Murphy, 2012). The CNN employs a deep-learning strategy which is inherently inspired by the structure of visual cortex within a biological brain, whereby the network will tune a set of weights based on the propagation of neurons that are derived from the input images through a set of hidden layers within the neural network. Unlike the previous approaches, the aim of a CNN architecture is to perform image classification by observing low level spatial features such as foreground edges and use this knowledge to develop a more abstract understanding through multiple convolution layers.

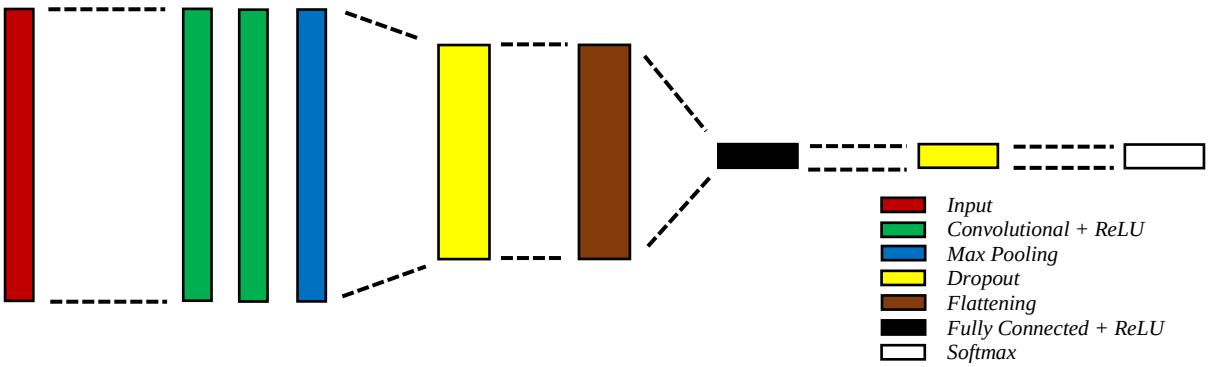

Figure 1. CNN architecture implemented for the handwritten digit recognition problem.

The CNN model propagates the model weights through a technique referred to as backward propagation, whereby the neural network effectively initialises with randomly defined parameters and continuously tunes the model weights over 12 epochs to ensure that the correct classifications can be achieved. For example, if a handwritten sample of the digit '7' was mistakenly classified as the digit '2', the CNN model would slowly adapt the model parameters such that the probability of this digit being correctly classified as the digit '7' is increased during the next epoch. As such, this training process continues until the full extent of all training batch of images have been processed during an epoch, at which point the training phase continues to the next epoch while maintaining the tuned internal parameters of the model. As shown in figure 1, each batch of images will be propagated through a total of 8 layers within the CNN architecture defined within this report. Unlike previously discussed approaches, a CNN deep learning strategy will operate on the input image data and attempt to learn features given the two-dimensional spatial proximities of the pixel intensities contained within each of the image matrices. As such, the input digit training images are input into the CNN in a series of batches, where a total of 128 different input images will be collated together in a single batch, where multiple batches will be congregated until the full stream of the input dataset has been processed. 12 epochs were selected as an appropriate amount of time steps for training the CNN as the validation loss and validation accuracy score could be observed as stabilising with this amount of time, thus additional epochs could be argued as being redundant and possibly lead to overfitting the model with the training dataset, thus the accuracy of the overfitted model would be compromised.

Furthermore, each batch of images are propagated through a convolutional layer, whereby a 3 x 3 filter, otherwise referred to as a neuron or kernel, scans the entirety of the input image and convolves with the pixel intensities by multiplying the filter values with the pixel intensity values depicted within the original input image. Upon all the pixels within the current receptive field being considered, a summation of all the multiplications is computed to produce a single output which represents the filter convolution within one location of the image. Furthermore, the convolutional layer then slides the filter with a set stride value and repeats the same order of operations, this continues until the entire grid of pixels within the input image have been considered to produce an activation map. Similarly, this same process is applied with multiple filters to preserve the spatial domain of the original input image as the data is propagated through the network, such that each new filter would become a new dimension of the final activation map output from the convolution layer, for example the first convolutional layer of the CNN model developed within this assignment features a total of 32 filters, therefore the dimensional size of the resulting activation map would be 28 x 28 x 32.

As such, a rectifier linear unit (ReLU) activation function subsequently operates on the generated activation map and ultimately determines whether a neuron should be activated by ensuring that all activation values evaluates to a value above zero. For example, any negative neuron would be ignored and assigned a value of 0, whereas any positive neuron value would be activated by the function and maintained at a constant positive value. In a comparative study exploring different selections of activation functions conducted by Ertam and Aydin (2017), the study found that the ReLU activation function accomplished the best classification performance of 98.43% with the test dataset in comparison to rivalling activation functions such as tanh and sigmoid. Thus, the ReLU activation function was selected as it could be argued to provide the most effective and efficient means for achieving an overall higher accuracy outcome from the CNN model.

However, a singular convolutional layer can be argued to be insufficient regarding the extraction of intelligible features from a batch of input images, therefore a second and third convolution layer were subsequently incorporated into the CNN architecture. Furthermore, the second layer will perform the same process of operations with 64 filters, where the activation map output extracted from the initial convolution layer will be used as the new input for the second convolutional layer. The initial convolutional layer provided the capabilities toward extracting low-level features such as lines and curves, however the application of additional features with these features can provide insight into higher level feature activation maps such as a combination of lines and curves. To surmise, the final convolutional layer activation map output should represent increasingly more complex features that reside within each training image sample, such that the final filters should activate when the various handwritten digits can be recognised.

A pooling layer was applied after the final convolutional layer, this operated such that a new small filter was applied which scanned the entirety of the output activation map and effectively down-sampled the matrix by spatially resizing each output matrix. Specifically, this operated such that a filter of size 2 x 2 was applied for each pixel pertaining to the previously output activation map at a stride value of 2, the pooling layer subsequently resized the original activation map by taking the highest pixel intensity within the pooling filter window and using it as representative of the new pixel intensity in the down-sampled version of the activation map. Additionally, a total of two dropout layers are featured within the developed CNN model; namely one after the previously elaborated max-pooling layer and one after a fully connected layer, whereby half of the neurons procured within the previous layer of the CNN model are deactivated and effectively dropped from the network. As such, the max pooling and dropout layers help to generalise the model and ensure that the model isn't tuned specifically for the training dataset, which in turn reduces the issue of overfitting.

However, the classification phase of the CNN encompasses around the implementation of a fully-connected layer, otherwise referred to as a dense layer. As such, the dimensionality of the output from the previous layer is flattened, such that all the previously calculated feature maps are collated into a single dimension and therefore producing a new 1D feature vector output. Effectively, the fully connected layer collates all the high-level features learned from the previous layer and uses this knowledge to determine which features are likely to correlate to each digit classification, for example determining if a high-level feature describes a '1' digit classification or a '2' digit classification. The activation map output from the fully connected layer is subsequently input into a softmax function, whereby the previously determined feature correlations are estimated within a probabilistic range of real values between 0 and 1, where a higher probabilistic value is indicative a higher likelihood that an input image belongs to a particular digit classification.

Results

The following section will entail a comparison between the previously implemented machine learning classifiers, thus discussing the overall performance of each of the models for recognising the handwritten digit images while also considering any advantages and disadvantages each approach may advocate. To test the three models, each model was trained with the training dataset and subsequent tested against the test dataset, whereby a confusion matrix was computed to compare the predicted values as each of the models was fit with the test dataset with the expected classification values of the test dataset (Polat et al, 2009). As elaborated by Kuhn and Johnson (2013), true positives (TP) and true negatives (TN) describe the cases where a classifier model correctly predicted the digit classification in correlation with the true classification value, whereas false positives (FP) and false negatives (FN) describe the cases where a different classification value was predicted in comparison to the true classification value.

Table 1. Output classification performance for each of the previously discussed models.

	Decision Tree	**SVM**	**CNN**
Accuracy Score	81.6%	91.4%	97.6%
Precision	82%	91%	98%
Recall	82%	91%	98%
F1-Score	82%	91%	98%

$$Precision = \frac{TP}{TP+FP} * 100$$

Formula 2. Formula for precision metric.

$$Recall = \frac{TP}{TP+FN} * 100$$

Formula 3. Formula for recall metric.

The precision metric measures the total number of correct predictions as a proportion of the number of positive predictions that were made, Buckland and Gey (1994) argue therefore that precision can be observed as a measure of purity regarding the successful prediction retrieval performance and the capability toward excluding nonrelevant classifications from the training dataset. Similarly, the recall metric can be perceived as a metric which measures the number of correct predictions as a proportion of all relevant predictions that were made, which as a result can be used to assess the classification performance with regards to the model effectiveness at including correct predictions within the set of predictions that were made (Buckland and Gey, 1994). To this end, all three classification models achieved a good performance, with the SVM and CNN models achieving a particularly high performance with a precision and recall value higher than 90%, demonstrating that the classification models can predict the correct digit classification more than incorrectly classifying a digit, while also demonstrating that the classifier classified most of the possible correct digit classifications. However, it can be observed that the decision tree classifier yielded a slightly lower precision and recall score of 82% for both metrics, therefore it could be inferred that the utilised decision tree classifier was underwhelming regarding the model classification robustness.

$$F_1Score = 2 * \frac{precision*recall}{precision+recall} * 100$$

Formula 4. Formula for F1-score metric.

$$Accuracy = \frac{TP+TN}{TP+TN+FP+FN} * 100$$

Formula 5. Formula for accuracy score metric.

Similarly, an F1-score, otherwise referred to as F-measure, was computed to estimate the harmonic mean of the previously calculated precision and recall metrics. Merely comparing the precision and recall metrics can be argued as being insufficient regarding assessing how well the developed classifiers accomplish the given task, thus the F1-score effectively creates a combined metric to compare the performance of the three classifiers. However, as the previously calculated precision and recall values were arguably high and consistent, for example the SVM classifier achieved a performance of 91% with both the precision and recall metrics, the calculation of the F-measure provided minimal additional inference with regards to the model performance.

However, an imperative objective which was being accomplished with the classifiers was developing a robust classification model which was capable of accurately classifying a high proportion of the test dataset with the correct classification, for example ensuring that the trained model is capable of correctly recognising the digit ‘7’ using the features learned from the training dataset. As such, the classification accuracy score would be estimated by comparing the total number of correct predictions, namely TP and TN instances extracted from the confusion matrix, with all the predictions that were made by the model. To this end, the decision tree model appeared to have acquired a low classification accuracy score of 81.6%, whereas the SVM classifier and the CNN approach procured high accuracy scores of 91.4% and 97.6% respectively. However, while it can be argued that the CNN architecture provided a higher performance overall by taking advantage of multi-layer processing available within a neural network, the SVM was also observed to obtain a good accuracy performance while requiring a lower amount of time to fully train the SVM model. As a result, it can be argued that these results depict a possible implementation trade-off, where an SVM classifier provides a model implementation requiring a lower training time whereas the CNN approach provides a considerable increase with the classification accuracy score.

In conclusion, the model which could be argued as being the best classifier for the currently presented task of recognising images of handwritten digits is the CNN approach. Despite requiring a larger amount of time during the training phase of the classifier, the CNN model can also be observed as being fast when predicting the new classifications for new digit image samples contained within the test dataset. Zhong et al (2018) suggest that deep learning algorithms such as a CNN architecture are being incorporated into many real-world applications at an impressive rate, where the traditional or state-of-the-art approaches towards a problem such as image classification can be argued to be considerably improved through deep-learning. Furthermore, the best accuracy performance provided through the CNN approach suggests that this model would likely perform well if additional digit examples were provided without an immediate necessity to acquire more data for the training dataset to which the CNN would inherit from.

Reference List

Buckland, M., Gey, F. (1994) The Relationship between Recall and Precision. *Journal of the American Society for Information Science*, 45(1) 12-19. Available from: https://search.proquest.com/openview/a74e8fe9fc458a7e5b1189ca52268178/1?pq-origsite=gscholar&cbl=1818555 [accessed: 27-04-2018].

Ertam, F., Aydin, G. (2017) Data classification with deep learning using Tensorflow. In: *(UBMK'17) 2nd International Conference on Computer Science and Engineering*, Antalya, Turkey, 5-8 Oct. New York: USA: IEEE, 755-758. Available from: https://ieeexplore-ieee-org.proxy.library.lincoln.ac.uk/document/8093521/ [accessed: 01-05-2018].

Fedorovici, L., Dragan, F. (2011) A Comparison between a Neural Network and a SVM and Zernike Moments Based Blob Recognition Modules. In: *6th IEEE International Symposium on Applied Computational Intelligence and Informatics*, Timişoara, Romania, 19-21 May. New York, USA: IEEE, 253-258. Available from: https://ieeexplore-ieee-org.proxy.library.lincoln.ac.uk/document/5873009/ [accessed: 20-04-2018].

Galar, M., Fernández, A., Barrenechea, E., Bustince, H., Herrera, F. (2011) An overview of ensemble methods for binary classifiers in multi-class problems: experimental study on one-vs-one and one-vs-all schemes. *Elsevier Pattern Recognition*, 44(8) 1761-1776. Available from: https://www.sciencedirect.com/science/article/pii/S0031320311000458 [accessed: 22-04-2018].

Kemal, P., Sadik, K., Ayşegül, G., Salih, G. (2009) Comparison of different classifier algorithms for diagnosing macular and optic nerve diseases. *Expert Systems - The Journal of Knowledge Engineering*, 26(1) 22-34. Available from: http://eds.a.ebscohost.com.proxy.library.lincoln.ac.uk/eds/pdfviewer/pdfviewer?vid=2&sid=e745106c-432f-40d8-883a-e78564d1cacc%40sessionmgr4009 [accessed: 24-04-2018].

Keras (2017) *Keras: The Python Deep Learning library.* Available from: https://keras.io/ [accessed: 01-04-2018].

Kuhn, M., Johnson, K. (2013) *Applied Predictive Modeling*. 1st edition. United States of America: Springer Publishing.

Kulkarni, A., Shrestha, A. (2017) Multispectral image analysis using decision trees. *International Journal of Advanced Computer Science and Applications*, 8(6) 11-18. Available from: https://thesai.org/Downloads/Volume8No6/Paper_2-Multispectral_Image_Analysis_using_Decision_Trees.pdf [accessed: 24-04-2018].

Lu, K., Yang, D. (2009) Image Processing and Image Mining using Decision Trees. *Journal of Information Science and Engineering*, 25(4) 989-1003. Available from: https://pdfs.semanticscholar.org/d70e/dd5b477924d58395c57d8aaea5ffa513afc0.pdf [accessed: 22-04-2018].

Murphy, K. (2012) *Machine Learning: a probabilistic perspective*. London: MIT Press.

Scikit-Learn (2017) *1.10. Decision Trees*. Available from: http://scikit-learn.org/stable/modules/tree.html [accessed: 09-04-2018].

Scikit-Learn (2017) *1.4. Support Vector Machines*. Available from: http://scikit-learn.org/stable/modules/svm.html [accessed: 08-04-2018].

Zhong, G., Ling, X., Wang, L (2018) From shallow feature learning to deep learning: benefits from the width and depth of deep architectures. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 0(0) 1-14. Available from: https://onlinelibrary-wileycom.proxy.library.lincoln.ac.uk/doi/full/10.1002/widm.1255 [accessed: 24-04-2018].

Appendix A: Python 3.6 script for implementing the Decision Tree classifier.

It should be noted that the Scikit-Learn (2017) library, sklearn, was used to implement the decision tree model that was discussed within this report.

```
import cv2
import os
import glob
import numpy as np
from sklearn.utils import shuffle
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score
from sklearn import metrics

def rgb2gray(rgb):
    r, g, b = rgb[:,:,0], rgb[:,:,1], rgb[:,:,2]
    gray = 0.2989 * r + 0.5870 * g + 0.1140 * b

    return gray

def load_dataset(dataset_path, classes):
    images = []
    labels = []
    img_names = []
    cls = [] #classes
    for fields in classes:
        index = classes.index(fields)
        print('Now going to read {} files (Index: {})'.format(fields,
index))
        path = os.path.join(dataset_path, fields, '*g')
        files = glob.glob(path)
        for fl in files:
            # Reading the image using OpenCV
            bgr_img = cv2.imread(fl)
            b,g,r = cv2.split(bgr_img)       # get b,g,r

            rgb_img = cv2.merge([r,g,b])     # switch it to rgb
            rgb_img = rgb_img.astype(np.float32)
            rgb_img = np.multiply(rgb_img, 1.0 / 255.0)

            gray_img = rgb2gray(rgb_img)
            images.append(gray_img)
            labels.append(index)

            flbase = os.path.basename(fl)
            img_names.append(flbase)
            cls.append(fields)
    images = np.array(images)
    labels = np.array(labels)
    img_names = np.array(img_names)
    cls = np.array(cls)

    return images, labels, img_names, cls

def disp_performance(clf,expected,predicted):
    print("Classification report for classifier %s:%s\n" % (clf,
metrics.classification_report(expected, predicted)))
    print("Confusion matrix:\n%s" % metrics.confusion_matrix(expected,
predicted))
    acc = accuracy_score(expected,predicted)
    print("\nClassification Accuracy Score: %s" % acc)
```

```
def classify_DecisionTree(features,labels):
    print("Training Decision Tree classifier...")
    n_samples = len(features)
    data = features.reshape((n_samples, -1))
    clf = tree.DecisionTreeClassifier(criterion='entropy')
    clf.fit(data, labels)

    return clf

################################################################################
######################### DATA ACQUISITION #####################################
################################################################################

classes = ['0','1','2','3','4','5','6','7','8','9']
train_in = ('.\dataset\\training\\')
valid_in = ('.\dataset\\validation\\')
test_in = ('.\dataset\\test\\')

# load input datasets. X___ = image features, Y___ = image labels. Store
image names for the shuffle function.
print("Loading training dataset...\n")
Xtrain, Ytrain, train_img_names, train_cls = load_dataset(train_in,
classes)
Xtrain, Ytrain, train_img_names, train_cls = shuffle(Xtrain, Ytrain,
train_img_names, train_cls)

print("\nLoading validation dataset...\n")
Xvalid, Yvalid, valid_img_names, valid_cls = load_dataset(valid_in,
classes)
Xvalid, Yvalid, valid_img_names, valid_cls = shuffle(Xvalid, Yvalid,
valid_img_names, valid_cls)

print("\nLoading test dataset...\n")
Xtest, Ytest, test_img_names, test_cls = load_dataset(test_in, classes)
Xtest, Ytest, test_img_names, test_cls = shuffle(Xtest, Ytest,
test_img_names, test_cls)

print("\nLoading Complete!\n")

################################################################################
######################### CLASSIFICATION PHASE #################################
################################################################################

epochs = 12
img_rows = 28
img_cols = 28
batch_size = 128
num_classes = len(classes)

dt_clf = classify_DecisionTree(Xtrain,Ytrain)

n_samples = len(Xtest)
expected = Ytest # for the sake of clarity

predicted_dt = dt_clf.predict(Xtest.reshape(n_samples,-1))

disp_performance(dt_clf,expected,predicted_dt)
```

Appendix B: Python 3.6 script for implementing SVM classifier.

It should be noted that the Scikit-Learn (2017) library, sklearn, was used to implement the SVM model that was discussed within this report.

```
import cv2
import os
import glob
import numpy as np
from sklearn.utils import shuffle
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score
from sklearn import metrics

def rgb2gray(rgb):
    r, g, b = rgb[:,:,0], rgb[:,:,1], rgb[:,:,2]
    gray = 0.2989 * r + 0.5870 * g + 0.1140 * b

    return gray

def load_dataset(dataset_path, classes):
    images = []
    labels = []
    img_names = []
    cls = [] #classes
    for fields in classes:
        index = classes.index(fields)
        print('Now going to read {} files (Index: {})'.format(fields,
index))
        path = os.path.join(dataset_path, fields, '*g')
        files = glob.glob(path)
        for fl in files:
            # Reading the image using OpenCV
            bgr_img = cv2.imread(fl)
            b,g,r = cv2.split(bgr_img)       # get b,g,r

            rgb_img = cv2.merge([r,g,b])     # switch it to rgb
            rgb_img = rgb_img.astype(np.float32)
            rgb_img = np.multiply(rgb_img, 1.0 / 255.0)

            gray_img = rgb2gray(rgb_img)
            images.append(gray_img)
            labels.append(index)

            flbase = os.path.basename(fl)
            img_names.append(flbase)
            cls.append(fields)
    images = np.array(images)
    labels = np.array(labels)
    img_names = np.array(img_names)
    cls = np.array(cls)

    return images, labels, img_names, cls

def disp_performance(clf,expected,predicted):
    print("Classification report for classifier %s:%s\n" % (clf,
metrics.classification_report(expected, predicted)))
    print("Confusion matrix:\n%s" % metrics.confusion_matrix(expected,
predicted))
    acc = accuracy_score(expected,predicted)
    print("\nClassification Accuracy Score: %s" % acc)
```

```
def classify_SVM(features,labels):
    print("Training SVM classifier...")
    n_samples = len(features)
    data = features.reshape((n_samples, -1))
    clf = SVC()
    clf.fit(data, labels)

    return clf

##############################################################################
######################### DATA ACQUISITION ###################################
##############################################################################

classes = ['0','1','2','3','4','5','6','7','8','9']
train_in = ('.\dataset\\training\\')
valid_in = ('.\dataset\\validation\\')
test_in = ('.\dataset\\test\\')

# load input datasets. X___ = image features, Y___ = image labels. Store
image names for the shuffle function.
print("Loading training dataset...\n")
Xtrain, Ytrain, train_img_names, train_cls = load_dataset(train_in,
classes)
Xtrain, Ytrain, train_img_names, train_cls = shuffle(Xtrain, Ytrain,
train_img_names, train_cls)

print("\nLoading validation dataset...\n")
Xvalid, Yvalid, valid_img_names, valid_cls = load_dataset(valid_in,
classes)
Xvalid, Yvalid, valid_img_names, valid_cls = shuffle(Xvalid, Yvalid,
valid_img_names, valid_cls)

print("\nLoading test dataset...\n")
Xtest, Ytest, test_img_names, test_cls = load_dataset(test_in, classes)
Xtest, Ytest, test_img_names, test_cls = shuffle(Xtest, Ytest,
test_img_names, test_cls)

print("\nLoading Complete!\n")

##############################################################################
######################### CLASSIFICATION PHASE ###############################
##############################################################################

epochs = 12
img_rows = 28
img_cols = 28
batch_size = 128
num_classes = len(classes)

svm_clf = classify_SVM(Xtrain,Ytrain)

n_samples = len(Xtest)
expected = Ytest # for the sake of clarity

predicted_SVM = svm_clf.predict(Xtest.reshape(n_samples,-1))

disp_performance(svm_clf,expected,predicted_SVM)
```

Appendix C: Python 3.6 script for implementing the CNN architecture

It should be noted that the library Keras (2018) was used as a fundamental component when designing the previously discussed CNN architecture.

```
import keras
import cv2
import os
import glob
import numpy as np
from sklearn.utils import shuffle
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score
from sklearn import metrics
from sklearn import tree
from sklearn.metrics import classification_report
from keras.models import Sequential
from keras.layers import Dense, Dropout, Flatten
from keras.layers import Conv2D, MaxPooling2D
from keras import backend as K

def rgb2gray(rgb):
    r, g, b = rgb[:,:,0], rgb[:,:,1], rgb[:,:,2]
    gray = 0.2989 * r + 0.5870 * g + 0.1140 * b

    return gray

def load_dataset(dataset_path, classes):
    images = []
    labels = []
    img_names = []
    cls = [] #classes
    for fields in classes:
        index = classes.index(fields)
        print('Now going to read {} files (Index: {})'.format(fields,
index))
        path = os.path.join(dataset_path, fields, '*g')
        files = glob.glob(path)
        for fl in files:
            # Reading the image using OpenCV
            bgr_img = cv2.imread(fl)
            b,g,r = cv2.split(bgr_img)       # get b,g,r

            rgb_img = cv2.merge([r,g,b])     # switch it to rgb
            rgb_img = rgb_img.astype(np.float32)
            rgb_img = np.multiply(rgb_img, 1.0 / 255.0)

            gray_img = rgb2gray(rgb_img)
            images.append(gray_img)
            labels.append(index)

            flbase = os.path.basename(fl)
            img_names.append(flbase)
            cls.append(fields)
    images = np.array(images)
    labels = np.array(labels)
    img_names = np.array(img_names)
    cls = np.array(cls)

    return images, labels, img_names, cls
```

```
def reshape_Features(features,img_rows,img_cols,shapeFlag):
    if K.image_data_format() == 'channels_first':
        features = features.reshape(features.shape[0], 1, img_rows,
img_cols)
        input_shape = (1, img_rows, img_cols)
    else:
        features = features.reshape(features.shape[0], img_rows, img_cols,
1)
        input_shape = (img_rows, img_cols, 1)
    if shapeFlag == 1:
        return features,input_shape
    else:
        return features

def disp_performance_cnn(expected,predicted):
    print("\nClassification report for CNN softmax classifier:\n")
    report = classification_report(expected, predicted)
    print(report)
    acc = accuracy_score(expected,predicted)
    print("\nClassification Accuracy Score: %s" % acc)

def
classify_CNN(x_train,y_train,x_valid,y_valid,batch_size,epochs,num_classes)
:
    print("Training CNN...")
    img_rows = 28
    img_cols = 28

    if K.image_data_format() == 'channels_first':
        x_train = x_train.reshape(x_train.shape[0], 1, img_rows, img_cols)
        x_valid = x_valid.reshape(x_valid.shape[0], 1, img_rows, img_cols)
        input_shape = (1, img_rows, img_cols)
    else:
        x_train = x_train.reshape(x_train.shape[0], img_rows, img_cols, 1)
        x_valid = x_valid.reshape(x_valid.shape[0], img_rows, img_cols, 1)
        input_shape = (img_rows, img_cols, 1)

    print('x_train shape:', x_train.shape)
    print(x_train.shape[0], 'train samples')
    print(x_valid.shape[0], 'validation samples')

    # convert class vectors to binary class matrices
    y_train = keras.utils.to_categorical(y_train, num_classes)
    y_valid = keras.utils.to_categorical(y_valid, num_classes)

    clf = Sequential()
    clf.add(Conv2D(32, kernel_size=(3, 3),
                   activation='relu',
                   input_shape=input_shape))
    clf.add(Conv2D(64, (3, 3), activation='relu'))
    clf.add(MaxPooling2D(pool_size=(2, 2)))
    clf.add(Dropout(0.25))
    clf.add(Flatten())
    clf.add(Dense(128, activation='relu'))
    clf.add(Dropout(0.5))
    clf.add(Dense(num_classes, activation='softmax'))

    clf.compile(loss=keras.losses.categorical_crossentropy,
                optimizer=keras.optimizers.Adadelta(),
                metrics=['accuracy'])
```

```
    clf.fit(x_train, y_train,
              batch_size=batch_size,
              epochs=epochs,
              verbose=1,
              validation_data=(x_valid, y_valid))

    return clf

###############################################################################
######################### DATA ACQUISITION ####################################
###############################################################################

classes = ['0','1','2','3','4','5','6','7','8','9']
train_in = ('.\dataset\\training\\')
valid_in = ('.\dataset\\validation\\')
test_in = ('.\dataset\\test\\')

# load input datasets. X___ = image features, Y___ = image labels. Store
image names for the shuffle function.
print("Loading training dataset...\n")
Xtrain, Ytrain, train_img_names, train_cls = load_dataset(train_in,
classes)
Xtrain, Ytrain, train_img_names, train_cls = shuffle(Xtrain, Ytrain,
train_img_names, train_cls)

print("\nLoading validation dataset...\n")
Xvalid, Yvalid, valid_img_names, valid_cls = load_dataset(valid_in,
classes)
Xvalid, Yvalid, valid_img_names, valid_cls = shuffle(Xvalid, Yvalid,
valid_img_names, valid_cls)

print("\nLoading test dataset...\n")
Xtest, Ytest, test_img_names, test_cls = load_dataset(test_in, classes)
Xtest, Ytest, test_img_names, test_cls = shuffle(Xtest, Ytest,
test_img_names, test_cls)

print("\nLoading Complete!\n")

###############################################################################
######################### CLASSIFICATION PHASE ################################
###############################################################################

epochs = 12
img_rows = 28
img_cols = 28
batch_size = 128
num_classes = len(classes)

cnn_clf =
classify_CNN(Xtrain,Ytrain,Xvalid,Yvalid,batch_size,epochs,num_classes)

n_samples = len(Xtest)

Xtest = reshape_Features(Xtest,img_rows,img_cols,0)
Ytest = keras.utils.to_categorical(Ytest, num_classes)

predicted_CNN = cnn_clf.predict_classes(Xtest)
predicted_CNN = keras.utils.to_categorical(predicted_CNN, num_classes)

score = cnn_clf.evaluate(Xtest, Ytest)
```

```
disp_performance_cnn(Ytest,predicted_CNN)
```

Appendix D: Python 3.6 script for data extraction

The following script defines the methodology undertaken for separating the input datasets into class-specific folders to enable the training of the various classification models.

```
from PIL import Image
import os

inputDir1 = ('.\digits-train-5000')
inputDir2 = ('.\digits-validation-1000')
inputDir3 = ('.\digits-test-500')

outTrainDir0 = ('.\dataset\\training\\0')
outTrainDir1 = ('.\dataset\\training\\1')
outTrainDir2 = ('.\dataset\\training\\2')
outTrainDir3 = ('.\dataset\\training\\3')
outTrainDir4 = ('.\dataset\\training\\4')
outTrainDir5 = ('.\dataset\\training\\5')
outTrainDir6 = ('.\dataset\\training\\6')
outTrainDir7 = ('.\dataset\\training\\7')
outTrainDir8 = ('.\dataset\\training\\8')
outTrainDir9 = ('.\dataset\\training\\9')

outValidDir0 = ('.\dataset\\validation\\0')
outValidDir1 = ('.\dataset\\validation\\1')
outValidDir2 = ('.\dataset\\validation\\2')
outValidDir3 = ('.\dataset\\validation\\3')
outValidDir4 = ('.\dataset\\validation\\4')
outValidDir5 = ('.\dataset\\validation\\5')
outValidDir6 = ('.\dataset\\validation\\6')
outValidDir7 = ('.\dataset\\validation\\7')
outValidDir8 = ('.\dataset\\validation\\8')
outValidDir9 = ('.\dataset\\validation\\9')

outTestDir0 = ('.\dataset\\test\\0')
outTestDir1 = ('.\dataset\\test\\1')
outTestDir2 = ('.\dataset\\test\\2')
outTestDir3 = ('.\dataset\\test\\3')
outTestDir4 = ('.\dataset\\test\\4')
outTestDir5 = ('.\dataset\\test\\5')
outTestDir6 = ('.\dataset\\test\\6')
outTestDir7 = ('.\dataset\\test\\7')
outTestDir8 = ('.\dataset\\test\\8')
outTestDir9 = ('.\dataset\\test\\9')

if not os.path.exists(outTrainDir0):
    os.makedirs(outTrainDir0)
if not os.path.exists(outTrainDir1):
    os.makedirs(outTrainDir1)
if not os.path.exists(outTrainDir2):
    os.makedirs(outTrainDir2)
if not os.path.exists(outTrainDir3):
    os.makedirs(outTrainDir3)
if not os.path.exists(outTrainDir4):
    os.makedirs(outTrainDir4)
if not os.path.exists(outTrainDir5):
    os.makedirs(outTrainDir5)
if not os.path.exists(outTrainDir6):
    os.makedirs(outTrainDir6)
if not os.path.exists(outTrainDir7):
    os.makedirs(outTrainDir7)
```

```
if not os.path.exists(outTrainDir8):
    os.makedirs(outTrainDir8)
if not os.path.exists(outTrainDir9):
    os.makedirs(outTrainDir9)

if not os.path.exists(outValidDir0):
    os.makedirs(outValidDir0)
if not os.path.exists(outValidDir1):
    os.makedirs(outValidDir1)
if not os.path.exists(outValidDir2):
    os.makedirs(outValidDir2)
if not os.path.exists(outValidDir3):
    os.makedirs(outValidDir3)
if not os.path.exists(outValidDir4):
    os.makedirs(outValidDir4)
if not os.path.exists(outValidDir5):
    os.makedirs(outValidDir5)
if not os.path.exists(outValidDir6):
    os.makedirs(outValidDir6)
if not os.path.exists(outValidDir7):
    os.makedirs(outValidDir7)
if not os.path.exists(outValidDir8):
    os.makedirs(outValidDir8)
if not os.path.exists(outValidDir9):
    os.makedirs(outValidDir9)

if not os.path.exists(outTestDir0):
    os.makedirs(outTestDir0)
if not os.path.exists(outTestDir1):
    os.makedirs(outTestDir1)
if not os.path.exists(outTestDir2):
    os.makedirs(outTestDir2)
if not os.path.exists(outTestDir3):
    os.makedirs(outTestDir3)
if not os.path.exists(outTestDir4):
    os.makedirs(outTestDir4)
if not os.path.exists(outTestDir5):
    os.makedirs(outTestDir5)
if not os.path.exists(outTestDir6):
    os.makedirs(outTestDir6)
if not os.path.exists(outTestDir7):
    os.makedirs(outTestDir7)
if not os.path.exists(outTestDir8):
    os.makedirs(outTestDir8)
if not os.path.exists(outTestDir9):
    os.makedirs(outTestDir9)

for file in os.listdir(inputDir1):
    fileName = os.fsdecode(file)
    selectedNumber = int(fileName[0])
    selectedFilePath = inputDir1 + '\\' + fileName
    im = Image.open(selectedFilePath)
    if (selectedNumber == 0):
        newFilePath = outTrainDir0 + '\\' + fileName
        im.save(newFilePath)
    elif (selectedNumber == 1):
        newFilePath = outTrainDir1 + '\\' + fileName
        im.save(newFilePath)
    elif (selectedNumber == 2):
        newFilePath = outTrainDir2 + '\\' + fileName
        im.save(newFilePath)
```

```
    elif (selectedNumber == 3):
        newFilePath = outTrainDir3 + '\\' + fileName
        im.save(newFilePath)
    elif (selectedNumber == 4):
        newFilePath = outTrainDir4 + '\\' + fileName
        im.save(newFilePath)
    elif (selectedNumber == 5):
        newFilePath = outTrainDir5 + '\\' + fileName
        im.save(newFilePath)
    elif (selectedNumber == 6):
        newFilePath = outTrainDir6 + '\\' + fileName
        im.save(newFilePath)
    elif (selectedNumber == 7):
        newFilePath = outTrainDir7 + '\\' + fileName
        im.save(newFilePath)
    elif (selectedNumber == 8):
        newFilePath = outTrainDir8 + '\\' + fileName
        im.save(newFilePath)
    elif (selectedNumber == 9):
        newFilePath = outTrainDir9 + '\\' + fileName
        im.save(newFilePath)
    else:
        continue

for file in os.listdir(inputDir2):
    fileName = os.fsdecode(file)
    selectedNumber = int(fileName[0])
    selectedFilePath = inputDir2 + '\\' + fileName
    im = Image.open(selectedFilePath)
    if (selectedNumber == 0):
        newFilePath = outValidDir0 + '\\' + fileName
        im.save(newFilePath)
    elif (selectedNumber == 1):
        newFilePath = outValidDir1 + '\\' + fileName
        im.save(newFilePath)
    elif (selectedNumber == 2):
        newFilePath = outValidDir2 + '\\' + fileName
        im.save(newFilePath)
    elif (selectedNumber == 3):
        newFilePath = outValidDir3 + '\\' + fileName
        im.save(newFilePath)
    elif (selectedNumber == 4):
        newFilePath = outValidDir4 + '\\' + fileName
        im.save(newFilePath)
    elif (selectedNumber == 5):
        newFilePath = outValidDir5 + '\\' + fileName
        im.save(newFilePath)
    elif (selectedNumber == 6):
        newFilePath = outValidDir6 + '\\' + fileName
        im.save(newFilePath)
    elif (selectedNumber == 7):
        newFilePath = outValidDir7 + '\\' + fileName
        im.save(newFilePath)
    elif (selectedNumber == 8):
        newFilePath = outValidDir8 + '\\' + fileName
        im.save(newFilePath)
    elif (selectedNumber == 9):
        newFilePath = outValidDir9 + '\\' + fileName
        im.save(newFilePath)
    else:
        continue
```

```
for file in os.listdir(inputDir3):
    fileName = os.fsdecode(file)
    selectedNumber = int(fileName[0])
    selectedFilePath = inputDir3 + '\\' + fileName
    im = Image.open(selectedFilePath)
    if (selectedNumber == 0):
        newFilePath = outTestDir0 + '\\' + fileName
        im.save(newFilePath)
    elif (selectedNumber == 1):
        newFilePath = outTestDir1 + '\\' + fileName
        im.save(newFilePath)
    elif (selectedNumber == 2):
        newFilePath = outTestDir2 + '\\' + fileName
        im.save(newFilePath)
    elif (selectedNumber == 3):
        newFilePath = outTestDir3 + '\\' + fileName
        im.save(newFilePath)
    elif (selectedNumber == 4):
        newFilePath = outTestDir4 + '\\' + fileName
        im.save(newFilePath)
    elif (selectedNumber == 5):
        newFilePath = outTestDir5 + '\\' + fileName
        im.save(newFilePath)
    elif (selectedNumber == 6):
        newFilePath = outTestDir6 + '\\' + fileName
        im.save(newFilePath)
    elif (selectedNumber == 7):
        newFilePath = outTestDir7 + '\\' + fileName
        im.save(newFilePath)
    elif (selectedNumber == 8):
        newFilePath = outTestDir8 + '\\' + fileName
        im.save(newFilePath)
    elif (selectedNumber == 9):
        newFilePath = outTestDir9 + '\\' + fileName
        im.save(newFilePath)
    else:
        continue
```