CS484-C01
Group 8
Barnabas Vizy
Zhiwen Luo

Kaggle - Digit Recognizer

For this assignment, we have chosen Python as our language, and Scikit/Scipy as our Library. We must decipher and pattern match which numbers these pixels form into.
Website: https://www.kaggle.com/c/digit-recognizer

All the codes and data sets: https://github.com/Vizzyy/digitrecognizer

We experimented with a lot of different classification methods within the library, namely :

- The lowest score is the decision tree(decision tree)
- Second Support Vector Machine (machine)
- Best was Randomforest.

This is the SVM.SVC support vector classification code, all the way through preprocess train, fit, predict.

Code:

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn import svm
import matplotlib as mpl
mpl.use('TkAgg')

import matplotlib.pyplot as plt
import matplotlib.image as mpimg
print("Successful import.")

labeled_images = pd.read_csv('/Users/barnabasvizy/Desktop/Kaggle/input/train.csv')
images = labeled_images.iloc[0:5000,1:]
labels = labeled_images.iloc[0:5000,:1]
train_images, test_images,train_labels, test_labels = train_test_split(images, labels, train_size=0.8, random_state=0)
print("Training Data loaded.")
i=1

test_images /=255
train_images /=255

img=train_images.iloc[i].as_matrix().reshape((28,28))
plt.imshow(img,cmap='binary')

clf = svm.SVC()
clf.fit(train_images, train_labels.values.ravel())

print(clf.score(test_images,test_labels))

test_data=pd.read_csv('/Users/barnabasvizy/Desktop/Kaggle/input/test.csv')
test_data[test_data>0]=1

results=clf.predict(test_data[0:28000])

print(results)
print("Printing out to results file.")

df = pd.DataFrame(results)
df.index.name='ImageId'
df.index+=1
df.columns=['Label']
df.to_csv('/Users/barnabasvizy/Desktop/Kaggle/input/results.csv', header=True)
```

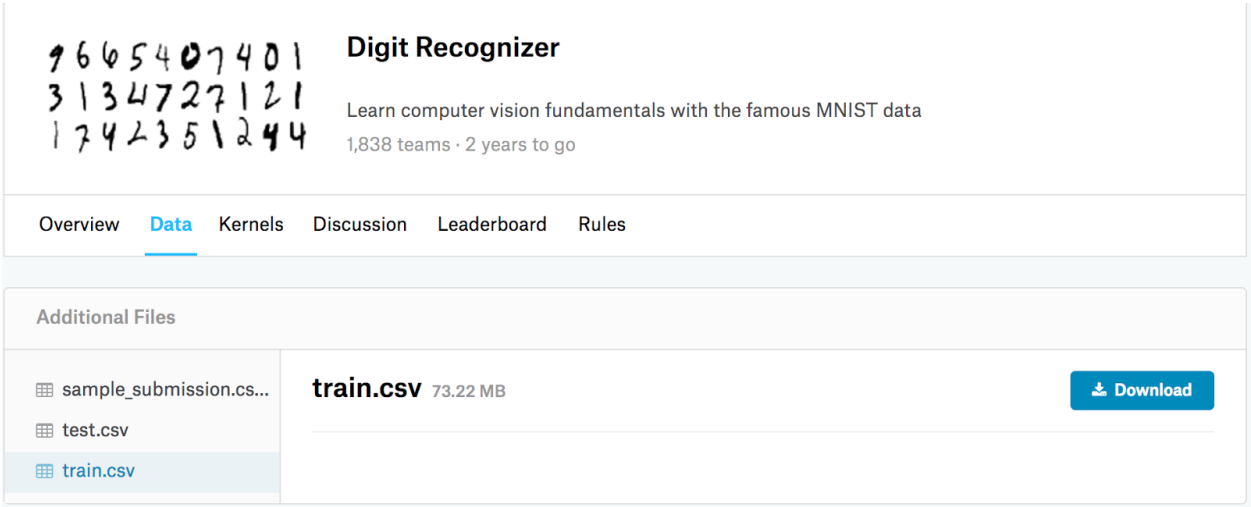
Data collection and sampling according to data distribution

- Sub sample of 5000 images to train with.
- The data set is a matrix of pixels
- Each pixel is in gray scale, so it has a value or 0-255
- 784 pixels per record

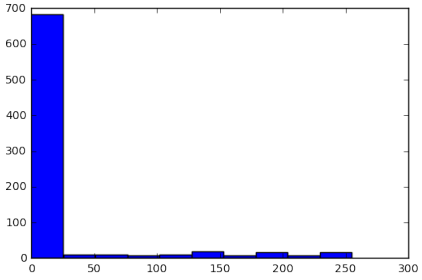
The data set is provided on Kaggle, as the csv file, and labeled as “test” and “train” sample sets.

The full testing set is 28000 records.

Website: https://www.kaggle.com/c/digit-recognizer/data



Histogram of Pixel Values:



Preprocessing (data integration and data normalization)

Using all of the values of the pixels offers too much resolution

- Take grayscale and convert to black and white, to reduce variability
- Resize data into 28,28 NxN matrix rather than 1x738

- visualization

We can now quickly sample a training set while holding out 40% of the data for testing

```
train_images, test_images,train_labels, test_labels = train_test_split(images, labels, train_size=0.4)
```

(evaluating) our classifier, as shown here with train_size = .4

- feature selection using some ranking criteria (e.g., Fisher linear discriminant ratio)

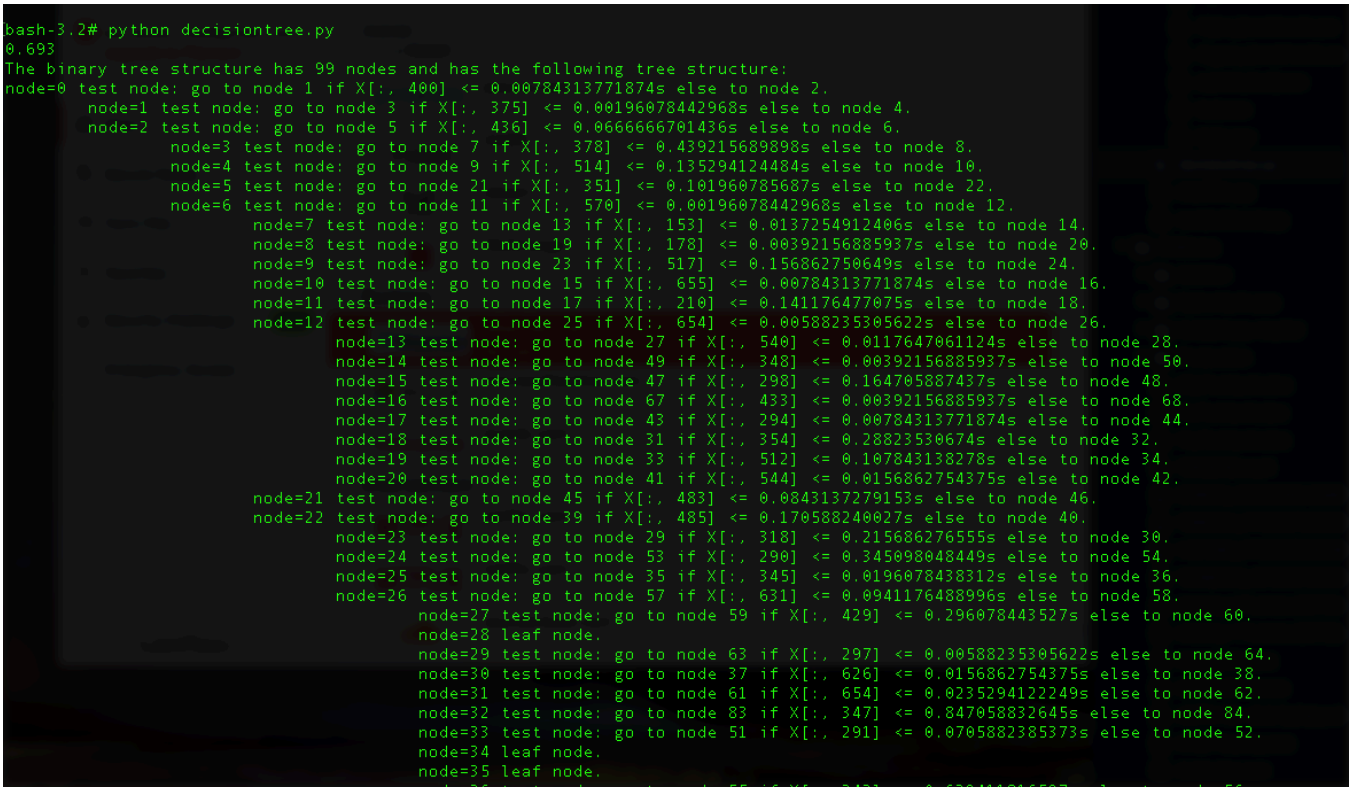
Selecting the features of this data set is tricky because of each dimension, and there are 784 per record, represents a single pixel in the digit image. We cannot really select which is most important because the whole image can be used to represent the symbol

Example Code:

```
import pandas as pd
from sklearn.model_selection import train_test_split, cross_val_score, ShuffleSplit
from sklearn import svm
import matplotlib as mpl
mpl.use('TkAgg')

import matplotlib.pyplot as plt
import matplotlib.image as mpimg
#print("Successful import.")

labeled_images = pd.read_csv('kaggle/input/train.csv')
images = labeled_images.iloc[0:10000,1:] #data to test (X)
labels = labeled_images.iloc[0:10000,:1] #answers (y)
train_images, test_images,train_labels, test_labels = train_test_split(images, labels, train_size=0.4, rand
print("Training Data loaded.")
i=1

test_images /=255
train_images /=255
#train images binary now
img=train_images.iloc[i].as_matrix().reshape((28,28))
plt.imshow(img,cmap='binary')

clf = svm.SVC()
clf = clf.fit(train_images, train_labels.values.ravel())

print(clf.score(test_images,test_labels))

scores = cross_val_score(clf, test_images, test_labels.values.ravel(), cv=5)
print(scores)
print("Accuracy: %0.2f (+/- %0.2f)" % (scores.mean(), scores.std() * 2))

cv = ShuffleSplit(n_splits=5, test_size=0.4, random_state=0)
from sklearn import preprocessing
from sklearn.pipeline import make_pipeline
clf2 = make_pipeline(preprocessing.StandardScaler(), clf)
scores = cross_val_score(clf2, images, labels.values.ravel(), cv=cv)
print(scores)
print("Accuracy: %0.2f (+/- %0.2f)" % (scores.mean(), scores.std() * 2))
```

- visualization

Example decision tree printout for literal tree. Note that it is very hard to interpret a tree like this, with this kind of data set, because each feature is just a pixel.



- dimensionality reduction, if any

We can reduce dimensions, but we also simultaneously risk reducing the resolution of the image. However, the reduction of the data from 255 different values, to binary, is a massive improvement in overall reduction and should be emphasized.

There are many methods of principal component analysis but they were above our understanding of implementation in the python library.

- 5-fold cross validation using different versions for decision trees, where the versions differ in terms of parameters used.

Here is the cross_val snippet :

```
scores = cross_val_score(clf, test_images, test_labels.values.ravel(), cv=5)
print(scores)
print("Accuracy: %0.2f (+/- %0.2f)" % (scores.mean(), scores.std() * 2))
```

- And the result for Vector Classification, using 5-fold cross Validation: 91% , which is what Kaggle gave us for this (scoring against 28000 test samples)

```
[ 0.9045  0.9025  0.9055  0.9105  0.91  ]
Accuracy: 0.91 (+/- 0.01)
```

- And the Decisiontree implementation:

```
[bash-3.2# python decisiontree.py
0.7685
[ 0.78225  0.7605   0.77075  0.771    0.76875]
Accuracy: 0.77 (+/- 0.01)
```

- Randomforest(our winner), also same as kaggle scoring

```
[bash-3.2# python randomforest.py
0.944
[ 0.94425  0.938    0.94275  0.9415   0.9415 ]
Accuracy: 0.94 (+/- 0.00)
```

Results using confusion matrix and metrics such as F-measure, Precision, Recall

These results are for our Randomforest implementation
Confusion matrix: (target = initial labels = y)

```
print(confusion_matrix(target, predicted_digits))
```

```
[[270 308 256 279 253 219 277 307 270 282]
 [301 371 315 306 291 269 305 303 338 308]
 [290 321 294 273 279 242 285 281 296 263]
 [282 327 286 303 298 240 276 296 301 300]
 [270 316 285 275 282 235 247 271 290 285]
 [274 263 252 227 233 237 255 249 268 263]
 [289 348 291 279 282 235 268 262 271 254]
 [268 335 295 287 301 279 336 277 265 281]
 [290 308 264 272 239 214 265 320 257 265]
 [282 324 289 269 261 258 262 284 256 280]]
```

Code:

```
print("precision_score: ",precision_score(target, predicted_digits, average=None))
print("recall_score: ",recall_score(target, predicted_digits, average=None))
```

These results indicate the individual precision and recall for every label [0-9], 10 total.

```
precision_score:  [ 0.0957827   0.11612702  0.10660078  0.10768126  0.1052826   0.09691992
  0.09544313  0.09968299  0.09329549  0.10344828]
recall_score:  [ 0.0984932   0.1200515   0.10694051  0.10312822  0.10341074  0.09361365
  0.09571788  0.09678523  0.09762435  0.10415913]
```

Discussion of results

- Tree classification Worst
- Support Vector Classification Second
- RandomForrest Best

The binarization of the input pixels has a fantastic improvement on score, furthermore increasing testing size also increases score upto a threshold of roughly 10k samples.

This doesn’t necessarily mean these solutions are the best for this model, obviously there are many other better solutions present on kaggle alone, but for our instance of the data set and our python knowledge this is what we were able to achieve.

Conclusions

- Randomforest implementation worked best for us
- Our implementation was quite rudimentary
- Neural Network Required for highest scores
- Our results from Kaggle improved with every iteration of our code. Trying different implementations with slightly different parameters for their respective algorithms allowed us to slowly increment our success score. Despite it being .941 score, this is not by any means profound, resting comfortably in the 85% percentile of submissions.