# Computer Vision
# Character recognition

A Kaggle challenge

By Rick Horeman

Obligatory funny picture (aka meme, /mi:m/):

# Contents:

Introduction – *The Challenge* ..... 3
The Paper Rabbit Hole – *Down we go* ..... 4
Dataset troubles – *100% doubts* ..... 4
Accurate Accuracy – *Alliteration!* ..... 5
Experimentation – *Expedite* ..... 5
Observations and Conclusions – *What do you elf-eyes see?* ..... 6
What can still be done – *Temporal limitations* ..... 7
Git – *good* ..... 7
The end – *Because all things must* ..... 8

## Introduction – *The Challenge*

For starters, this whole thing was for a school assignment, for computer vision to be precise.

We had two choices: A replication study or a (Kaggle) challenge. Initially I wanted to do the first, but after choosing this paper we found out there was a Kaggle challenge focused around the same paper and dataset, so in order to have some more freedom, I chose this option after all.

The paper in question is 「T. E. de Campos, B. R. Babu and M. Varma. Character recognition in natural images. In Proceedings of the International Conference on Computer Vision Theory and Applications (VISAPP), Lisbon, Portugal, February 2009.」 and can be found on their website at: http://www.ee.surrey.ac.uk/CVSSP/demos/chars74k/

The paper, put very abridgedly, uses a Bag-Of-Visual words technique, with six different feature extraction methods, to achieve a fairly accurate optical character recognition on real-life text, think of packaging, advertisements, road signs, etc.

The above website provides said "Img" dataset, as well as a "Hnd" dataset, containing handwritten letters, and a "Fnt" dataset, comprised of computer fonts in various versions such as Normal, **Bold**, *Italic* and ***Bold Italic***.

I ended up using the "Fnt" dataset, so as to not make things too hard on myself, and with the idea to actually achieve a usable accuracy, say, >80%.

*Some examples form the Fnt dataset*

## The Paper Rabbit Hole – *Down we go*

At first my plan was to implement the exact same methods as used in the paper.

The three methods I decided to go with, of which I wanted to implement at least one, are as follows:

- Geometric Blur
- Spin Image
- Patch Descriptor

I started with looking into the first one, but the sources referenced in the paper were rather abstract and didn't provide almost any implementation details; not that I could understand anyways.

So I decided to move on to a different method and patch descriptor seemed like the easiest one of the three. But after looking into it for pretty much an entire day I couldn't really make enough sense of it to go and implement it. Notably none of these papers had any code examples or pseudo code referenced.

After consulting with my teacher we concluded it was the best course of action to go for a simpler implementation, so that I would at least have something that could make *a* prediction, even if not that accurate. With that out of the way I started working on loading in the dataset with the provided lists for training, testing, etc.

## Dataset troubles – *100% doubts*

To start things off I wanted to keep it simple and just chuck the images, after a resize to 32x32[1], into the sklearn Support Vector Machine. At first I tried this with all 62 classes, but after waiting some time there didn't really seem to be much happening, or rather, it seemed to take *ages*. So I decided to start with just the first 10 classes, the digits 0 through 9. After training the SVM and testing it with the provided train/test set split I got a rather curious result… The thing was 100% accurate.

While this seemed too good too be true I thought that maybe the sklearn implementation of the support vector machine was just *that* good that it could get such accuracy with only 10 classes. So I decided to let it run with all classes, which took a good couple of hours. The result was a roughly 95.9% accuracy, which honestly still seemed too good to be true.

So I went and investigated the provided lists and after some testing it turned out that both sets contained *all* images, so they overlapped 100%, no wonder the accuracy was so high :/. And to make things worse the sets contained all images multiple times, making training and testing times take way longer than they really needed to.

I then went and made my own rather simple split for training and testing. I took all images and used a random number generator to split them roughly 2/3 training and 1/3 testing. The major downside here being that they would be different sets each time and there was no guaranteeing that there was a good balance of classes in both sets (it's theoretically possible no instances of the class 0 would be in training).

[1] The images were originally all 128x128

## Accurate Accuracy – *Alliteration!*

Now that the training and testing sets no longer overlap, I ran everything again and got an accuracy of roughly 87.9%, which seems much more believable. With a usable dataset and the basic testing code set up it was time to start switching stuff up and seeing what it does to the accuracy.

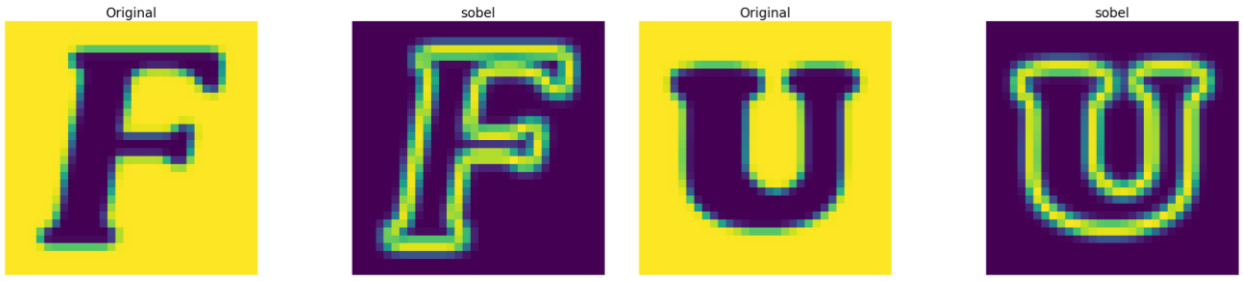
## Experimentation – *Expedite*

At this point I started to run out of time for extensive testing with various different methods. I had wanted to test a range of things from different resolution to filters and other methods from the paper.

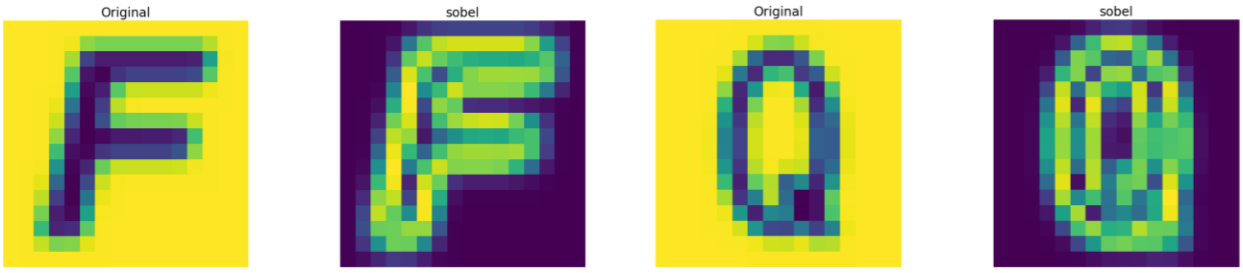
But sadly I only had time to mess with the resolution and try out a Sobel filter, before putting the images into the SVM. The below table contains the results of those experiments, under that are some example images and then my observations and conclusions.

| Resolution | No filter | Sobel Edge Detection |
|---|---|---|
| 64x64 | 88.7% | - |
| 32x32 | 87.9% | 85.7% |
| 16x16 | 83.0% | 81.8% |
| 8x8 | - | 74.9% |

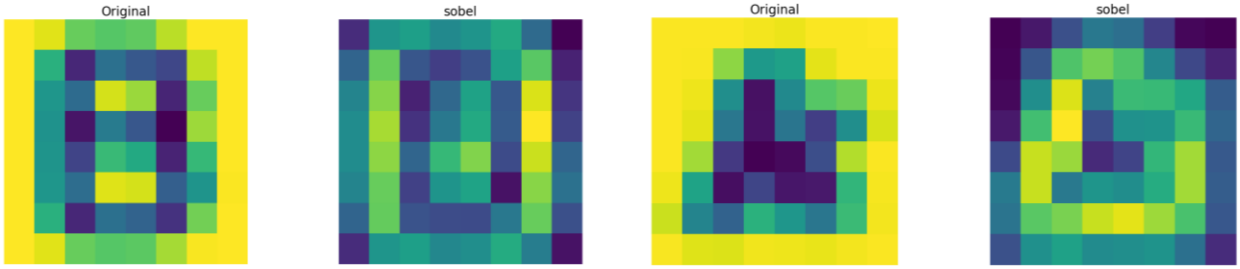
These values may deviate with roughly 0.5%, due to the test and train sets being somewhat variable.




*Original and Sobel at 32x32*




*Original and Sobel at 16x16*




*Original and Sobel at 8x8 – I can understand why this isn't very accurate*

## Observations and Conclusions – *What do you elf-eyes see?*

I started with the unprocessed images at 32x32, which gave me really good results. I then decided to lower the resolution, because I'd seen other (although with the Img dataset) have success with this.

Well that didn't turn out like expected, for both unprocessed and Sobel lowering the resolution seems to only decrease the accuracy in my particular use case.

So why not go the other way? Increase the resolution… Well that's easier said than done, increasing to 64x64 made everything *so much* slower, like, maybe exponentially? Anyways it was enough for me to feel like the marginal increase in accuracy was not worth the additional amount of time it took to train and test everything.

In the end it seems like, from what I have had the time to test, just letting the Support Vector Machine make it's magic work using the unprocessed, downscaled to 32x32, images.

Which leads me to the next point;

## What can still be done – *Temporal limitations*

> "One of the basic rules of the universe is that nothing is perfect. Perfection simply doesn't exist.....Without imperfection, neither you nor I would exist"

Or put rather straight-forwardly, it should be possible to make this thing far more accurate!

There are still six whole methods described/referenced in the paper that could possibly, either on their own or combined, help increase the accuracy of this computer vision solution. They go as follows:

- Shape Contexts
- Geometric Blur
- Scale Invariant Feature Transform
- Spin image
- Maximum Response of filters
- Patch descriptor

In particular I'm interested to see what a relatively simple version of the patch descriptor would do.

I would take 8x8 patches from the 32x32 downscaled images and put them, as is, into a K-means clustering, for example the sklearn implementation. And then use the cluster responses to train the Support Vector Machine, then at prediction-time get the cluster responses for the input images and see what the SVM has to say about it. I reckon this might boast significantly higher accuracy than my current methods, because this method retains the clean-ness of the source images, but reduces how exact the test images would have to resemble the train images. Basically it reduces the issue to general shape recognition rather than exact pixels.

This method could then be expanded upon by first putting filters over the images, although I doubt this would help much with the already rather clean Fnt dataset, but with the Img dataset it would certainly do good.

## Git – *good*

All code as well as another document I kept while working called 'worklog.txt' can be found on my github repo at 『https://github.com/RickHoreman/Computer-Vision』.

There is also a README containing links to the paper, dataset and Kaggle challenge.

This document too can be found there, and that is likely how you found it.

## The end – *Because all things must*

I don’t really have much left to say, so enjoy this picture of a concrete slab.