Artificial Intelligence
Computer Vision System (May 2016)

Brandon Hardy, *University of Surrey*

***Abstract* – This research looks at the development of a Computer Vision System and its results of accuracy in relations to its given testing data. The reports aim is to give insightful findings of the system and its accuracy of successfully extracting, classifying and detecting text and words. The methods involved prior research to the problems, Data Preparation, Feature Extraction and Classification. We show that results vary depending on the classification methods and training data given to the system. Recommendations to the current Computer Vision System are also introduced.**

***Index Terms* – AI Artificial Intelligence, SVM Support Vector Machine, KNN K-Nearest Neighbour, HOG Histograms of Orientated Gradients.**

I. Introduction

THIS document sets out a report on findings and personal understandings of Character Recognition. This report will show how by training a letter recogniser using training datasets, it is able to detect characters and feedback accuracy based of testing datasets, such as printed pages, handwritten notes, and signatures. To enable the letter recogniser to learn how to recognise characters, classifiers are used, such as KNN, SVM, these classifiers give a basis of comparison, by the percentage of accuracy that is returned by the program. This document will present thorough analysis of the problem, along with research and challenges faced. This document will display results and findings of the problem, discussing the data used along with the accuracy of the data. Furthermore, discussions of the findings will be included to aid demonstrate understanding. The objective of the document is to demonstrate results and understandings of each problem, giving insightful reasons to why each classifier was used with each problem – but mainly demonstrating that machines can learn how to detect text in documents.

II. Relevant Topics

This section of the report is simply to demonstrate a relevant topic to the Computer Vision System in recognising text. http://myscript.com is a leading company that specialises in handwriting recognition software. With the use of Artificial Intelligence, they have created a software that recognises complex Mathematical Equations, Geometric Shapes, and Music Notation "Ref. 1". This relates to the work being done laid out in this report, due to the fact that the system that is being created detects handwriting, printed text, and signatures.

Paper submitted for review 09/05/2016. Author Brandon Hardy (BH00125@surrey.ac.uk)

III. The Problems & Evaluations

This section of the report will lay out the various problems and comment on the research, technical issues and challenges, furthermore describing the approach of development to achieve results and determining accuracy between classifiers.

A. Recognising Printed Pages

The first problem of the project was to extract letters and words from an image of a printed page. The goal of this is to take the image of the printed page and output a text of words that are in the image. The problem required you to build a dataset that is suited to training the program to ensure that the program could detect the various letters and words in the image of the printed page. To do this, creating various documents with letters A-Z in lowercase, A-Z in uppercase and the numbers 1-9 with varying fonts and sizes remained an appropriate way to train the program to detect words and text in the printed pages "See Fig 13.". This ensured that when the program was trained, it was set to detect all different types of font and varying shapes of the text. This increased the accuracy of the program. If given less training data, the program has further chance to not detect certain letters, or mistakenly detect letters as other letters, resulting in lower accuracy to the truth of the document. Once the training data had been created and there were various documents to allow the program to learn the differing types of letters from A-Z, the next problem was to implement a classifier to build a program to identify the text and words in an image. Once the classifier had been created, testing that the system can correctly identify words in the document followed. There were two testing documents. "Shazam.png" and "Adobe.png" (See "Fig 11. & 12"). These documents had different fonts; this was to help compare the differences in the classifiers to help differentiate the accuracy results that were printed.

The first classifier that was used to build the system was Support Vector Machine (SVM). SVM takes some data to start with that's already classified (the training set), and tries to predict a set of unclassified data (the testing set). As words and text has many different features, the SVM plots each data item as a point in space, with each value of each feature (letter) being the value of a coordinate. The objective is to find some line that splits the data between classified groups of data; best as possible. This will be the line such that the distances from the closest point in each of the two groups will be farthest away. Once the line has been retrieved, that is our

classifier. Depending on where the test data plots on either side of the line that's the class we can classify the new data as.

The second classifier that was used to build the system was K-Nearest Neighbour (KNN). KNN is fairly simple – yet effective. The KNN will search through a given dataset (training data) for the k-most similar instances. In most cases, Euclidean Distance can be used for this. For example, take x to be the point labelled, the point closest to x let it be y. Once the nearest neighbour is found (y), if the number of 'data points' is large then there is a high chance that x and y are the same.

Research that was undertaken before attempting the problem of recognising text on printed pages involved creating a simple digit recognition system. This used KNN to help identify the closest matches in the dataset; which is very similar to what is happening in this problem of recognising text on a printed page.

Problems that arose when recognising printed pages were as follows. The classifiers that were used were never fully accurate. Regardless of the training data that was used, the data was never fully accurate. The reason being was that when extracting the letters there were a few letters that did not get recognised correctly, and the letters were put into a dataset that had incorrect letters. The main letters that were not extracted correctly were I's. The reason being was that the lower case 'i's' appear separated due to the dot. Therefore, when reading in the printed pages, there were a few issues with not picking up I's correctly. The I's that were picked up were of those that were uppercase. The program could not detect the fact that the dot above the I's line goes together and therefore an I. The training therefore would have dots and lines and not explicit I's. As a result, the program would add a few letters onto the result due to picking up some I's as a dot and a line, meaning when there was an I in the text, instead of counting it as a single letter, it would count as two; therefore, lowering the accuracy. As shown in "Fig 9.", the accuracy is thrown off because of the miss read I at the beginning of the file. Once it is thrown off at the start the program is then thrown off by a character for the rest of the time. Although the truth is very similar to the actual result as shown in "Fig 9." because the program is thrown off by a character, the accuracy is very low.

A further issue to the printed pages were that the program did not allow for brackets to be cancelled out. The program, was not trained to allow for brackets, and therefore caused a problem for the system, and could lower the accuracy results. To counter this problem including the brackets into the truth String seemed to have fixed the issue, and the accuracy was upped. However, the program could in some cases could pick these up as I's or L's as they are similar in certain fonts.

In evaluation to the 'detecting printed pages' task, there were varying results from the dataset, with the classifiers used such as SVM & KNN producing varying results. These are as follows. When testing the classifiers with the computer vision system, the program would print the accuracy of each classifier, giving a view on how well the classifier performed against the truth string of the document, the results were interesting and there were a few problems that arose when testing the accuracy, these are mentioned below, however, for now the result were of follows. The accuracy with the testing data "Shazam.png" using the SVM classifier gave a fairly accurate result of 75%. As shown in "Fig 1.". This is fairly high, indicating a high level of accuracy. However, there is a further 25% that remained incorrect. This may be due to the fact, as mentioned above the I's were not correctly extracted – however SVM does not react well to large volumes of training data, and may play a factor in the lower result. Using the KNN classifier with the testing data gave a very high accuracy result of 95%. As shown in "Fig 2.". This, as I understand may be down to the fact that the training data was large, with many variations of letters – which KNN reacts to very well. The KNN classifier seemed to pick up the I's in a much more consistent manor, compared to SVM.

The results for the second testing dataset 'Adobe.png' produced exceptionally poor results. As shown in "Fig 9." The results of this were 9% accuracy for SVM, the KNN result was slightly higher at 11% as shown in "Fig 10." & "Table 1.", however as mentioned above in the problems and challenges, the reason for this is that the program detected an early I in the document, for an unknown reason, and therefore put the document one string ahead, as a result throwing it off of the truth string. When looking at the returned string it is accurate and would normally produce a high percentage of accuracy, however due to the program being a character ahead and detecting I's incorrectly, the accuracy was reduced considerably. In order to improve the program within the future, the recommendation would be to ensure that the bounding boxes account for I's and L's correctly. The inaccurate reading of I's and L's reduced the accuracy considerably and therefore would recommend finding a further way to increase bounding box accuracy.

B. Handwritten Notes

The second task was to create an engine that recognised handwritten notes that I personally wrote. The handwritten data was written onto a A4 Plain white sheet of paper using a black pen. The handwriting was separated and non-continuous, if the handwriting was continuous, then it would be difficult for the system to segment the words and letters. The pen was also black to ensure that when scanning in the document there will be no loss in the shape of a letter.

The training data was a page for each letter from A-Z (26 total pages), each letter was written 25 times on the paper. This meant that the system was trained on each letter a fair amount of times and therefore presumed an increase in the accuracy of the results from the large amount of testing data. The testing data was also handwritten in black pen, with non-continuous writing. This was then tested against the handwritten data. The methods used were similar to the printed pages, meaning that there was not much to change. This was because the training and testing data was simply being changed.

There was still a use of SVM and KNN classifiers to train the data efficiently, and both provided different results in terms of accuracy. A problem with using the SVM classifier with the handwritten training data was the SVM does not react well to large amounts of data, and therefore using SVM caused some performance issues, but nothing major. There was no use for HOG in this problem, but there was a use of Ravel instead. Ravel, part of SciPy numpy, returns a contiguous flattened array. The use of this was instead of HOG due to the fact that HOG is not good for small images, unless you change the parameters.

There were various issues with Handwritten notes, this was mainly due to the fact that handwriting varies and is easily prone to human error. Letters may vary large amounts, which may not appear fully to humans, but within the system, if there was a single a with too much of a long tail, it may pick it up as a Q. This caused issues with the system, and in some cases, letters were recognised incorrectly, due to length of tails, or shape of the letter.

The major problem with Handwritten was I's. The system was unable, most of the time to extract I's correctly, and therefore gave inaccurate data when learning 'I' characters. The system would recognise the straight line, but there would be random pieces of other letters in the folders also, and therefore, 'I's' were not implemented properly.

When implementing training data for the handwritten task, the training data had to be in black / dark pen, and what was found that using a thicker pen, such as a sharpie may increase the chance of the system picking up different letters and separating words. Therefore, if you were to use a thin pen, with light coloured ink there may be some inaccuracy in the training data extraction.

In evaluation to the 'Handwritten Notes' task and its Data results, the results that were produced were disappointing, and did not represent accurate reading of the training data and testing data. "Fig 3." Shows the results for the Handwritten problem using the SVM Classifier. The accuracy result shown is very low at 15% "See Table 2.", getting only 5 characters out of 33 (truth string) correct. SVM does not react well to single letter training data, and therefore may explain a lower result. However, the main fact that the program is detecting very low amounts of letters is down to the fact that handwriting is very inconsistent. The letters and words can be written in a poor manor that may not look like the training data. Handwritten text is prone to human error, and therefore makes it harder for the program to detect letters. "Fig 4." Shows the KNN accuracy have a slightly increased result of 24% "See Table 2.". Picking up 8 out of 33 letters. This may be because of KNN reacting well to single letter datasets, compared to SVM. The results are still very low. This is mainly because human error, and handwriting may differ large amounts. What would have improved the handwriting accuracy may have been using a sharpie to help the program separate the words effectively. The problem with the handwriting task, was also that the program was picking up bounding boxes within bounding boxes; which causes a problem with adding letters to the dataset.

C. Signature Recognition

The final task of the computer vision system was to make the program recognise given signatures. As part of training the system, we will use a dataset of signatures from two different people. As seen in "Fig 5 & 6." there are 12 variations of signatures for each person. These signatures were done with pen, and scanned into the computer. The signatures, unlike handwritten text recognition do not require to extract letters, but simply use a system to detect the differences in the signatures, and learn what differs the given signature to the other.

Research that was carried out before undertaking this part of the problem for the computer vision system was to identify buildings and highways in pictures. Research involved creating a program that detected if there were buildings in a picture, with varying backgrounds and scenarios. This was related heavily to the signature recognition task, as the same features of the program for the buildings and highways could be implemented into the program to help detect the signatures. The research implemented HOG and SVM, which leads onto the description of development.

The signature recognition uses HOG to help identify which signature is which. HOG is a type of 'feature descriptor'. Intentions of a feature descriptor is to generalise an object in a way which the same object produces a clone as the same feature descriptor when viewed under different conditions. The HOG feature uses a sliding detection window that is moved around the image. At each position of the window a HOG descriptor is calculated for the detection window. The descriptor is then shown to the SVM which then classifies it as either person A or person B. To fill the training dataset, the instruction is to load the sample, e.g. the signature to detect, resize it to 200x200 so that all images are the same size, it then gets the HOG features, saves them to the data list that holds the HOG features and saved the label of that sample in the labels list. To compute the HOG descriptor, we operated on using 16x16 pixel cells within the detection window. The use of HOG and SVM for the signature recognition is the fact that it is the most popular and successful detectors. The success is fully reflected in the results of the signature recognition in the problem.

In terms of problems with the signature recognition, there were no real problems, the intention to teach the program the different signatures and variations was straight forward and when implementing it, the program ran as intended and correctly showed the intended results after testing with eight different testing data objects. "See Table 3".

The signature recognition system, as mentioned above had two sets of data. One being person A's signature, in this case, Brandon. Then person B's signature, written out 12 times for each. One signature was taken from each set to test the dataset upon. The program code was then edited to change the testing

file name to the signature taken from each set. The results were accurate every time, and successfully recognised the different signatures. Shown in "Fig 7 & 8." the signature detector returns the correct name with every sample and therefore represents accurate detection of the signatures.

IV. Discussions & Highlighted Findings

In summary, the results that have been presented within this document have been varied, with some odd findings. In terms of the results that have stood out within the document are most certainly those of which the results are below 20%. Recognising text on a printed page has given an insight to the amount that results can be altered by one character being read incorrectly. With the first results from using "Shazam.png" compared to "Adobe.png". 95% to 11% (See "Fig 2. & 10.") is a large drop and reflects on the difference a single character can make on accuracy. However, as a result of the lower percentage of accuracy it is key to take note that the program can be improved considerably. The system can be improved, however in terms of accuracy it may show that the training data could also be improved to help better the accuracy of the results. SVM may not perform well to this, as it does not perform optimally with large amounts of data; however, KNN may perform considerably better, which has already been demonstrated above, with the 95% accuracy over 75% accuracy with SVM. (See "Fig 1. & 2." & Table 1.)

Results that also have to be highlighted are of those in the handwritten problem. The results were considerably poor. The fact may be down to that there was not enough training data to fully train the system to detect letters in the words. However, it may be argued that the text written on the scanned in paper could have been written in bolder text to help the system detect letters to train clearer.

V. Conclusion

"A breakthrough in machine learning would be worth ten Microsofts" (Bill Gates, Chairman, Microsoft). Machine Learning can allow systems be trained to detect text in documents with some training data. But, how accurate can this be? The document aimed to present findings and demonstrate understanding of simple text recognition using classifiers. The findings in this report can answer the question itself. Using simplistic code, with some training data can train a system to detect text in documents; this report demonstrates the possibility. In order to improve the detection in documents larger volumes of training data can be used, the main downfall in lower accuracy results in this report can be argued that it is down to the low amount of training data given to the machine. The larger the dataset of training, the more variations of font and handwritten text the system can detect. Within the findings of the report, the most effective classifier with demonstrated success was KNN. The classifier reacted well to large amounts of training data, providing higher accuracy results for the printed text problem, and handwritten text recognition.

References

[1] MyScript, [Online]. Available: http://myscript.com/about/our-story/

[2] HOG Person Detector. Available: http://mccormickml.com/2013/05/09/hog-person-detector-tutorial/

Appendix

Fig 1. Results for Shazam SVM.

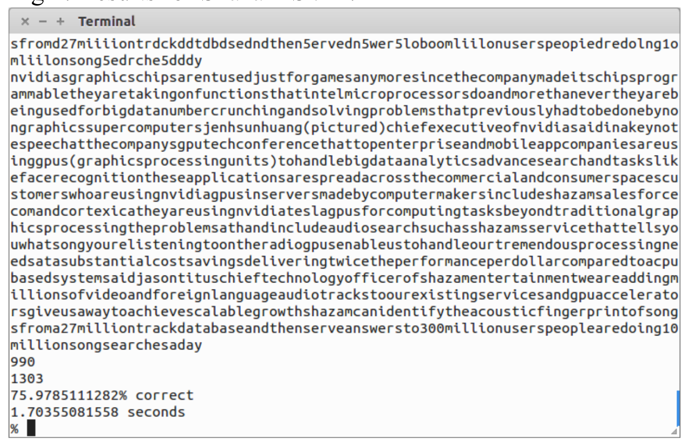


Fig 2. Results for Shazam KNN.

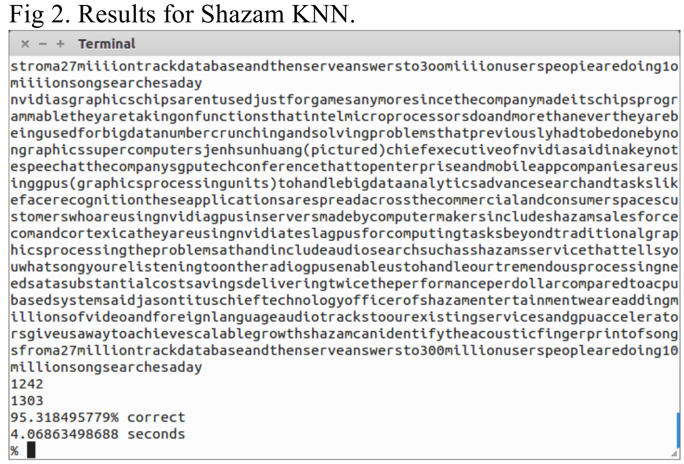


Fig 3. Results for Handwirtten SVM

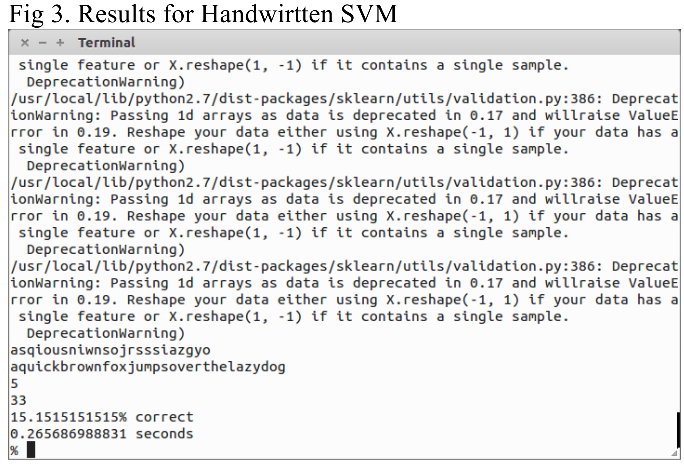


Fig 4. Results for Handwritten KNN

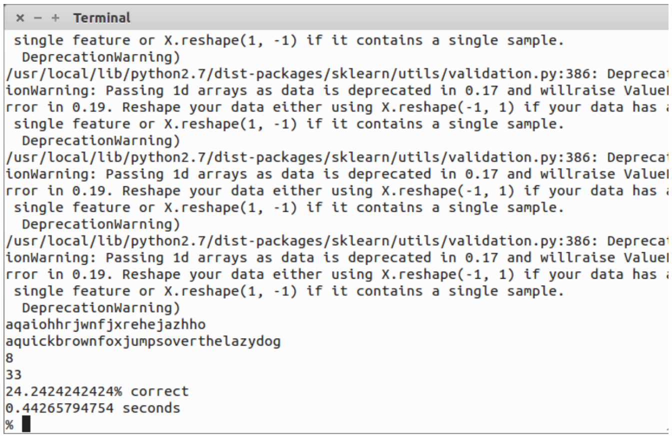


Fig 5. Training Data for Signature Recognition

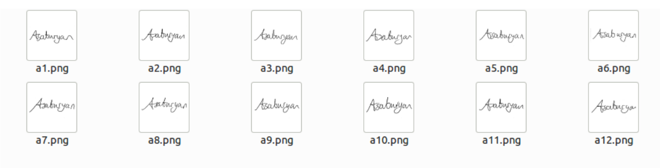


Fig 6. Training Data for Signature Recognition

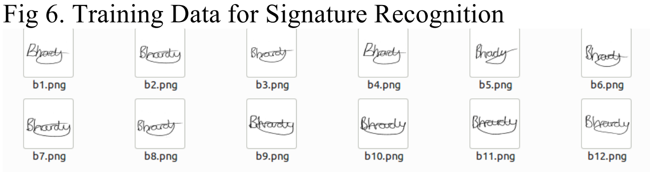


Fig 7. Andre Correct Result for Signature Recognition

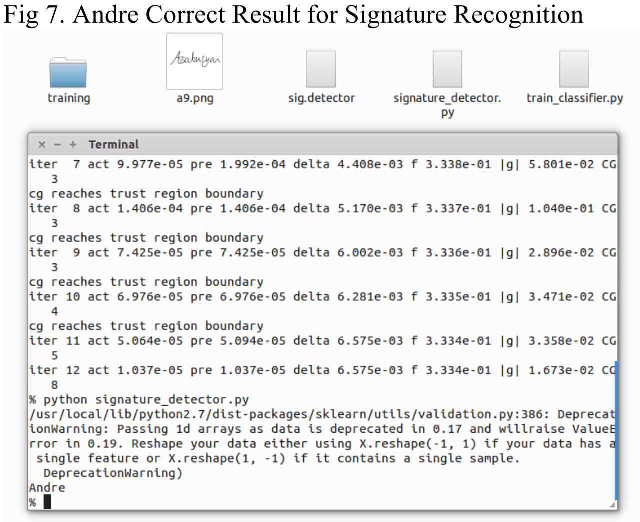


Fig 8. Brandon Correct Result for Signature Recognition

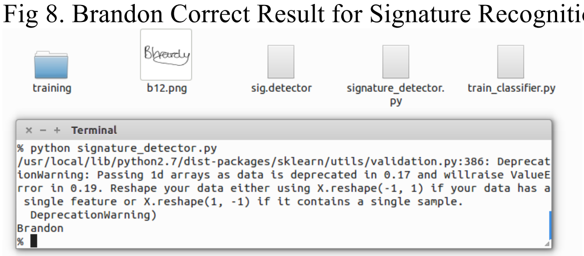


Fig 9. SVM Result for Adobe Printed Text

Fig 10. KNN Result for Adobe Printed Text

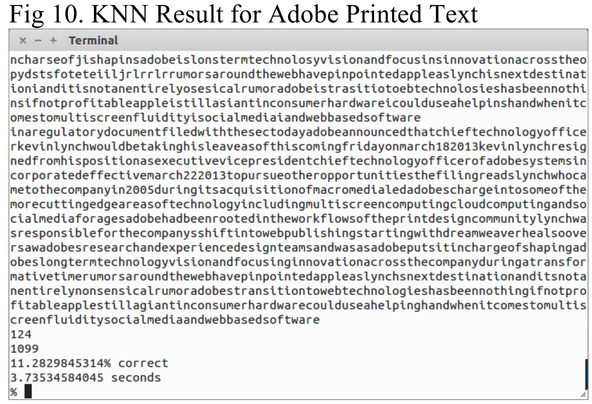
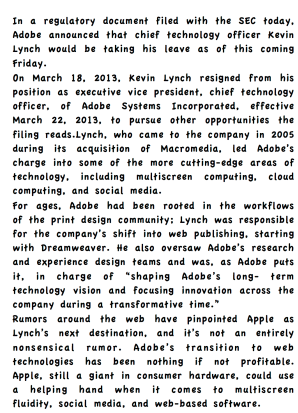


Fig 11. Original Adobe PNG Testing Data

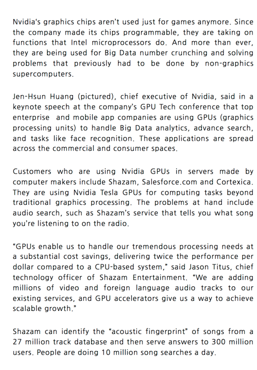

In a regulatory document filed with the SEC today, Adobe announced that chief technology officer Kevin Lynch would be taking his leave as of this coming Friday.

On March 18, 2013, Kevin Lynch resigned from his position as executive vice president, chief technology officer, of Adobe Systems Incorporated, effective March 22, 2013, to pursue other opportunities the filing reads.Lynch, who came to the company in 2005 during its acquisition of Macromedia, led Adobe's charge into some of the more cutting-edge areas of technology, including multiscreen computing, cloud computing, and social media.

For ages, Adobe had been rooted in the workflows of the print design community; Lynch was responsible for the company's shift into web publishing, starting with Dreamweaver. He also oversaw Adobe's research and experience design teams and was, as Adobe puts it, in charge of "shaping Adobe's long- term technology vision and focusing innovation across the company during a transformative time."

Rumors around the web have pinpointed Apple as Lynch's next destination, and it's not an entirely nonsensical rumor. Adobe's transition to web technologies has been nothing if not profitable. Apple, still a giant in consumer hardware, could use a helping hand when it comes to multiscreen fluidity, social media, and web-based software.

Fig 12. Original Shazam PNG Testing Data


Nvidia's graphics chips aren't used just for games anymore. Since the company made its chips programmable, they are taking on functions that Intel microprocessors do. And more than ever, they are being used for Big Data number crunching and solving problems that previously had to be done by non-graphics supercomputers.

Jen-Hsun Huang (pictured), chief executive of Nvidia, said in a keynote speech at the company's GPU Tech conference that top enterprise and mobile app companies are using GPUs (graphics processing units) to handle Big Data analytics, advance search, and tasks like face recognition. These applications are spread across the commercial and consumer spaces.

Customers who are using Nvidia GPUs in servers made by computer makers include Shazam, Salesforce.com and Cortexica. They are using Nvidia Tesla GPUs for computing tasks beyond traditional graphics processing. The problems at hand include audio search, such as Shazam's service that tells you what song you're listening to on the radio.

"GPUs enable us to handle our tremendous processing needs at a substantial cost savings, delivering twice the performance per dollar compared to a CPU-based system," said Jason Titus, chief technology officer of Shazam Entertainment. "We are adding millions of video and foreign language audio tracks to our existing services, and GPU accelerators give us a way to achieve scalable growth."

Shazam can identify the "acoustic fingerprint" of songs from a 27 million track database and then serve answers to 300 million users. People are doing 10 million song searches a day.

Fig 13. Training Data Example for Printed Pages.

Fig 14. Example Training Data for Handwritten.

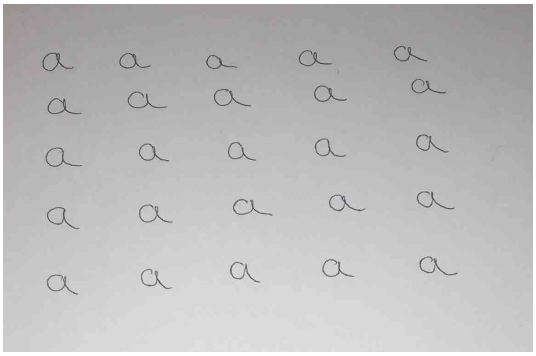


Table 1.

Printed Page	SVM Accuracy	KNN Accuracy
Shazam.png	76%	95.4%
Adobe.png	9.6%	11.3%

Table 2.

Handwritten	SVM Accuracy	KNN Accuracy
Brownfox.png	15.2%	24.2%

Table 3.

Signature	Result
A1.png	Correct
B1.png	Correct
A3.png	Correct
B3.png	Correct
A7.png	Correct
B8.png	Correct
A12.png	Correct
B11.png	Correct

Code Used

Recognising Printed Pages
Extract Letters.py

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.patches as mpatches
from scipy.misc import imread,imresize
from skimage.segmentation import clear_border
from skimage.morphology import label
from skimage.measure import regionprops

class Extract_Letters:
    def extractFile(self, filename):
        image = imread(filename,1)

        #apply threshold in order to make the image binary
        bw = image < 120

        # remove artifacts connected to image border
        cleared = bw.copy()
        #clear_border(cleared)

        # label image regions
        label_image = label(cleared,neighbors=8)
        borders = np.logical_xor(bw, cleared)
        label_image[borders] = -1


        fig = plt.figure()
        #ax = fig.add_subplot(131)
```

```
        #ax.imshow(bw, cmap='jet')

        letters = list()
        order = list()

        for region in regionprops(label_image, ['Area', 'BoundingBox']):
            minc, minr, maxc, maxr = region['BoundingBox']
            # skip small images
            if maxc - minc > len(image)/250: # better to use height rather than area.
                rect = mpatches.Rectangle((minc, minr), maxc - minc, maxr - minr,
                                          fill=False, edgecolor='red', linewidth=2)
                order.append(region['BoundingBox'])


        #sort the recognised characters left->right, top->bottom
        lines = list()
        first_in_line = ''
        counter = 0

        #worst case scenario there can be 1 character per line
        for x in range(len(order)):
            lines.append([])

        for character in order:
            if first_in_line == '':
                first_in_line = character
                lines[counter].append(character)
            elif abs(character[0] - first_in_line[0]) < (first_in_line[2] - first_in_line[0]):
                lines[counter].append(character)
            elif abs(character[0] - first_in_line[0]) > (first_in_line[2] - first_in_line[0]):
                first_in_line = character
                counter += 1
                lines[counter].append(character)


        for x in range(len(lines)):
            lines[x].sort(key=lambda tup: tup[1])

        final = list()
        prev_tr = 0
        prev_line_br = 0

        for i in range(len(lines)):
            for j in range(len(lines[i])):
                tl_2 = lines[i][j][1]
                bl_2 = lines[i][j][0]
                if tl_2 > prev_tr and bl_2 > prev_line_br:
                    tl,tr,bl,br = lines[i][j]
                    letter_raw = bw[tl:bl,tr:br]
                    letter_norm = imresize(letter_raw ,(20 ,20))
                    final.append(letter_norm)
                    prev_tr = lines[i][j][3]
                if j == (len(lines[i])-1):
                    prev_line_br = lines[i][j][2]
            prev_tr = 0
            tl_2 = 0
        print 'Characters recognized: ' + str(len(final))
        return final


    def __init__(self):
        print "Extracting characters..."
```

```
import time
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.patches as mpatches
from scipy.misc import imread,imresize,imsave
from skimage.segmentation import clear_border
from skimage.morphology import label
from skimage.measure import regionprops
from extract_letters import Extract_Letters
start_time = time.time()
extract = Extract_Letters()
training_files = ['./ocr/training/training1.png',
'./ocr/training/training2.png','./ocr/training/training3.png','./ocr/training/training4.png','.
/ocr/training/training6.png']

folder_string = 'abcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyz123456789'
name_counter = 600
for files in training_files:
     letters = extract.extractFile(files)
     string_counter = 0


     for i in letters:
      if string_counter > 60:
        string_counter = 0
      imsave('./training_type/' + str(folder_string[string_counter]) + '/' + str(name_counter) +
'_snippet.png', i)
      #print 'training character: ' + str(folder_string[string_counter]) + ' (' +
str(name_counter) + '/' + str(len(letters)) + ')'
      string_counter += 1
      name_counter += 1
print time.time() - start_time, "seconds"
```

Train SVM.py

```
import numpy as np
import os
import time
import itertools
import operator
import random
import matplotlib.pyplot as plt
import matplotlib.cm as cm
from skimage.feature import hog
from skimage import color, exposure
from scipy.misc import imread,imsave,imresize
import numpy.random as nprnd
from sklearn.svm import SVC
from sklearn import linear_model
from sklearn.svm import LinearSVC
import matplotlib
import pickle

start_time = time.time()
#paths for the training samples
path_char =
['a','b','c','d','e','f','g','h','i','j','k','l','m','n','o','p','q','r','s','t','u','v','w','x
','y','z','1','2','3','4','5','6','7','8','9']
path_root = './training_type/'
filenames = list()

for x in range(len(path_char)):
     filenames.append([])

print 'Loading files...'
```

```
#get all the images in the above folders
#note that we are looking for specific extensions .png
for i in range(len(filenames)):
     filenames[i] = sorted([filename for filename in os.listdir(path_root+path_char[i]) if
filename.endswith('.png')])

#add the full path to all the filenames
for i in range(len(filenames)):
     filenames[i] = [path_root+path_char[i]+'/'+filename for filename in filenames[i]]

for i in range(len(filenames)):
     print 'Number of training images -> '+str(path_char[i])+': ' + str(len(filenames[i]))

data = list()
labels = list()

for i in range(len(filenames)):
     for filename in filenames[i]:
      #print filename
      #read the images
      image = imread(filename,1)
        #if using hog features then the parameters should be appropriate for the image size.
The following won't be good if the size is 20x20
      #hog_features = hog(image, orientations=12, pixels_per_cell=(16, 16),cells_per_block=(1,
1))

      #print hog_features.shape
      data.append(image.ravel())
      labels.append(path_char[i])
     print 'Finished adding ' + str(path_char[i]) + ' samples to dataset'

#print data
#print labels
print 'Training the SVM'
#create the SVC
clf = LinearSVC(dual=False,verbose=1)
#train the svm
clf.fit(data, labels)

#pickle it - save it to a file
pickle.dump( clf, open( "char.detector", "wb" ) )
print time.time() - start_time, "seconds"
```

Train KNN.py

```
import numpy as np
import os
import time
import itertools
import operator
import random
import matplotlib.pyplot as plt
import matplotlib.cm as cm
from skimage.feature import hog
from skimage import color, exposure
from scipy.misc import imread,imsave,imresize
import numpy.random as nprnd
from sklearn.svm import SVC
from sklearn import linear_model
from sklearn.svm import LinearSVC
import matplotlib
import pickle
from sklearn.neighbors import KNeighborsClassifier

start_time = time.time()
```

```
#paths for the training samples
path_char =
['a','b','c','d','e','f','g','h','i','j','k','l','m','n','o','p','q','r','s','t','u','v','w','x
','y','z','1','2','3','4','5','6','7','8','9']
path_root = './training_type/'
filenames = list()

for x in range(len(path_char)):
     filenames.append([])

print 'Loading files...'

#get all the images in the above folders
#note that we are looking for specific extensions .png
for i in range(len(filenames)):
     filenames[i] = sorted([filename for filename in os.listdir(path_root+path_char[i]) if
filename.endswith('.png')])

#add the full path to all the filenames
for i in range(len(filenames)):
     filenames[i] = [path_root+path_char[i]+'/'+filename for filename in filenames[i]]

for i in range(len(filenames)):
     print 'Number of training images -> '+str(path_char[i])+': ' + str(len(filenames[i]))

data = list()
labels = list()

for i in range(len(filenames)):
     for filename in filenames[i]:
      #print filename
      #read the images
      image = imread(filename,1)
        #if using hog features then the parameters should be appropriate for the image size.
The following won't be good if the size is 20x20
      #hog_features = hog(image, orientations=12, pixels_per_cell=(16, 16),cells_per_block=(1,
1))

      #print hog_features.shape
      data.append(image.ravel())
      labels.append(path_char[i])
     print 'Finished adding ' + str(path_char[i]) + ' samples to dataset'

k=10
knn_classifier = KNeighborsClassifier(n_neighbors=k)
knn_classifier.fit(data, labels)
print 'Trained the knn-classifier'
clf = knn_classifier

#pickle it - save it to a file
pickle.dump( clf, open( "char.detector", "wb" ) )
print time.time() - start_time, "seconds"
```

Read File.py

```
import numpy as np
import string
import time
import matplotlib.pyplot as plt
import matplotlib.patches as mpatches
from skimage.feature import hog
from scipy.misc import imread,imresize
from skimage.segmentation import clear_border
from skimage.morphology import label
from skimage.measure import regionprops
from sklearn.svm import SVC
```

```
from sklearn.svm import LinearSVC
import pickle
from sys import stdout
from extract_letters import Extract_Letters

start_time = time.time()
extract = Extract_Letters()


test_file_path = './ocr/testing/adobe.png'

#load the detector
clf = pickle.load( open("char.detector","rb"))

print 'extracting files...'
letters = extract.extractFile(test_file_path)

recognised = list()

print 'recognised text:'
for letter in letters:
     result_type = clf.predict(letter.ravel())#hog isn't good for small images unless you
change the parameters
     recognised.append(result_type)


for i in range(len(recognised)):
     stdout.write(recognised[i])
print ''

truth
='inaregulatorydocumentfiledwiththesectodayadobeannouncedthatchieftechnologyofficerkevinlynchwo
uldbetakinghisleaveasofthiscomingfridayonmarch182013kevinlynchresignedfromhispositionasexecutiv
evicepresidentchieftechnologyofficerofadobesystemsincorporatedeffectivemarch222013topursueother
opportunitiesthefilingreadslynchwhocametothecompanyin2005duringitsacquisitionofmacromedialedado
beschargeintosomeofthemorecuttingedgeareasoftechnologyincludingmultiscreencomputingcloudcomputi
ngandsocialmediaforagesadobehadbeenrootedintheworkflowsoftheprintdesigncommunitylynchwasrespons
ibleforthecompanysshiftintowebpublishingstartingwithdreamweaverhealsooversawadobesresearchandex
periencedesignteamsandwasasadobeputsitinchargeofshapingadobeslongtermtechnologyvisionandfocusin
ginnovationacrossthecompanyduringatransformativetimerumorsaroundthewebhavepinpointedappleslync
hsnextdestinationanditsnotanentirelynonsensicalrumoradobestransitiontowebtechnologieshasbeennot
hingifnotprofitableapplestillagiantinconsumerhardwarecoulduseahelpinghandwhenitcomestomultiscre
enfluiditysocialmediaandwebbasedsoftware                '
#'nvidiasgraphicschipsarentusedjustforgamesanymoresincethecompanymadeitschipsprogrammabletheyar
etakingonfunctionsthatintelmicroprocessorsdoandmorethanevertheyarebeingusedforbigdatanumbercrun
chingandsolvingproblemsthatpreviouslyhadtobedonebynongraphicssupercomputersjenhsunhuang(picture
d)chiefexecutiveofnvidiasaidinakeynotespeechatthecompanysgputechconferencethattopenterpriseandm
obileappcompaniesareusinggpus(graphicsprocessingunits)tohandlebigdataanalyticsadvancesearchandt
askslikefacerecognitiontheseapplicationsarespreadacrossthecommercialandconsumerspacescustomersw
hoareusingnvidiagpusinserversmadebycomputermakersincludeshazamsalesforcecomandcortexicatheyareu
singnvidiateslagpusforcomputingtasksbeyondtraditionalgraphicsprocessingtheproblemsathandinclude
audiosearchsuchasshazamsservicethattellsyouwhatsongyourelisteningtoontheradiogpusenableustohand
leourtremendousprocessingneedsatasubstantialcostsavingsdeliveringtwicetheperformanceperdollarco
mparedtoacpubasedsystemsaidjasontituschieftechnologyofficerofshazamentertainmentweareaddingmill
ionsofvideoandforeignlanguageaudiotrackstoourexistingservicesandgpuacceleratorsgiveusawaytoachi
evescalablegrowthshazamcanidentifytheacousticfingerprintofsongsfroma27milliontrackdatabaseandth
enserveanswersto300millionuserspeoplearedoing10millionsongsearchesaday'
truth = truth.lower()
print truth

score = 0
for i in range(len(recognised)):
     if truth[i] == recognised[i]:
      score += 1
print score
```

```
print len(truth)
percent = (float(score)/len(truth))*100
print str(percent) + '% correct'
print time.time() - start_time, "seconds"
```

Handwritten

Extract Letters.py

```
# Sample code for AI Coursework
# Extracts the letters from a printed page.

# Keep in mind that after you extract each letter, you have to normalise the size.
# You can do that by using scipy.imresize. It is a good idea to train your classifiers
# using a constast size (for example 20x20 pixels)
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.patches as mpatches
from scipy.misc import imread,imresize
from skimage.segmentation import clear_border
from skimage.morphology import label
from skimage.measure import regionprops

class Extract_Letters:
    def extractFile(self, filename):
        image = imread(filename,1)

        #apply threshold in order to make the image binary
        bw = image < 120

        # remove artifacts connected to image border
        cleared = bw.copy()
        #clear_border(cleared)

        # label image regions
        label_image = label(cleared,neighbors=8)
        borders = np.logical_xor(bw, cleared)
        label_image[borders] = -1


        fig = plt.figure()
        #ax = fig.add_subplot(131)
        #ax.imshow(bw, cmap='jet')

        letters = list()
        order = list()

        for region in regionprops(label_image, ['Area', 'BoundingBox']):
            minc, minr, maxc, maxr = region['BoundingBox']
            # skip small images
            if maxc - minc > len(image)/250: # better to use height rather than area.
                rect = mpatches.Rectangle((minc, minr), maxc - minc, maxr - minr,
                                          fill=False, edgecolor='red', linewidth=2)
                order.append(region['BoundingBox'])


        #sort the recognised characters left->right, top->bottom
        lines = list()
        first_in_line = ''
        counter = 0

        #worst case scenario there can be 1 character per line
        for x in range(len(order)):
            lines.append([])
```

```
        for character in order:
            if first_in_line == '':
                first_in_line = character
                lines[counter].append(character)
            elif abs(character[0] - first_in_line[0]) < (first_in_line[2] - first_in_line[0]):
                lines[counter].append(character)
            elif abs(character[0] - first_in_line[0]) > (first_in_line[2] - first_in_line[0]):
                first_in_line = character
                counter += 1
                lines[counter].append(character)


        for x in range(len(lines)):
            lines[x].sort(key=lambda tup: tup[1])

        final = list()
        prev_tr = 0
        prev_line_br = 0

        for i in range(len(lines)):
            for j in range(len(lines[i])):
                tl_2 = lines[i][j][1]
                bl_2 = lines[i][j][0]
                if tl_2 > prev_tr and bl_2 > prev_line_br:
                    tl,tr,bl,br = lines[i][j]
                    letter_raw = bw[tl:bl,tr:br]
                    letter_norm = imresize(letter_raw ,(20 ,20))
                    final.append(letter_norm)
                    prev_tr = lines[i][j][3]
                if j == (len(lines[i])-1):
                    prev_line_br = lines[i][j][2]
            prev_tr = 0
            tl_2 = 0
        print 'Characters recognized: ' + str(len(final))
        return final


    def __init__(self):
        print "Extracting characters..."
```

Train Handwriting.py

```
import time
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.patches as mpatches
from scipy.misc import imread,imresize,imsave
from skimage.segmentation import clear_border
from skimage.morphology import label
from skimage.measure import regionprops
from extract_letters import Extract_Letters
start_time = time.time()
extract = Extract_Letters()

training_files =
['./ocr/training/a.png','./ocr/training/b.png','./ocr/training/c.png','./ocr/training/d.png','.
/ocr/training/e.png','./ocr/training/f.png','./ocr/training/g.png','./ocr/training/h.png','./oc
r/training/i.png','./ocr/training/j.png','./ocr/training/k.png','./ocr/training/l.png','./ocr/t
raining/m.png','./ocr/training/n.png','./ocr/training/o.png','./ocr/training/p.png','./ocr/trai
ning/q.png','./ocr/training/r.png','./ocr/training/s.png','./ocr/training/t.png','./ocr/trainin
g/u.png','./ocr/training/v.png','./ocr/training/w.png','./ocr/training/x.png','./ocr/training/y
.png','./ocr/training/z.png']

letter_string = 'abcdefghijklmnopqrstuvwxyz'

letter_counter = 0
```

```
for files in training_files:
     letters = extract.extractFile(files)
     name_counter = 0
     for i in letters:
      imsave('./training_type/' + str(letter_string[letter_counter]) + '/' + str(name_counter) +
'_snippet.png', i)
      name_counter += 1
     letter_counter += 1
print time.time() - start_time, "seconds"
```

Train SVM.py

```
import numpy as np
import os
import time
import itertools
import operator
import random
import matplotlib.pyplot as plt
import matplotlib.cm as cm
from skimage.feature import hog
from skimage import color, exposure
from scipy.misc import imread,imsave,imresize
import numpy.random as nprnd
from sklearn.svm import SVC
from sklearn import linear_model
from sklearn.svm import LinearSVC
import matplotlib
import pickle

start_time = time.time()
#paths for the training samples
path_char =
['a','b','c','d','e','f','g','h','i','j','k','l','m','n','o','p','q','r','s','t','u','v','w','x
','y','z','1','2','3','4','5','6','7','8','9']
path_root = './training_type/'
filenames = list()

for x in range(len(path_char)):
     filenames.append([])

print 'Loading files...'

#get all the images in the above folders
#note that we are looking for specific extensions .png
for i in range(len(filenames)):
     filenames[i] = sorted([filename for filename in os.listdir(path_root+path_char[i]) if
filename.endswith('.png')])

#add the full path to all the filenames
for i in range(len(filenames)):
     filenames[i] = [path_root+path_char[i]+'/'+filename for filename in filenames[i]]

for i in range(len(filenames)):
     print 'Number of training images -> '+str(path_char[i])+': ' + str(len(filenames[i]))

data = list()
labels = list()

for i in range(len(filenames)):
     for filename in filenames[i]:
      #print filename
      #read the images
      image = imread(filename,1)
        #if using hog features then the parameters should be appropriate for the image size.
The following won't be good if the size is 20x20
```

```
      #hog_features = hog(image, orientations=12, pixels_per_cell=(16, 16),cells_per_block=(1,
1))

      #print hog_features.shape
      data.append(image.ravel())
      labels.append(path_char[i])
     print 'Finished adding ' + str(path_char[i]) + ' samples to dataset'

#print data
#print labels
print 'Training the SVM'
#create the SVC
clf = LinearSVC(dual=False,verbose=1)
#train the svm
clf.fit(data, labels)

#pickle it - save it to a file
pickle.dump( clf, open( "char.detector", "wb" ) )
print time.time() - start_time, "seconds"
```

Train KNN.py

```
import numpy as np
import os
import time
import itertools
import operator
import random
import matplotlib.pyplot as plt
import matplotlib.cm as cm
from skimage.feature import hog
from skimage import color, exposure
from scipy.misc import imread,imsave,imresize
import numpy.random as nprnd
from sklearn.svm import SVC
from sklearn import linear_model
from sklearn.svm import LinearSVC
import matplotlib
import pickle
from sklearn.neighbors import KNeighborsClassifier

start_time = time.time()
#paths for the training samples
path_char =
['a','b','c','d','e','f','g','h','i','j','k','l','m','n','o','p','q','r','s','t','u','v','w','x
','y','z','1','2','3','4','5','6','7','8','9']
path_root = './training_type/'
filenames = list()

for x in range(len(path_char)):
     filenames.append([])

print 'Loading files...'

#get all the images in the above folders
#note that we are looking for specific extensions .png
for i in range(len(filenames)):
     filenames[i] = sorted([filename for filename in os.listdir(path_root+path_char[i]) if
filename.endswith('.png')])

#add the full path to all the filenames
for i in range(len(filenames)):
     filenames[i] = [path_root+path_char[i]+'/'+filename for filename in filenames[i]]

for i in range(len(filenames)):
     print 'Number of training images -> '+str(path_char[i])+': ' + str(len(filenames[i]))
```

```
data = list()
labels = list()

for i in range(len(filenames)):
     for filename in filenames[i]:
      #print filename
      #read the images
      image = imread(filename,1)
        #if using hog features then the parameters should be appropriate for the image size.
The following won't be good if the size is 20x20
      #hog_features = hog(image, orientations=12, pixels_per_cell=(16, 16),cells_per_block=(1,
1))

      #print hog_features.shape
      data.append(image.ravel())
      labels.append(path_char[i])
     print 'Finished adding ' + str(path_char[i]) + ' samples to dataset'

k=10
knn_classifier = KNeighborsClassifier(n_neighbors=k)
knn_classifier.fit(data, labels)
print 'Trained the knn-classifier'
clf = knn_classifier

#pickle it - save it to a file
pickle.dump( clf, open( "char.detector", "wb" ) )
print time.time() - start_time, "seconds"
```

Read File.py

```
import numpy as np
import string
import time
import matplotlib.pyplot as plt
import matplotlib.patches as mpatches
from skimage.feature import hog
from scipy.misc import imread,imresize
from skimage.segmentation import clear_border
from skimage.morphology import label
from skimage.measure import regionprops
from sklearn.svm import SVC
from sklearn.svm import LinearSVC
import pickle
from sys import stdout
from extract_letters import Extract_Letters

start_time = time.time()
extract = Extract_Letters()


test_file_path = './ocr/testing/brownfox.png'

#load the detector
clf = pickle.load( open("char.detector","rb"))

print 'extracting files...'
letters = extract.extractFile(test_file_path)

recognised = list()

print 'recognised text:'
for letter in letters:
     result_type = clf.predict(letter.ravel())#hog isn't good for small images unless you
change the parameters
     recognised.append(result_type)
```

```
for i in range(len(recognised)):
     stdout.write(recognised[i])
print ''


truth = 'aquickbrownfoxjumpsoverthelazydog'

truth = truth.lower()
print truth

score = 0
for i in range(len(recognised)):
     if truth[i] == recognised[i]:
      score += 1
print score
print len(truth)
percent = (float(score)/len(truth))*100
print str(percent) + '% correct'
print time.time() - start_time, "seconds"
```

Signature Recongition
Train Classifier.py

```
import numpy as np
import os
import itertools
import operator
import random
import matplotlib.pyplot as plt
import matplotlib.cm as cm
from skimage.feature import hog
from skimage import color, exposure
from scipy.misc import imread,imsave,imresize
import numpy.random as nprnd
from sklearn.svm import SVC
from sklearn import linear_model
from sklearn.svm import LinearSVC
import matplotlib
import pickle

if __name__ == '__main__':
    #paths for the training samples
    path_b = './training/brandon/'
    path_a = './training/andre/'

    b_filenames = sorted([filename for filename in os.listdir(path_b) if
(filename.endswith('.jpg') or filename.endswith('.png') or (filename.endswith('.bmp'))) ])
    a_filenames = sorted([filename for filename in os.listdir(path_a) if
(filename.endswith('.jpg') or filename.endswith('.png') or (filename.endswith('.bmp'))) ])

    #add the full path to all the filenames
    b_filenames = [path_b+filename for filename in b_filenames]
    a_filenames = [path_a+filename for filename in a_filenames]

    print 'Number of training images -> brandon: ' + str(len(b_filenames))
    print 'Number of training images -> andre: ' + str(len(a_filenames))

    #create the list that will hold ALL the data and the labels
    #the labels are needed for the classification task:
    # 0 -> brandon
    # 1 -> andre
    data = []
    labels = []
```

```
    #fill the training dataset
    # the flow is
    # 1) load sample
    # 2) resize it to (200,200) so that we have save size for all the images
    # 3) get the HOG features of the resized image
    # 4) save them in the data list that holds all the hog features
    # 5) also save the label (target) of that sample in the labels list
    for filename in b_filenames:
        #read the images
        image = imread(filename,1)
        #flatten it
        image = imresize(image, (200,200))
        hog_features = hog(image, orientations=12, pixels_per_cell=(16, 16),
                    cells_per_block=(1, 1))
        data.append(hog_features)
        labels.append(0)
    print 'Finished adding brandon samples to dataset'

    for filename in a_filenames:
        image = imread(filename,1)
        image = imresize(image, (200,200))
        hog_features = hog(image, orientations=12, pixels_per_cell=(16, 16),
                    cells_per_block=(1, 1))
        data.append(hog_features)
        labels.append(1)
    print 'Finished adding andre samples to dataset'

    print 'Training the SVM'
    #create the SVC
    clf = LinearSVC(dual=False,verbose=1)
    #train the svm
    clf.fit(data, labels)

    #pickle it - save it to a file
    pickle.dump( clf, open( "sig.detector", "wb" ) )
```

Signature Detector.py

```
import numpy as np
from skimage.feature import hog
from scipy.misc import imread,imresize
from sklearn.svm import SVC
from sklearn.svm import LinearSVC
import pickle

if __name__ == '__main__':
    #load the detector
    clf = pickle.load( open("sig.detector","rb"))

    #now load a test image and get the hog features.
    test_image = imread('b12.png',1)
    test_image = imresize(test_image, (200,200))

    hog_features = hog(test_image, orientations=12, pixels_per_cell=(16, 16),
                    cells_per_block=(1, 1))

    result_type = clf.predict(hog_features)
    result = ["Brandon", "Andre"]

    print result[result_type]
```