Intro to image classification with KNN

Akash Goswami Following

Feb 8, 2018 · 9 min read

The first time I came across a camera that could detect faces, placing a square around and suggesting your age, my jaw simply dropped.

When I first used text-based image search on my smartphone, or checked out Google Cloud Vision, I was further intrigued. I wondered about the powerful algorithms behind those systems (who wouldn't, right?) so I started looking into the concepts around them.

I was overwhelmed by the sheer volume of prerequisite knowledge needed to understand them!

These algorithms tap into concepts from several branches of mathematics to make sense of what's going on under the hood. Of course, you can directly start working with libraries which abstract most of these concepts for you, but without a basic grasp over the terrain, chances are that you will hit a dead end.

That's why I want to address few essentials on k-nearest neighbors (KNN) algorithms for image classification — tiny baby steps which helped me understand more complex ideas.

Before we begin, let's get the expectation right.

This write-up is mostly about me taking you on my learning journey. At the end of this, you won't know how to make vision systems which can be used in production. You won't be learning about SVM, softmax, gradient descent, back propagation, neural nets or anything on those lines.

But you will grow intuition on how classification problems are approached and how a machine thinks of an entity (the definition of 'entity' is entirely up to you; an image, a word, or some text in paragraph).

. . .

What is image classification?

We can assume a blackbox function which takes an animal as image and returns the text label of the animal.

(You might say, that's absurd! How can I possibly take a cat as input? Hold on to your horses, I'll literally show you how to take a cat in a function. You might also ask how a

function can associate a label with a shape. You will have to wait to get the answer, because if I explain now you won't get the full picture.)

This, in a nutshell, is image classification. The mind-blowing system which detects faces or suggests age or powers Google Cloud Vision API, implements computer vision tasks (such as object detection or segmentation) which basically boils down to image classification.

I'm explaining image classification with KNN because this is one algorithm that needs almost no prerequisites. So, the cognitive part of your brain does not have to worry about too many unknowns.

. . .

What is KNN algorithm?

Let's start with a new problem here:

If two sets of points are given on an XY plane, with one set colored in green and other set colored in red, what would be the color of the any new point added on the same XY plane?

Here is the setup:

Assume that the colored points belong to two separate distinct groups. Now intuitively, we know if the new point leans towards the bottom-left side of the canvas, it probably belongs to the green (or shades of green) group, otherwise it would be in the red (or shades of red) group.

In a way you are searching for nearest neighbors. Simple, isn't it? And that is what KNN is, from a human intuition perspective.

However, the rub is in the following line:

> leans towards the bottom-left side of the canvas

Computers don't understand contextual information of this kind. We have to explicitly quantify the meaning of 'bottom-left' and pass it on to the program. To quantify this, we use Euclidean distance between two points, described as the following:

This equation is generalized to cover any dimension. For 2D space, the equation reduces to the following:

This is the same as:

We are all familiar with the last equation — it's the distance between two points on an XY plane. Understanding the output of these equations is a cakewalk for our computer program, so once we have this tool, the rest of the algorithm is pretty intuitive.

- Calculate the distance between the new point and every other point

- Sort the distances

- Pick K minimum distances from the list of sorted distances (thats why it's called **K**-NN)
- Use majority voting to get the color of the point

. . .

Here is the link of the full interactive demo of the algorithm.

. . .

Image classification intuition with KNN

Each point in the KNN 2D space example can be represented as a vector (for now, a list of two numbers). All those vectors stacked vertically will form a matrix representing all the points in the 2D plane.

On a 2D plane, if every point is a vector, then the Euclidean distance (scalar) can be derived from the following diagram:

This, essentially, is geometrical representation of the equation we discussed in the last section. Here's a quick recap:

and

Now think of a *32 x 32* cat image.

We have a total 32 x 32 = 1024 pixels. So, if we have to represent an image using a structure which the computer can understand, we will have a large vector that's 1024 element long.

But what will be the value of the numbers? Of course, the color which the pixel holds.

A pixel's color can be represented using 3 color values (assume RGB color channel) so a pixel holds a list of 3 values (1 value for r, 1 value for g and 1 value for b of RGB channel). We code this in a similar structure, but we create 1024 values of r, followed by 1024 values of g, followed by 1024 values of b. That's a total of 3072 values in a vector.

If we have no more than two animals then we have vector length of 3072 stacked vertically. The matrix would look like this:

Now this is the answer to the question asked earlier

> *How can I possibly take a cat as input?*

This is how we can pass a cat into the function. And we are doing all this because obviously, a program does not know how to handle a cat. That's why we've encoded the cat in a way that our program can understand.

A point represented in 2D plane has 2 components in its vector. Similarly, the image that we see in 2D is a point in higher dimensions. The beauty of linear algebra is that it scales the idea which we apply in 2D to higher dimensions (hence procedure for calculating Euclidean distance for higher dimensions is still valid).

Just like we grouped (clustered) our points in 2D space, grouping (clustering) can be done in higher dimensions also where image is represented as a point. So the cat image should cluster around some specific point also (it's hard to visualize, but you can think the idea in a lower dimension and scale it up to a higher dimension).

Once we are comfortable with this idea, we can comfortably apply KNN for the image, just like we had done it for our 2D points.

As you may have noticed, the setup for our 2D problem statement already had some points in place before a new point added for classification. The same concept scales for classifying images too. There are image points in higher dimensional space which are labelled as cat, dog etc. similar to categorization of green and red.

And this is the answer of other question we asked earlier:

> *How can a function associate a label with a shape*

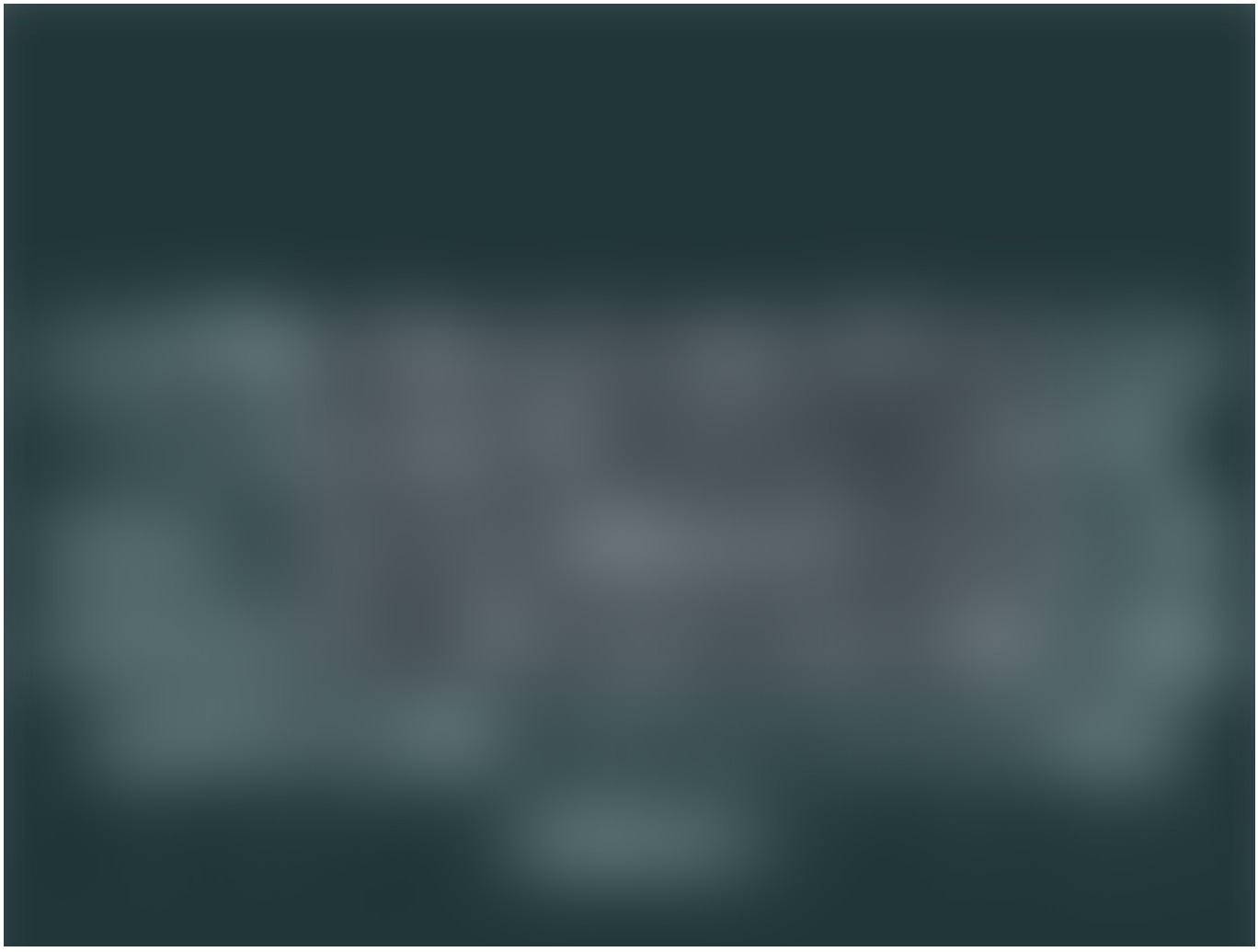
We pass an encoded shape (cat, dog, etc.) and tell the program this is ‘cat’, this is ‘dog’. The program represents images as points on the higher dimensional space and assigns them a label. This is called *training the model*.

Once training is complete, we pass test data to the model to predict the animal, just like we gave a new point in the 2D space to determine its color.

. . .

Food for thought

If high dimensionality intrigues you, this is what you might want to read. Here is my one of many shower thoughts



. . .

Practical

Enough with theory, it's now time to get hands on.

You can find all the code here. I used snippets from this notebook to explain the practical workflow. The notebook explain most of the process, so I won't repeat it here.

I will, however, add the missing links which are not mentioned or concepts that are not explained.

Preparing the data

When I write a classifier/regression function, I prepare my own data in 2D or 3D space. Yes, it takes a bit of time but it helps you debug really fast.

Debugging a model with real training and test data is a nightmare. Of course creating my own data won't make this program bug free, but this approach ensures the rudimentary bugs could be found fast and fixed timely.

The code for preparing the data is here.

The idea is simple: create cluster centers based on the number of classes you want to create. Randomly generate more points around the cluster center, scaled by a predefined standard deviation.

Return value of this function is (X, y). X contains vectors stacked vertically and y is the color of the point. Every item in the stack is a point on XY plane.

Visualize

Visualize the points on the plane to check if the clusters are created correctly.

Algorithm

The actual KNN algorithm is here.

I used numpy arrays to perform the distance calculation. If you don't understand in one shot, that's okay. What I do when I don't understand a program, is to execute each line in Python REPL to see what the input is and what's the output. This helps understand why it was done this particular method.

It takes some time, but it really helps you build a solid concept. And every so often, I find useful functions / transformations / operations which reduces 5–10 lines of code!

Test

Test the algorithm with our custom generated data.

Training set, Validation set, Test set

I used data from CIFAR-10 which is a small dataset of 10 classes. This dataset is already segregated into training set and test set. However, we should create a validation set from training set using the slicing ratio 1:9 (if training set had 1k rows post slicing 100 rows goes in validation set 900 rows goes in training set).

And we need this validation set to tweak our hyperparameter (parameter which are not derived from learning process but is a part of learning process) — K.

Note that I've used training set and test set only for illustration purpose in the codebase.

Outro

The rest of the process is very simple and sufficiently explained in the notebook.

But as you can see, the accuracy of the algorithm is not something we can be too proud of. What's interesting is that even the state of the art implementation of the algorithm produces similar results — clearly, we still have a long way to go, once our bases are covered.

. . .

Further Reading & Ref

1.6. Nearest Neighbors - scikit-learn 0.19.1 documentation

Despite its simplicity, nearest neighbors has been successful in a large number of classification and regression...

scikit-learn.org

http://vision.stanford.edu/teaching/cs231n-demos/knn/

. . .

Codebase

adotg/knn-what-how-why

knn-what-how-why - What is knn? How is knn done? Why is knn needed?

github.com

Machine Learning Knn Image Recognition Classification Image Classification

About Help Legal