K-Nearest-Neighbors (Sacha BESSON)

DataBase : https://www.kaggle.com/datasets/jerems69/german-signs

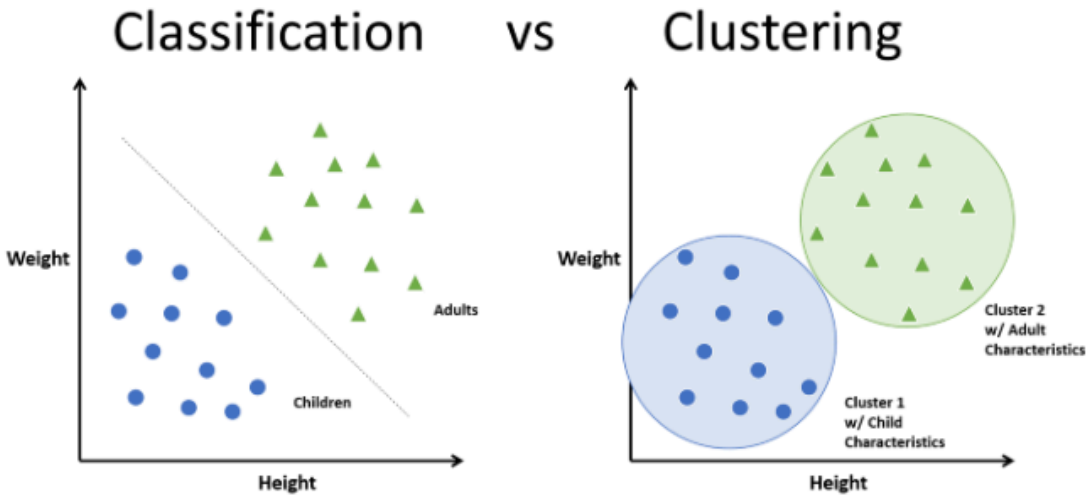
The method I used is called K-Nearest-Neighbors (KNN) and this is a very common technic in classification. I'm gonna explain the main goal of this technic, the algorithms I used and why, and the main results and conclusion I can take from this.

1 - What is classification / clustering ?

In Machine Learning, there is a whole topic in order to explain data that is called classification. The main purpose is to guess an information on data based on other parameters. For example, if you have data that contains different informations on real estate (for example surface, location, its exposition,…) and you want to guess the price range, you can use classification methods for that. You just need to have a few data that we know the real price range, and then you could guess price ranges for new data. "Simple as that". This is a **supervised** technic.

There is a quite important difference with clustering : you cluster data when **you don't know the information you want to guess in advance**, for our example it's the price range. Let's just imagine that this time, you have a dataset that contains only informations on different flats but we never know the real price range. You use clustering technics to "group" data with similarities. This is an **unsupervised** technic.

The following picture illustrates well this difference. We take the example of a dataset with humans physical informations and we want to guess if it's a child or an adult. On the left picture, we can assure that, based on weight and height we know if it's an adult or a child. On the right picture, all we did was to group individuals with similarities. Up to us now to determine if one group is the "children group" or the "adult group".



Determination of age group based on physical characteristics

That is the difference between these two technics. My comrades used clustering technics in order to group signs that show similarities, and what I did was a classification technic to try to guess what sign it is when you give the algorithm a new picture (this is a n image recognition technic and it's very useful now, if we take the sign recognition example it's the basis of self-driving cars for example).

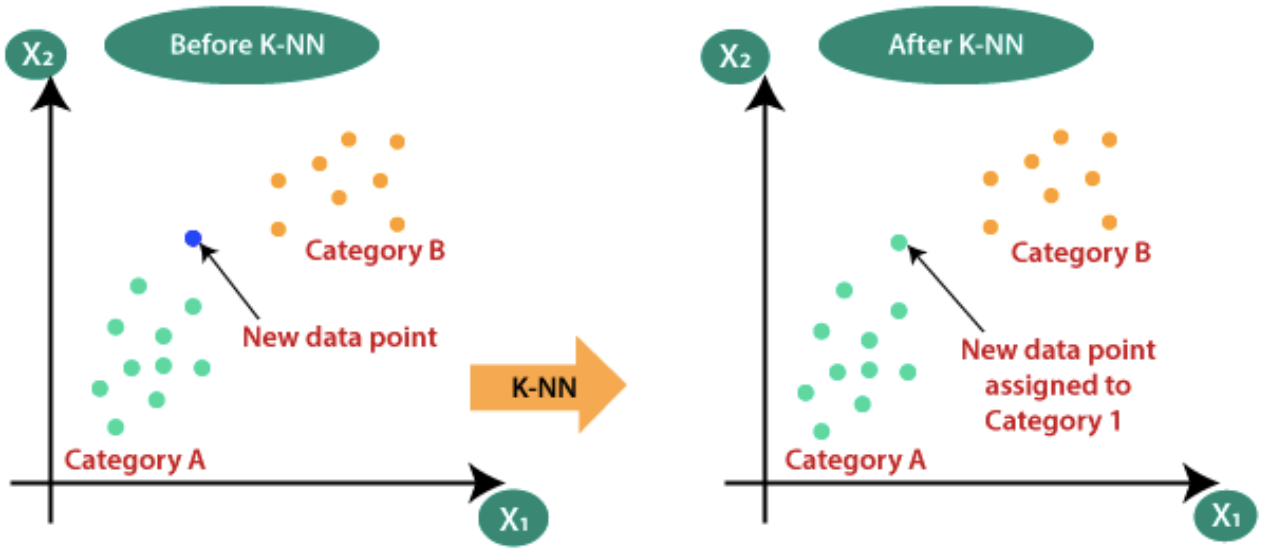
2 - KNN algorithm

2.1 - Principle

The main idea behind this technic is quite simple : let's have an intuitive example.

Look at the following picture and imagine we want to put a new data point on this map, by knowing only two characteristics on the two axes X1 and X2 (we can take the same example than before with height and weight values and try to guess if it's a child or an adult). We can place it, but what if we want to determine its category (if it's a green or an orange point) based on its characteristics ?

Well, the most "logical" way to determine its category is to look the nearest points and look their colour. In our example, if we look the 5-nearest neighbours, we have 3 that are green, and 2 that are orange. Then, after this computation, we assume that our point is a green one. We don't just look at the distance on a graph obviously, because we work with thousands of data that are not always in dimension-2, but there is the main idea summarised with this simple example.

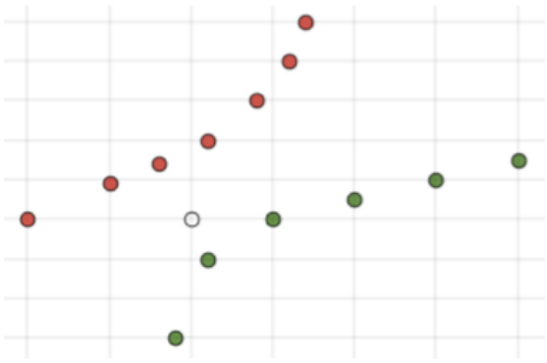


More scientifically talking, we compute the Euclidian-distance (this is the **Minkowski**-distance special case where our dimension is equal to 2, but let's keep that for our example) on our "new" point and others. Then, we compare the **k** first neighbours, with the smallest Euclidian distance, and then we can take a decision based on this number. But, this technic is not always 100% accurate, so a more optimised method exists, and this it what I used to begin my coding work.

$$\sqrt{\sum_{i=1}^{i=n} (x_i - y_i)^2}$$

2.2 - Weights

Look at the following picture. If we want to guess the group of the white point, and if we use K-NN for that, it will compute a red class, although the good answer is the green class. To avoid this, we can use a **weighted classification**, i.e we take into account the distance between our points in order to determine the most decisive points that will compute our result.



The thing we do is that we have a function (called kernel function) that determines our weight. A common use is to take the inverse of the distance. We take the k first neighbours (with the smallest distance) and we sum all that points multiplied by their weights, then we keep the "maximum" of this. The predicted class is defined by the following formula.

$$y' = \underset{v}{\operatorname{argmax}} \sum_{(\mathbf{x}_i, y_i) \in D_z} w_i \times I(v = y_i)$$

This method takes in account the distance of the points, it's more balanced, because it can be difficult to predict an outlier class for example. There is a function I wrote myself, but many other examples exist, for example on this page : https://www.geeksforgeeks.org/weighted-k-nn/

```
[ ]  def weighted_knn_classification(x_test, x_train, y_train,minlen,k=10):
       y_pred=pd.DataFrame(columns=["ClassId"])
       for i in x_test.index.to_list():
         dist=np.linalg.norm(x_train-x_test.loc[i],axis=1)
         indexs=dist.argsort()[:k]
         dist=np.sort(dist)[:k]
         weights=1./dist
         a=np.bincount(y_train.iloc[indexs],weights=weights,minlength=minlen)
         class_pred=a.argsort()[-1]
         y_pred.loc[i]=class_pred
       return y_pred
```

This function gave me the same results than the pre-built model available with **sklearn**, so we prefer using this model for the rest of our work as it is already implemented in Python.

3 - Standard K-NN

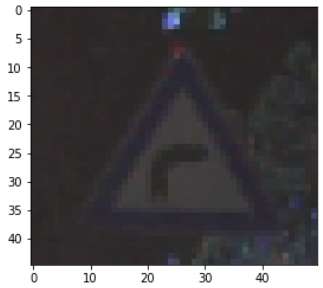
The goal of our project is to classify German traffic signs with a K-NN algorithm we saw before. For that, we have around 39.000 pictures of signs in order to train our model, then around 12.000 to test it (the classification part). The goal of our model is, when we enter a new sign, try to guess what sign it is.

But, how can we use K-NN with pictures, when we are supposed to compute distances with discret parameters ?

3.1 - Pixel informations

Actually, each picture can be transformed in a 3-dimension array : the elements of this array are the pixels values used to describe the picture :

```
<matplotlib.image.AxesImage at 0x7efdd7db0ee0>
```



The sign picture you see here is described by this array

```
array([[[45, 40, 39],
        [46, 41, 41],
        [47, 42, 43],
        ...,
        [42, 37, 38],
        [41, 37, 38],
        [40, 36, 38]],

       [[48, 42, 42],
        [46, 40, 40],
        [47, 42, 43],
        ...,
        [42, 37, 39],
        [43, 39, 41],
        [42, 38, 40]],

       [[46, 40, 42],
        [46, 40, 41],
        [46, 41, 42],
        ...,
        [42, 37, 40],
        [40, 35, 39],
        [39, 34, 37]],

       ...,

       [[46, 41, 44],
        [48, 42, 45],
        [46, 41, 44],
        ...,
        [62, 67, 68],
        [45, 48, 54],
        [47, 49, 52]],

       [[46, 42, 44],
        [47, 41, 43],
        [45, 40, 43],
        ...,
        [57, 58, 64],
        [48, 49, 57],
        [49, 51, 55]],
```

But, how can we use K-NN with a dimension 3 array ? We have to **flatten** our array for that, and resize it, in order to keep the informations in a simple vector. This function “compress” the pixel values in a single-dimension vector, but each picture is now a 3072-long vector instead of a *nxmx3* (the length *n* and width *m* can vary depending on the pictures).

The problem is that we will train a 39209 by 3072 array in our model : it can takes a lot of time to compute, because of the high-dimensional vectors. But the principle here is easy : compute the K-NN results with flattened pictures.

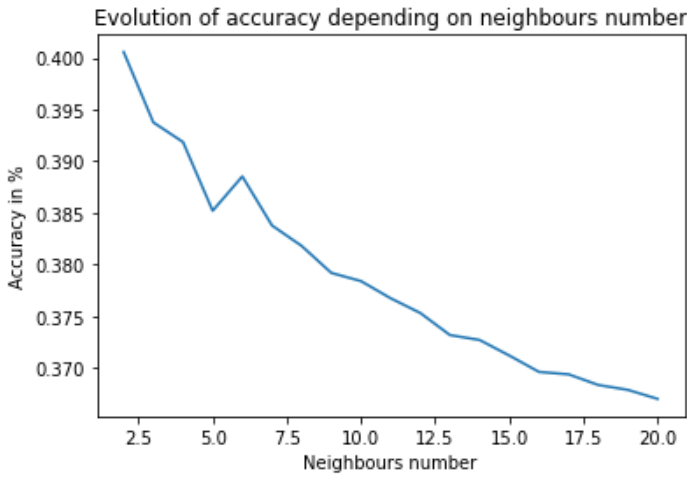
3.2 - Results

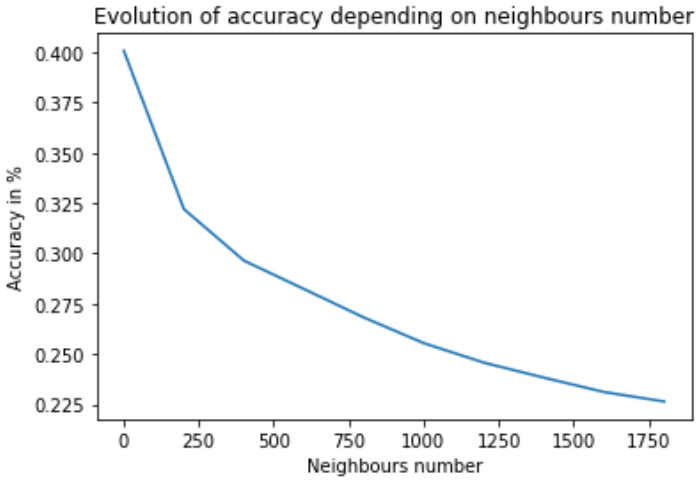
I put the results of this model in a classification report available on the code. It depends on the classes, some are well classified (for example sign classes **17** or **39** have a >90% accuracy), some are not really classified, with a ~0% accuracy for class **24** or 6% for class **36**.

The general accuracy on the test sample is around 37%, that's not really bad but not really good either.

accuracy			0.38	12630
macro avg	0.37	0.32	0.33	12630
weighted avg	0.41	0.38	0.38	12630

I used a **k=10** to start this exercice, and then tried to change this number to check if the accuracy is changing according to the number of neighbours. It's not really better, but we can reach a 40% accuracy with **k=2**. I tried to reach huge numbers of neighbours, but it's does not really improve the accuracy.





We can see that accuracy is decreasing while **k** is increasing. It seems logic : the more is **k**, the more chances you can meet another class that is not the good one to classify, especially if you have a huge **k** value.

Now, we have several things to study :

- Does **k** value really change something ?
- Why some classes are well classified and other not ?
- Is there a possibility to reduce our data dimension to improve our results ?

We will answer the second question after using other K-NN technics.

The last question is interesting and there is a common technic used in Machine Learning to reduce our data dimension : **Principal Component Analysis**

4 - PCA

4.1 - Principle

Principal Component Analysis (PCA) is a dimensionally reduction technic used to obtain a set of uncorrelated variables with less dimension. This could be really useful here, because the standard K-NN algorithm is not that powerful in high-dimension : https://www.baeldung.com/cs/k-nearest-neighbors**.** We project our data in a less-dimension space in order to represent it (sometimes in 2 or 3 dimension to have a precise idea of the data scattering, but the more we reduce our dimension, the less it will be "precise"). In this new space, we obtain new variables that are linear combinaisons of the previous ones.

Briefly, we start with a N x M matrix , where N is the number of observations (here more than 39.000) and M is the number of explicative (quantitative) variables (here around 3070). We need to **standardise** and **scale** our matrix values, in order to keep the same magnitude for all our data. Now that we have a "new" matrix, we calculate what is called **inertia** : this is roughly speaking the part of information (variance) we keep for each axe of our new space. Of course, four our 3070 space, we can't reduce it in 2 or 3 dimensions, we would loose too much variance :

$$I = \frac{1}{n}\sum_{i=1}^{n}\sum_{j=1}^{p}\left(\frac{x_{ij} - \bar{x}_j}{\sigma_j}\right)^2$$

Inertia formula

```
sum(pca.explained_variance_ratio_)
```

```
0.6253468092190573
```

Explained variance in 2-dimension

```
sum(pca.explained_variance_ratio_)
```

```
0.6677566966870827
```

Explained variance in 3-dimension

```
sum(pca.explained_variance_ratio_)
```

```
0.9190482124511636
```

Explained variance in 64-dimension

A 91% explained variance is a quite good portion of kept information, so we can try in a 64-dimension space. This of course way too much to project it for us, as wee can only visualise data in spaces of dimension 3 or less, but it could maybe be more efficient because we reduced a lot our dimension, and it could improve our results for K-NN.

4.2 - Results

Results are not that good. Doing a PCA is not improving that much our precision in the inference (testing) phase. It's even decreasing our precision. We obtain a 7% accuracy, what is even less than what we had with our previous "standard" K-NN algorithm. Actually, this could be explained because we "bring our data closer" by using PCA, as we reduce their absolute value. It could be an explanation of ur poor results. Check below the transformation of our values by showing the difference between the first 5 pixel values or the first picture of both our train and test set :

```
[85] x_train_pca[0,0:5,]

     array([-19.20876989,   3.70651435,   5.04274275,  -3.51974612,
             3.96467569])
```

```
[84] x_train[0,0:5]

     array([56, 51, 46, 56, 51], dtype=uint8)
```

Train set value changes

```
[86] x_test_pca[0,0:5]

     array([ 21.68520383,  20.28246124, -12.56508012,  -2.37343284,
            -11.49861409])
```

```
[87] x_test[0,0:5]

     array([174, 139, 116, 171, 137], dtype=uint8)
```

Test set value changes

Using PCA was not a bad idea in theory, because it would have reduced our data dimension, but the fact K-NN is using distance between our data and that we reduce this distance makes our algorithm less efficient because it “false” our analysis, as it bring closer some data that are not from the same class.

5 - Semi-Supervised K-NN

5.1 - Principle

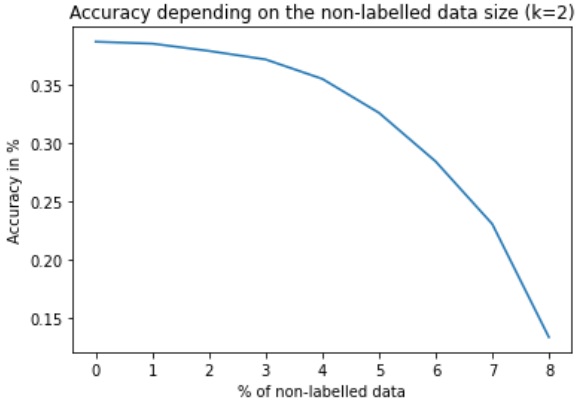
Semi-Supervised Learning is a combination of Supervised and Unsupervised Learning. The principle is really easy to understand : if we have all our class labels, we can get rid of some to focus on the data similarities and try to analyse our data in that sense. We use a technic that is called Label Propagation, or Label Spreading, that is more detailed in that paper : https://arxiv.org/pdf/1904.04717.pdf, or this one : http://www.jsoftware.us/vol8/jsw0804-02.pdf

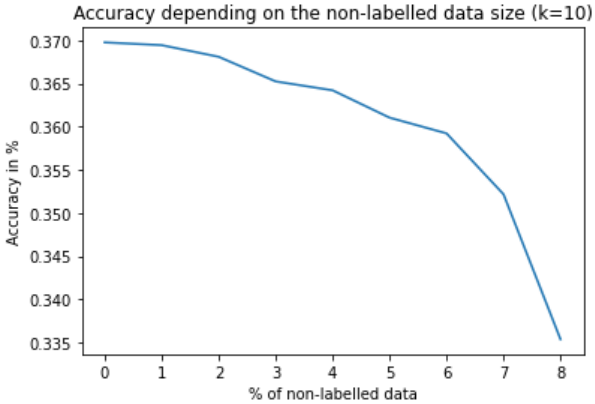
This article also explains the principle easily : https://towardsdatascience.com/how-to-get-away-with-few-labels-label-propagation-f891782ada5c

In practise, we replace labels of a certain percentage of our data by the **-1** value (a new class that indicates an absence of label). We then use the **Label Spreading** model, with a K-NN kernel to classify our pictures.

5.2 - Results

I used a long procedure to check if there is some important changes on our accuracy depending on the non-labelled data size. I only checked it with two K-NN models, with a k=2 and a k=10, and there are our results summarised on these two plots :





6 - Deep Learning K-NN

6.1 - Principle

For this part, I decided to try a combination of a Neural Network model and the K-NN principle. I read some really interesting papers that I quote here :

- https://arxiv.org/pdf/1803.04765.pdf
- https://www.researchgate.net/publication/345239222_Deep_kNN_for_Medical_Image_Classification
- http://www.cleverhans.io/security/2019/05/20/dknn.html

I also found a paper about a loss function that I could try on my Neural Network model :

- https://arxiv.org/pdf/1902.01889.pdf

But, the most interesting model in my opinion was a combination between the **ResNet50** famous Deep Learning image recognition model and a K-NN model. I found some really interesting explanations here about how to combine those two models : https://medium.com/@sorenlind/nearest-neighbors-with-keras-and-coreml-755e76fedf36

I'm not going to detail how a neural network is working and all the functions used for, that was a whole part of our classes, and the **ResNet50** is enough know and proved that it can be very efficient on image recognition.

6.2 - Join models

The main part of this model is to combine a multi-layer neural network with a method that can compute same way as K-NN algorithm. The creation of our "big" model is explained with this function :

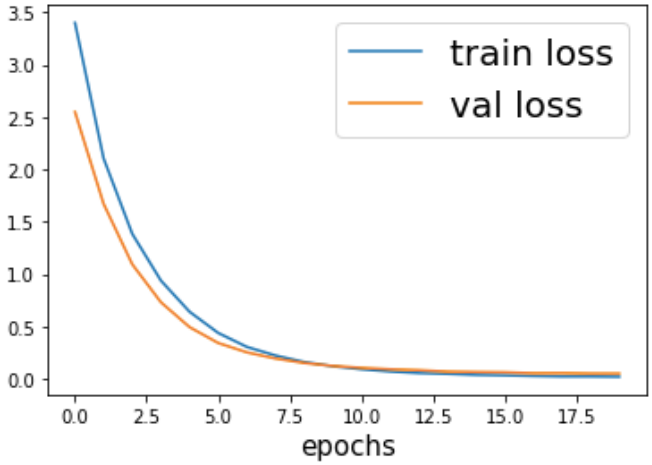
```
def build_knn(model, output_size):

    flat_dim_size = np.prod(model.output_shape[1:])
    x = Reshape(target_shape=(flat_dim_size,),
                name='features_flat')(model.output)

    x = Dense(units=output_size,
              activation='linear',
              name='dense_1',
              use_bias=False)(x)

    classifier = Model(inputs=[model.input], outputs=x)
    return classifier

joined_model = build_knn(encoder_model, 43)
joined_model.summary()
```

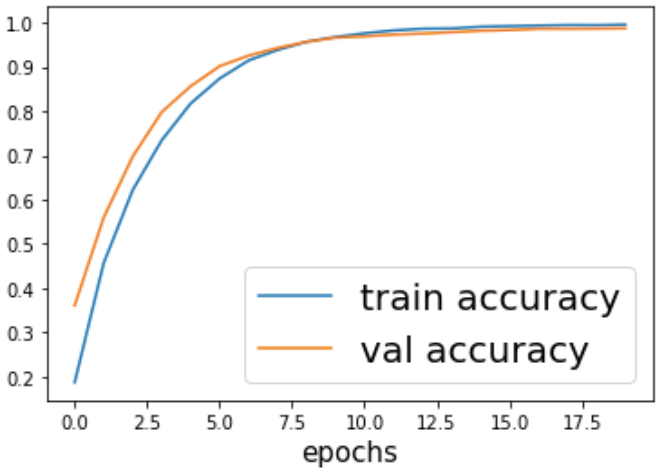
We have to flatten our pictures in order to compute in 1-dimension as we saw previously with the "standard" K-NN algorithm. We add a **Dense** layer that is a "simple" dot product between feature vector and reference vector in order to give a probability as our K-NN classifier. The output is so 43, our number of different pictures classes.

6.3 - Results

6.3.1 - ResNet50 results

When we just compile the **ResNet50** model without our final **Dense-KNN** layer at the end, we obtain very good results (I tried with 20 epochs, the convergence is not improving that much after) :





And moreover we get around a 71% accuracy, what is really much better than our previous models.

But, is it still the case when we combine it with our **Dense K-NN** model ?

6.3.2 - Joined model results

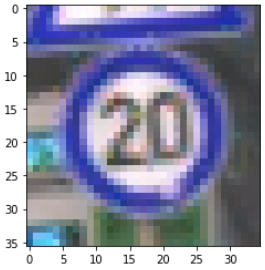
Unfortunately, when we combine the two models, our accuracy decrease sharply. Lot of classes are not classified properly, and only **2%** of them are. Maybe doing too much modifications on our pictures makes them “all look alike” and it becomes difficult to classify. Maybe we also **overfit** and not learn tour data he good way.

7 - Conclusions

7.1 - Why such deviations ?

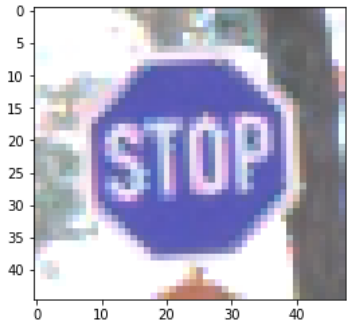
We saw, with the **classification report** function, that lot of classes are really well classified, like the numbers **14**, **17** or **39**. Conversely, the classes **24**, **36** or **0**, for examples, have not been well classified.

My opinion is that these classes represent signs that really look like others. For example, the class 0 represent this kind of signs : (I put some other examples in my **.ipnyb** file)



But, there exist 5 or 6 signs that really look like this for speed limits. We can do a comparison between our model, that learn with pixel data, which code the pictures and consequently look similar for similar pictures, and with a human learning. If you give this kind of picture to a 4-years old child and that he has to classify all the pictures, he could confuse the speed limit signs easily. Same things for our other misclassified signs. Conversely, if you give him a sign that is “unique” and don’t look like other ones, he will classify it very well. That is the case for the class **14** that represent “STOP” signs, what is difficult to confuse with others.

```
<matplotlib.image.AxesImage at 0x7f4c78358430>
```



7.2 - Conclusion

That work was really interesting to do. My accuracies are not always good, but K-NN is not really powerful in high-dimensional spaces. I really like the idea to separate some pictures just with an algorithm, I think as I said above that this kind of algorithms are the future, the basis of self-driving cars, automatic detections of abnormalities, and it has a lot of other capacities. I really want to continue in that way and do other algorithms and models like that. K-NN was a good idea to train, but definitely not the best one in my opinion. My comrades tried other models and that was really interesting to have the comparison between Supervised, Semi-Supervised, Self-Supervised and Unsupervised Learning.

To go further, I read a paper about a more powerful Deep-KNN Pytorch model, but I did not had time to implement it as my way : https://github.com/bam098/deep_knn

Thanks for your reading :)

Sacha BESSON