Homework 4

MNIST Handwritten Digits Recognition

Phys 243

06/01/19

Faddy Sunna

DATA: MNIST Handwritten digits. All 60,000 images were used for training, and 10,000 images for testing. No preprocessing was needed.

At first it was very time consuming to run the whole dataset because sklearn was only using 1 cpu core. I realized that adding, n_jobs = -1 , allows it to take advantage of all CPU cores, greatly speeding up the runtime.

1.Finding all the 9s, using KNN Sklearn.
First, I found the optimal K value by testing values (1, 3, 5, 7) and measuring the misclassification error. K = 3 had the lowest misclassification error, and thus was used for the rest of the KNN models. Fig 1: Code used to find the optimal K, by iterating k values and measuring misclassification error. I was able to switch from classifying just the 9s and all the digits by adjusting line 42 in figure 3.
Setting Minkowski P=1,2,3 parameters: Fig 3, line 39.
Table 1 and 2 show the final results of the KNN classifier.

FIND NINE	Minkowski	KNN
Mean Accuracy	**P**	**K neighbors**
94.50%	1	3
95.90%	2	3
96.80%	3	3

Table 1: Classifying just the 9s. Using KNN Minkowski 1,2,3.

FIND ALL	Minkowski	KNN
Mean Accuracy	**P**	**K neighbors**
96.30%	1	3
97.00%	2	3
97.18%	3	3

Table 2: Classifying all digits 0-9. Using KNN Minkowski 1,2,3.

```python
import struct
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split, cross_val_score
from sklearn.neighbors import KNeighborsClassifier

def read_idx(filename):
    with open(filename, 'rb') as f:
        zero, data_type, dims = struct.unpack('>HBB', f.read(4))
        shape = tuple(struct.unpack('>I', f.read(4))[0] for d in range(dims))
        return np.fromstring(f.read(), dtype=np.uint8).reshape(shape)

raw_train = read_idx("train-images-idx3-ubyte")
train_data = np.reshape(raw_train, (60000, 28*28))
train_label = read_idx("train-labels-idx1-ubyte")

raw_test = read_idx("t10k-images-idx3-ubyte")
test_data = np.reshape(raw_test, (10000, 28*28))
test_label = read_idx("t10k-labels-idx1-ubyte")

idx = (train_label ==0) | (train_label == 1) | (train_label == 2) | (train_label == 3) | (train_label == 4) | (train_label == 5) | (train_label == 6) | (train_label == 7) | (train_label == 8) |
x_train = train_data[idx]
y_train = train_label[idx]

idx = (test_label == 0) | (test_label == 1) | (test_label == 2) | (test_label == 3) | (test_label == 4) | (test_label == 5) | (test_label == 6) | (test_label == 7) | (test_label == 8) | (test_la
x_test = test_data[idx]
y_true = test_label[idx]

# creating odd list of K for KNN
myList = list(range(1,7))

# subsetting just the odd ones
neighbors = list(filter(lambda x: x % 2 != 0, myList))

# empty list that will hold cv scores
cv_scores = []

#15-fold cross validation comparing accuracy of k values

for k in neighbors:
    clf = KNeighborsClassifier(n_neighbors=k, n_jobs=-1)
    scores = cross_val_score(clf, x_test, y_true, cv=3, scoring='accuracy', n_jobs=-1)
    cv_scores.append(scores.mean())

# misclassification error
MSE = [1 - x for x in cv_scores]

# determining best k
optimal_k = neighbors[MSE.index(min(MSE))]
print("The optimal number of neighbors is %d" % optimal_k)

# plot misclassification error vs k
plt.plot(neighbors, MSE)
plt.xlabel('Number of Neighbors K')
plt.ylabel('Misclassification Error')
plt.show()
```

read_idx()

Fig 1 : Finding Optimal k , by iterating k=1,3,5,7, and measuring misclassification error

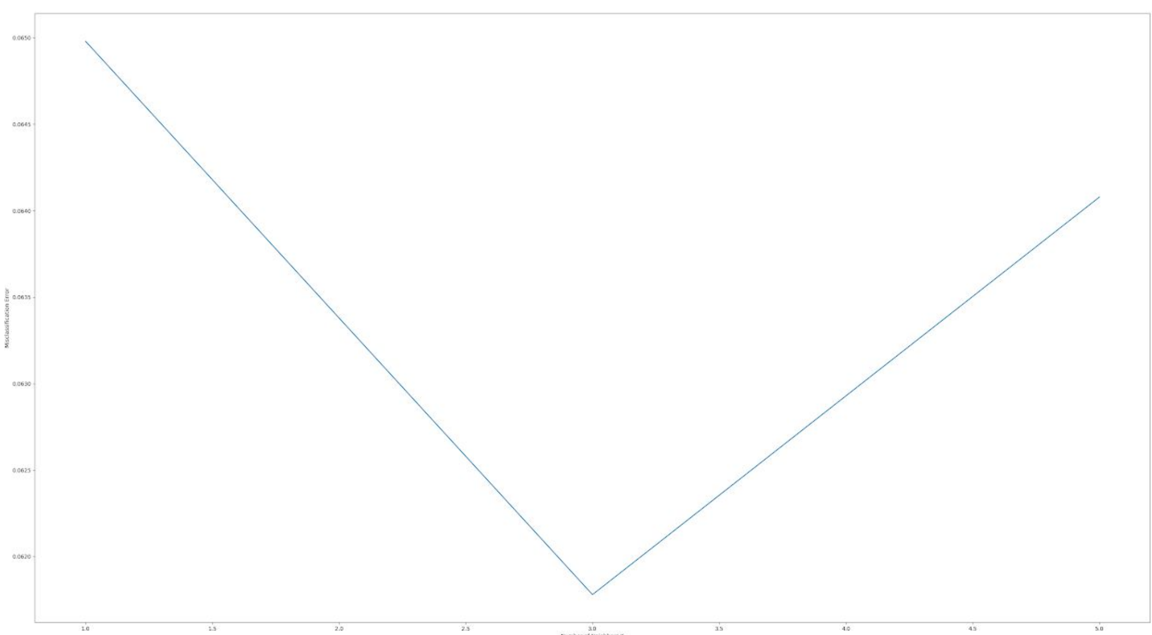

Fig 2: X-axis is k neighbors, Y-axis is misclassification error.

```python
import struct
import numpy as np
from sklearn import neighbors, metrics
import matplotlib.pyplot as plt
from sklearn import preprocessing, cross_decomposition, neighbors
from sklearn.neighbors import KNeighborsClassifier


#This code converts the MNIST idx format into numpy arrays
""" A function that can read MNIST's idx file format into numpy arrays.
    The MNIST data files can be downloaded from here:

    http://yann.lecun.com/exdb/mnist/
    This relies on the fact that the MNIST dataset consistently uses
    unsigned char types with their data segments.
"""
def read_idx(filename):
    with open(filename, 'rb') as f:
        zero, data_type, dims = struct.unpack('>HBB', f.read(4))
        shape = tuple(struct.unpack('>I', f.read(4))[0] for d in range(dims))
        return np.fromstring(f.read(), dtype=np.uint8).reshape(shape)


#import the train and test data and labels. Using all 60,000 images for training and 10,000 images for testing
raw_train = read_idx("train-images-idx3-ubyte")
train_data = np.reshape(raw_train, (60000, 28*28))
train_label = read_idx("train-labels-idx1-ubyte")

raw_test = read_idx("t10k-images-idx3-ubyte")
test_data = np.reshape(raw_test, (10000, 28*28))
test_label = read_idx("t10k-labels-idx1-ubyte")
#selecting all the data, digits 0-9 for training
idx = (train_label ==0) | (train_label == 1) | (train_label == 2) | (train_label == 3) | (train_label == 4) | (train_label == 5) | (train_label == 6) | (train_label == 7) | (train_label =
x = train_data[idx]
y = train_label[idx]


#Creating our knn classifier, and setting paramters to minkowski and p=1,2,3, then fitting the data
knn = neighbors.KNeighborsClassifier(n_neighbors=3, metric='minkowski', p=3, n_jobs=-1).fit(x, y)

#selecting our test data, in this case, it will be all the digits
idx = (test_label == 0) | (test_label == 1) | (test_label == 2) | (test_label == 3) | (test_label == 4) | (test_label == 5) | (test_label == 6) | (test_label == 7) | (test_label == 8) | (
x_test = test_data[idx]
y_true = test_label[idx]
y_pred = knn.predict(x_test)

import itertools

#Create confusion matrix
def plot_confusion_matrix(cm, classes,
                          normalize=False,
                          title='Confusion matrix',
                          cmap=plt.cm.Blues):
    """
    This function prints and plots the confusion matrix.
    Normalization can be applied by setting `normalize=True`.
    if normalize:
        cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]
        print("Normalized confusion matrix")
    else:
        print('Confusion matrix, without normalization')

    print(cm)

    plt.imshow(cm, interpolation='nearest', cmap=cmap)
    plt.title(title)
    plt.colorbar()
    tick_marks = np.arange(len(classes))
    plt.xticks(tick_marks, classes, rotation=45)
    plt.yticks(tick_marks, classes)

    fmt = '.2f' if normalize else 'd'
    thresh = cm.max() / 2.
    for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
        plt.text(j, i, format(cm[i, j], fmt),
                 horizontalalignment="center",
                 color="white" if cm[i, j] > thresh else "black")

    plt.tight_layout()
    plt.ylabel('True label')
    plt.xlabel('Predicted label')

#Calculate mean accuracy using built in score function
score = KNeighborsClassifier.score(knn, x_test, y_true)
scorestr = str(score)
print("The mean accuracy score using minkowski metric and p=3 is: " + scorestr)
#plot confusion matrix non normalized
cm = metrics.confusion_matrix(y_true, y_pred)
plot_confusion_matrix(cm, ["0", "1", "2", "3", "4", "5", "6", "7", "8", "9"])
plt.show()

#plot normalized confsuion matrix
plot_confusion_matrix(cm, ["0", "1", "2", "3", "4", "5", "6", "7", "8", "9"], normalize=True)
plt.show()
```

Fig 3: KNN Sklearn classifier

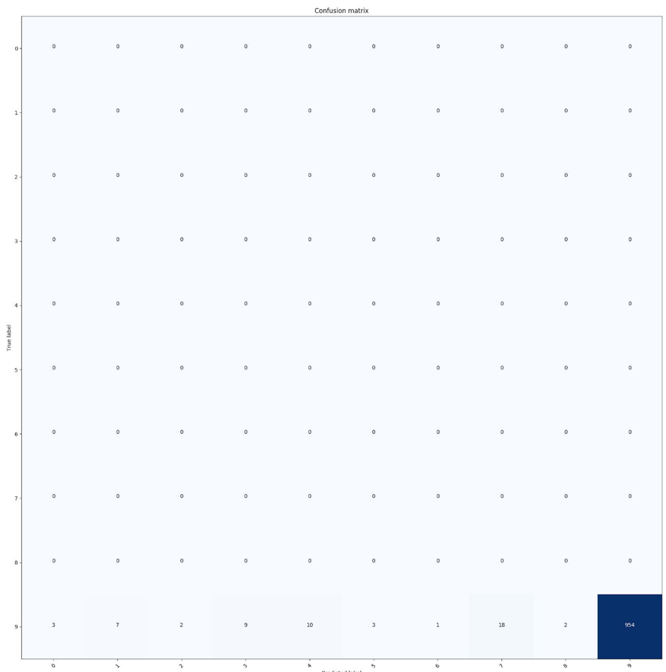

Fig: 4 Confusion matrix, pred 9 , p=1

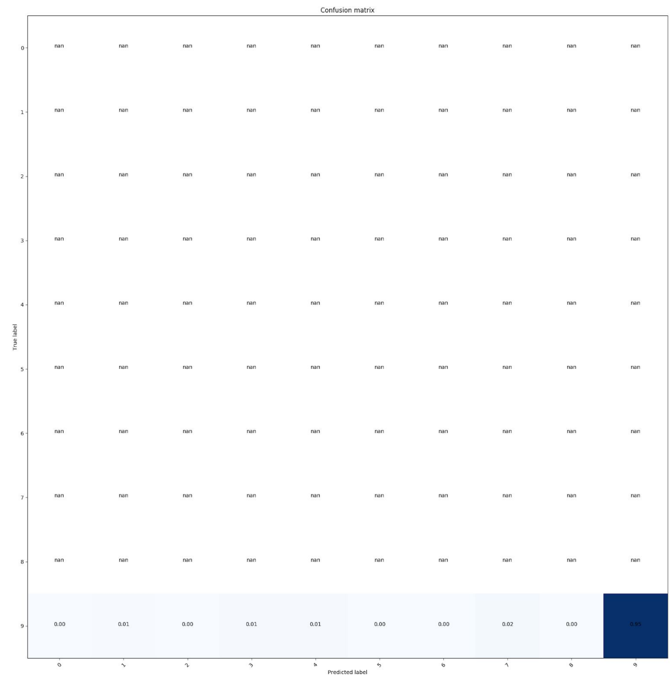

Fig 5: Normalized, classifying 9, P=1

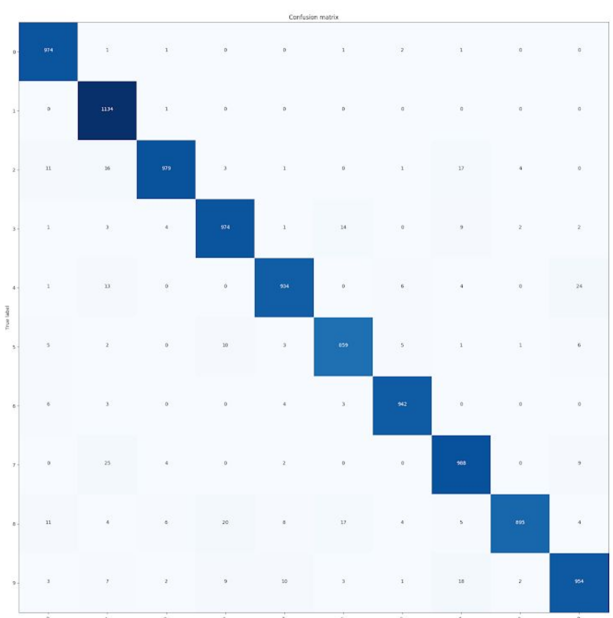

Fig 6: Classifying 0-9, P=1 Minkowski

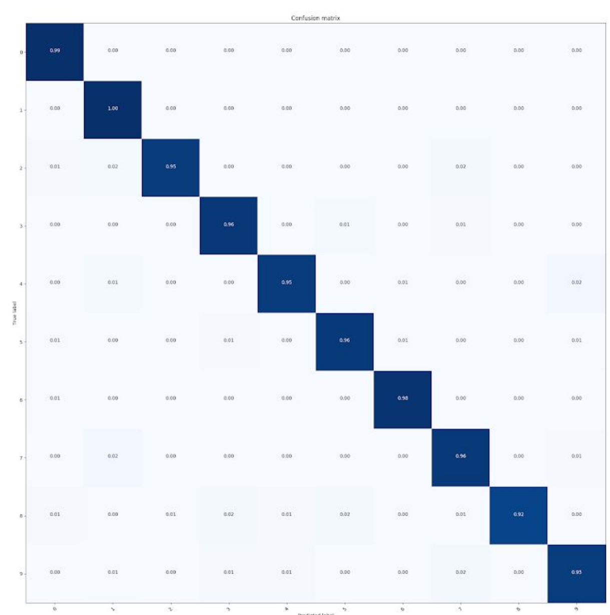

Fig 7: Classifying 0-9, P=1 Normalized

2. Decision Tree: Sklearn was used to run the decision tree classifier. The model was run using entropy and gini criterion for dept =8,16, default. Where the default is to run until the leaf is pure. Mean accuracy was measured using the building sklearn scoring function.

FIND ALL				**FIND NINE**		
Mean Accuracy	**Depth**	**Criterion**		**Accuracy**	**Depth**	**Criterion**
81.80%	8	gini		82.40%	8	gini
88.20%	16	gini		87.00%	16	gini
87.60%	Until pure	gini		84.00%	Until pure	gini
Mean Accuracy	**Depth**	**Criterion**		**Accuracy**	**Depth**	**Criterion**
83.60%	8	entropy		80.30%	8	entropy
88.90%	16	entropy		86.20%	16	entropy
88.50%	Until pure	entropy		86.90%	Until pure	entropy

Table 3: Decision tree final results

```python
from sklearn.tree import DecisionTreeClassifier, DecisionTreeRegressor
import struct
import itertools
import numpy as np
from sklearn import neighbors, metrics
import matplotlib.pyplot as plt
from sklearn.tree import export_graphviz
from sklearn.externals.six import StringIO
from IPython.display import Image
import pydotplus


#This code converts the MNIST idx format into numpy arrays
def read_idx(filename):
    with open(filename, 'rb') as f:
        zero, data_type, dims = struct.unpack('>HBB', f.read(4))
        shape = tuple(struct.unpack('>I', f.read(4))[0] for d in range(dims))
        return np.fromstring(f.read(), dtype=np.uint8).reshape(shape)

#import the train and test data and labels. Using all 60,000 images for training and 10,000 images for testing
raw_train = read_idx("train-images-idx3-ubyte")
train_data = np.reshape(raw_train, (60000, 28*28))
train_label = read_idx("train-labels-idx1-ubyte")

raw_test = read_idx("t10k-images-idx3-ubyte")
test_data = np.reshape(raw_test, (10000, 28*28))
test_label = read_idx("t10k-labels-idx1-ubyte")
#selecting all the data, digits 0-9 for training
idx = (train_label ==0) | (train_label == 1) | (train_label == 2) | (train_label == 3) | (train_label == 4) | (train_lab
x = train_data[idx]
y = train_label[idx]


#creating the decision tree classifier and setting paramters, either gini or entropy criterion. As well as depth. Defaul
DT = DecisionTreeClassifier(criterion='entropy', max_depth= 8).fit(x, y)

#Setting the test Label
idx = (test_label == 9)
x_test = test_data[idx]
y_true = test_label[idx]
y_pred = DT.predict(x_test)

import itertools

#Create confusion matrix
def plot_confusion_matrix(cm, classes,
                          normalize=False,
                          title='Confusion matrix Decision Tree with entropy criterion max depth = 8',
                          cmap=plt.cm.Blues):
    """
    This function prints and plots the confusion matrix.
    Normalization can be applied by setting `normalize=True`.
    """
    if normalize:
        cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]
        print("Normalized confusion matrix")
```

Figure 8: Decision tree Sklearn model

```python
    else:
        print('Confusion matrix, without normalization')

    print(cm)

    plt.imshow(cm, interpolation='nearest', cmap=cmap)
    plt.title(title)
    plt.colorbar()
    tick_marks = np.arange(len(classes))
    plt.xticks(tick_marks, classes, rotation=45)
    plt.yticks(tick_marks, classes)

    fmt = '.2f' if normalize else 'd'
    thresh = cm.max() / 2.
    for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
        plt.text(j, i, format(cm[i, j], fmt),
                 horizontalalignment="center",
                 color="white" if cm[i, j] > thresh else "black")

    plt.tight_layout()
    plt.ylabel('True label')
    plt.xlabel('Predicted label')

#create decision tree image
dot_data = StringIO()
export_graphviz(DT, out_file=dot_data,
                filled=True, rounded=True,
                special_characters=True,class_names=['0','1', '2', '3', '4', '5', '6', '7', '8', '9'])
graph = pydotplus.graph_from_dot_data(dot_data.getvalue())
graph.write_png('tree.png')
Image(graph.create_png())


#using thee sklearn score function, we find the mean accuracy of the decision tree
score = DecisionTreeClassifier.score(DT, x_test, y_true)
scorestr = str(score)
print("The mean accuracy score using entropy criterion and max depth = 8 is: " + scorestr)

param = DecisionTreeClassifier.get_params(DT, deep= True)

print(param)

#plots confusion matrix non-normalized
cm = metrics.confusion_matrix(y_true, y_pred)
plot_confusion_matrix(cm, ["0", "1", "2", "3", "4", "5", "6", "7", "8", "9"])
plt.show()

#plots confusion matrix normalized
plot_confusion_matrix(cm, ["0", "1", "2", "3", "4", "5", "6", "7", "8", "9"], normalize=True)
plt.show()
```

Figure 9 : Decision Tree Sklearn continued

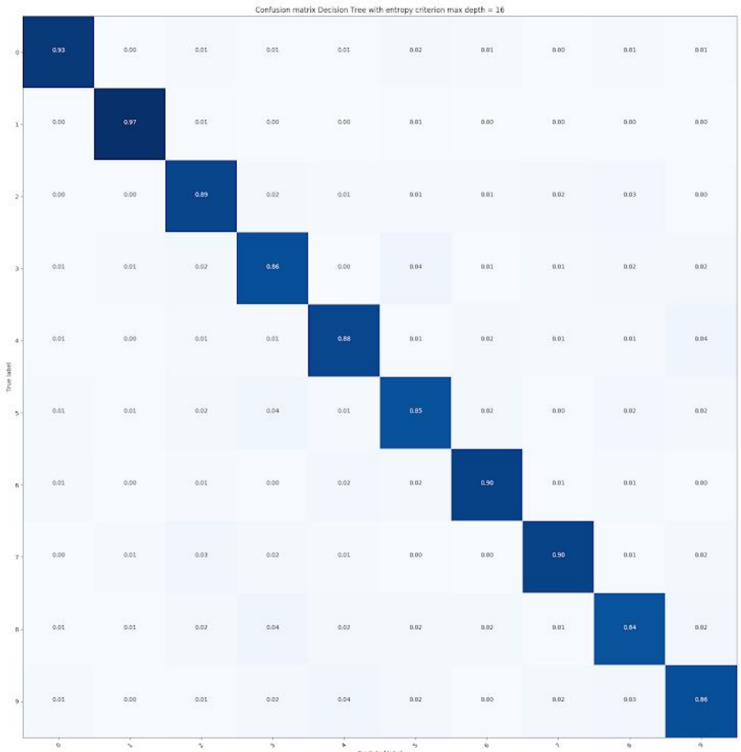

Fig10: Normalized Confusion matrix: Decision Tree, Max depth = 16, Entropy, Digits 0-9

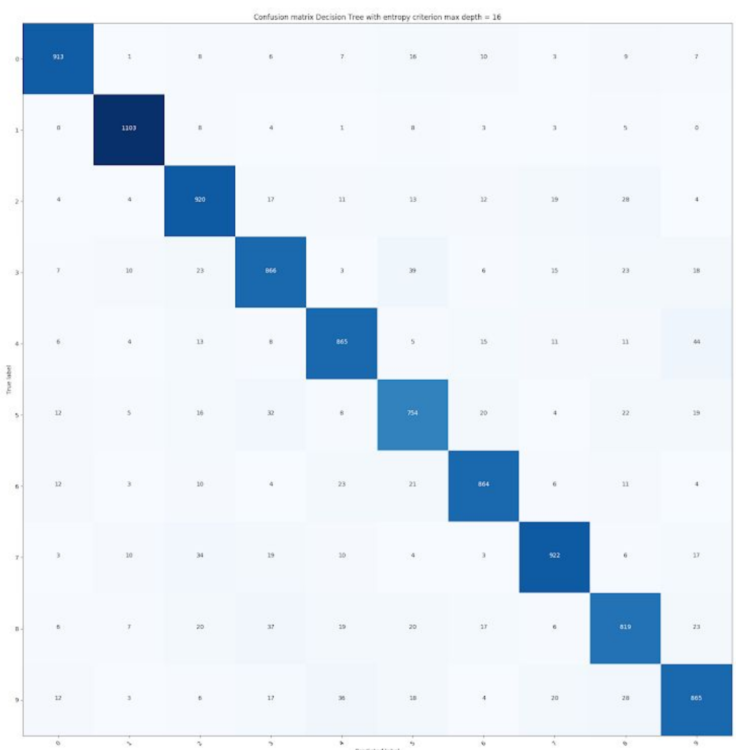

Fig10: Confusion matrix: Decision Tree, Max depth = 16, Entropy, Digits 0-9

3. **Random Forest**

To run a random forest model, i was able to reuse all of my previous code. As shown in Figure 11, I simply had to import the Random Forest Sklearn module and change the classifier. Parameters are similar to Decision Trees.

FIND ALL				**FIND NINE**		
Mean Accuracy	**Depth**	**Criterion**		**Mean Accuracy**	**Depth**	**Criterion**
90.70%	8	gini		89.90%	8	gini
94.68%	16	gini		92.60%	16	gini
94.32%	Until pure	gini		92.66%	Until pure	gini
Mean Accuracy	**Depth**	**Criterion**		**Mean Accuracy**	**Depth**	**Criterion**
90.45%	8	entropy		88.20%	8	entropy
94.84%	16	entropy		92.16%	16	entropy
95.02%	Until pure	entropy		92.17%	Until pure	entropy

Table 3: Random Forest model runs final results

```
#creating the RANDOM FOREST classifier and setting paramters, either gini or entropy criterion. As well as dep
# x, y = make_classification(n_features=784)
# DT = RandomForestClassifier(criterion='gini', n_jobs=-1).fit(x, y)
DT = RandomForestClassifier(criterion='entropy', max_depth = None, n_jobs=-1)
RndTree = DT.fit(x,y)
print (RndTree)
```

Fig 11: Code change to use random forest instead of decision tree, with Sklearn.

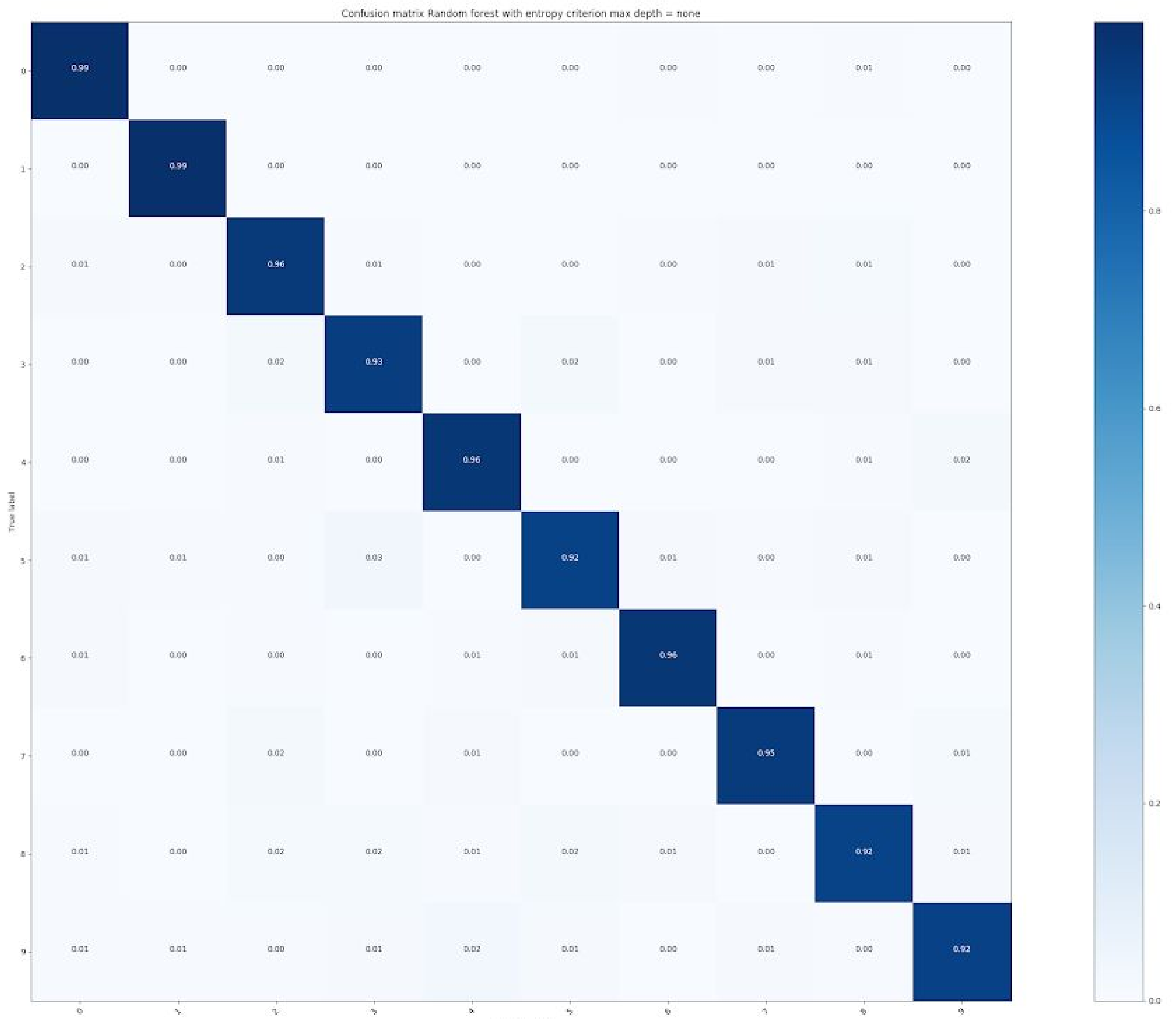

Fig 12: Normalized confusion matrix, random forest, Entropy , Max depth= None.

Final Results: Finding all digits 0-9

KNN- Mean Accuracy **97.3%** - K neighbors = 3, Minkowski P=3

Decision Tree- Mean Accuracy **88.9%** -Max Depth = 16, Entropy

Random Forest- Mean Accuracy **95.0%** - Max Depth = None, Entropy

Final Results: Finding just the 9s

KNN- Mean Accuracy **96.8%** - K neighbors = 3, Minkowski P=3

Decision Tree- Mean Accuracy **87.0%** -Max Depth = 16, Gini

Random Forest- Mean Accuracy **92.66%** - Max Depth = None, Gini

The Multiclass classification was more accurate in all three models, when compared to the binary classification. This is probably due to the fact that their is more data available to fit the clusters, nodes, and leafs.

It is also shown in the confusion matrices, that the digits 0 and 1, have a higher classification accuracy.