Alex Chagan Python Version:3.8

Machine Learning Final Project
Image Classification

Introduction

This project is the final part of the Machine Learning course in the University of Ariel.
I decided to make a project about Image Classification by implementing 4 machine learning methods: K-nearest neighbors, Decision trees, Support Vector Machine and Multi-Layer Perceptron. For each method, we will see the results and try to answer some questions:
Which methods worked the best?
What made them successful and why other methods were not?
What were the difficulties with implementing and testing each method?
Can we change the image size for better results?
We will also discuss various topics like: Efficiency, Training and Testing optimal ratio and Overfitting.

Dataset

This dataset contains over 9000 handwritten digits and arithmetic operators.
Total no of classes: 16, **Digits**: 0 1 2 3 4 5 6 7 8 9
Operators: Plus Minus Multiplication Division Decimal Equals
Most images are of resolution 400x400 pixels. Some may be 155x155.
All of the Images are resized to the same resolution for the training and the testing of the methods.
Here's the distribution of classes in the dataset:
{0: 595, 1: 562, 2: 433, 3: 541, 4: 526, 5: 433, 6: 581, 7: 533, 8: 554, 9: 546, 10: 596, 11: 624, 12: 618, 13: 634, 14: 577, 15: 655} (10=add, 11 =dec ,12=div ,13=eq ,14=mul 15=sub)
We will try to see how the class distribution affects each method.

Data preparation

I converted the images to 2D matrices with the help of the "CV" library where every value represents the intensity of a pixel.
At first I decided to resize all images to 100x100 for no particular reason, just to have something to start with and then I can adjust based on quality of results and runtime.
Next, I converted all images to grayscale because color has no impact on classifying handwritten symbols and also I had some classes that weren't represented numerically so I had to encode them with the help of a label encoder.
In addition, I had to change the 2D matrices into 1D arrays for the data to work with the "sklearn" library which contains our models.

Some examples:

Methods

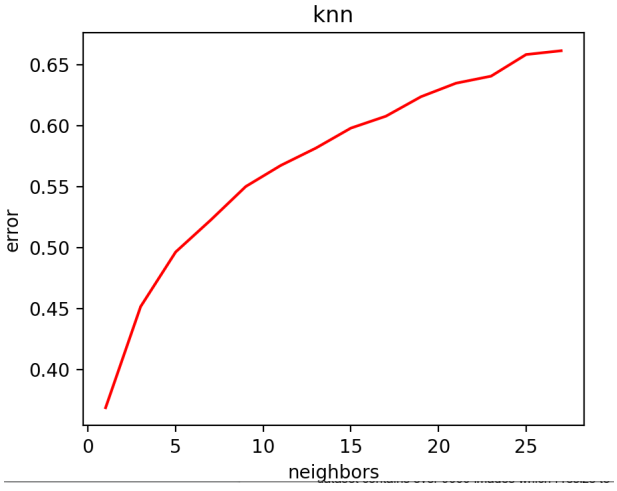
1. K nearest neighbors

The k-NN algorithm classifies unknown data points by finding the *most common class* among the *k-closest examples.* Each data point in the *k* closest examples casts a vote and the category with the most votes wins. The training data is the set of points that exist in the space and the testing data is the set of points that we want to find classification for by calculating the distance with each existing point.

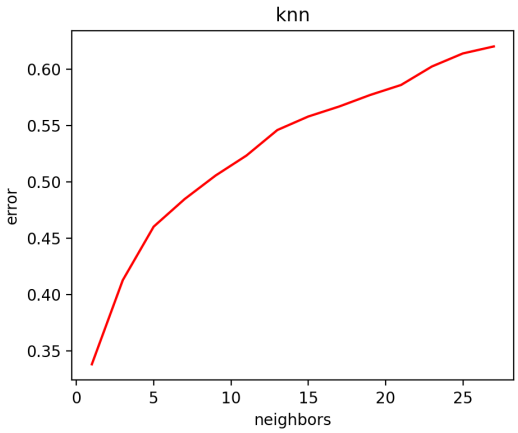
- The first obvious problem with this algorithm is that it's very slow with images, my dataset contains over 9000 images which I resize to 100x100 resolution and each pixel in each image is an independent feature. So it has to iterate over 200000 values in order to compare 2 images and calculate the euclidean distance between each feature in both images.

- The second problem is that small variations in the data might affect the distance calculation and give an inaccurate result.

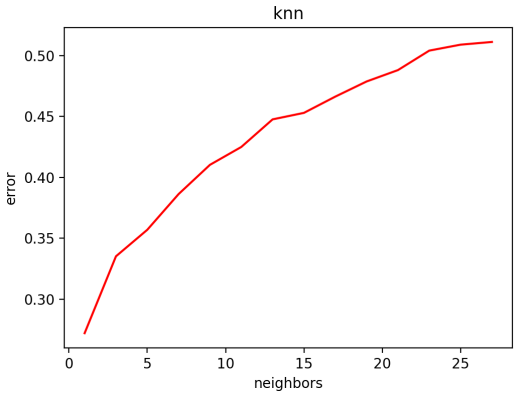
These are the results for 1-29 (odd numbers only) neighbors and 100x100 sized images:



What if we will try to make it less slow by reducing the size?
Results for 50x50 sized images:



We can see that the results are better and we spared some runtime
Let's try to make them even smaller:
25x25:



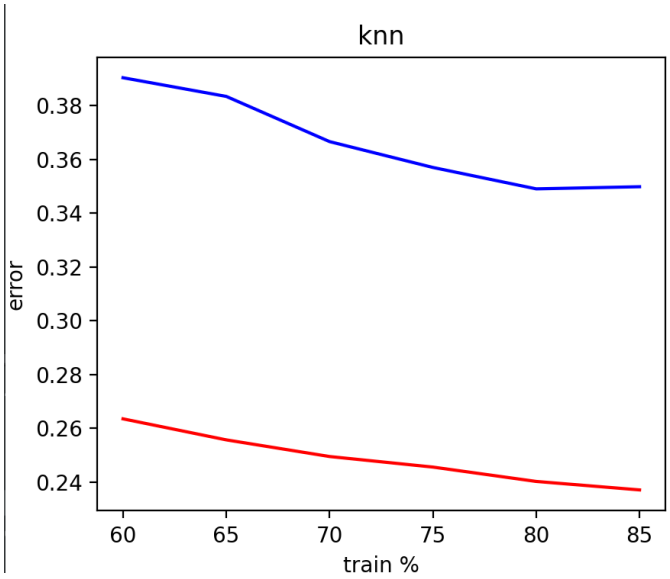
Even better results. Why is that?
It makes sense why the results aren't worse. Because images of handwritten numbers and math symbols are not complex and have very simple patterns, so even if we lose some pixels, the data is still very clear.
But why are the results better?
Because we remove high-frequency information when we reduce the size and because our images contain very simple patterns like lines and shapes, we get better results.

Here's a report of results for each class for k=15 and 25x25 size images:

	precision	recall	f1-score	support
0	0.64	0.38	0.47	161
1	0.25	0.81	0.38	134
2	0.88	0.40	0.55	114
3	0.75	0.42	0.54	129
4	0.84	0.33	0.47	125
5	0.74	0.35	0.47	120
6	0.68	0.55	0.61	145
7	0.45	0.33	0.38	129
8	0.94	0.33	0.49	135
9	0.66	0.24	0.35	129
add	0.85	0.66	0.74	150
dec	0.39	0.99	0.56	163
div	0.93	0.57	0.71	143
eq	0.90	0.56	0.69	163
mul	0.90	0.73	0.80	142
sub	0.42	0.85	0.56	170
accuracy			0.55	2252
macro avg	0.70	0.53	0.55	2252
weighted avg	0.69	0.55	0.56	2252

We can see that the amount of samples for each class don't affect the results very much.

What about train and test distribution? (25x25 images and 5 neighbors)



(Blue = test error, Red = train error)

We can see that the results are slightly better with more training data.

More neighbors increase error. Why?
I came to the conclusion that it's because of the small variations in data of the same classes that cause different distance calculations and as a result we get a lot of uncertainty. More neighbors means: more miscalculated distances to choose from.

Most of the mismatching happens because our current model uses euclidean distance to choose neighbors, by comparing each same pixel in both images and as a result, the mismatches happen because of those small variations in data we talked about.

Question: Can we reduce the uncertainty by implementing a different distance measurement? There is such a measurement and it's called "Cosine Similarity". It's basically a measurement that finds similarity in two or more vectors. The cosine similarity is the cosine of the angle between vectors. The vectors are typically non-zero and are within an inner product space.
In other words, it can detect similar patterns in two sets of data.

$$\text{similarity} = \cos(\theta) = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\|\|\mathbf{B}\|} = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2}\sqrt{\sum_{i=1}^{n} B_i^2}},$$

Unfortunately, the cosine similarity is not a distance metric so I couldn't implement it in the scikit-learn KNeighborsClassifier model.

Conclusions:

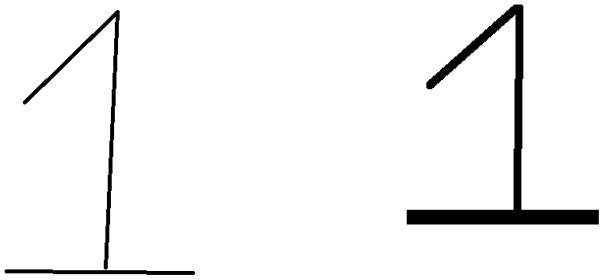
- Smaller images = better results
- More neighbors = worse results
- More training samples = worse results
- Results are not optimal because of variation in data of samples from the same class when using euclidean distance. It can be optimised using methods like "Cosine Similarity".
- The method is relatively slow because we have a big dataset and images contain a lot of features.

2. Decision Trees

The Decision Trees algorithm breaks down a dataset into smaller and smaller subsets based on it's features while at the same time an associated decision tree is incrementally developed. The final result is a tree with decision nodes and leaf nodes. A decision node has two or more branches . Leaf node represents a classification or decision.

Decision Trees algorithm is one of the less effective methods for image classification because of the instability of the model with complicated datasets.

- Like KNN, small variations in the data might result in a completely different tree being generated. For example if we train an image of the number "1" and then test another image of "1" but the size and placement are slightly different.

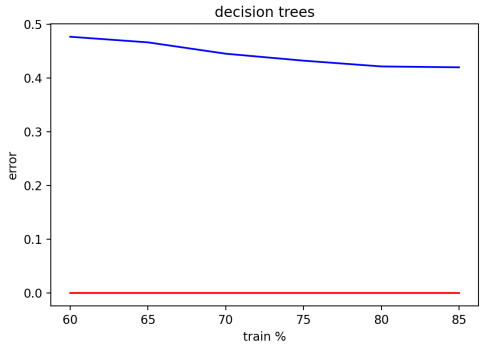


These 2 images can generate different decision trees and cause a wrong classification.

- Because the features in our dataset are pixels, decision-tree learners can create over-complex trees that do not generalize the data well and cause overfitting.

From the previous method we have seen that it's more efficient to work with smaller images with our dataset.
Just like in KNN, the reason for that is because our images are very simple and we get less complicated decision trees with smaller images and less features.

Results with 25x25 images based on train and test distribution:



(Blue = test error, Red = train error)

Report of results for 80% train 20% test with 25x25 images:

	precision	recall	f1-score	support
0	0.47	0.36	0.41	127
1	0.65	0.81	0.72	107
2	0.47	0.47	0.47	90
3	0.46	0.35	0.40	111
4	0.34	0.32	0.33	98
5	0.29	0.31	0.30	87
6	0.47	0.41	0.44	120
7	0.54	0.64	0.58	106
8	0.35	0.37	0.36	108
9	0.45	0.44	0.45	102
add	0.60	0.67	0.63	117
dec	0.86	0.90	0.88	132
div	0.65	0.65	0.65	115
eq	0.71	0.69	0.70	135
mul	0.71	0.68	0.69	110
sub	0.79	0.85	0.82	137
accuracy			0.57	1802
macro avg	0.55	0.56	0.55	1802
weighted avg	0.57	0.57	0.57	1802

Is there an advantage for classes with more samples?
When classes contain a lot of features that interact with each other, it is rather demanding for a tree to recognise the right pattern for a minority class. And even if it does, it will probably be a deep and unstable tree that will cause overfitting

Let's check by undersampling the training data:

```
# define undersample strategy
undersample = RandomUnderSampler(random_state=42)
# fit and apply the transform
X_over, y_over = undersample.fit_resample(X, y)
return X_over, y_over
```

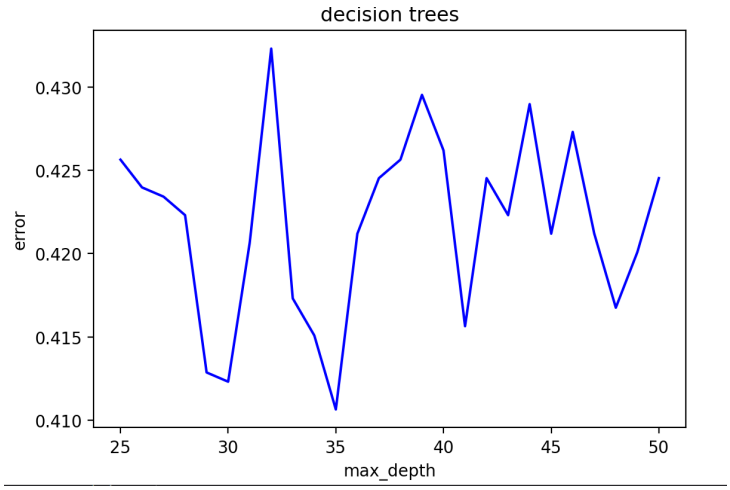
In this example we got 343 samples for each class:

	precision	recall	f1-score	support
0	0.46	0.35	0.40	127
1	0.60	0.67	0.63	107
2	0.45	0.48	0.46	90
3	0.41	0.36	0.38	111
4	0.30	0.30	0.30	98
5	0.38	0.38	0.38	87
6	0.41	0.39	0.40	120
7	0.50	0.55	0.52	106
8	0.35	0.32	0.34	108
9	0.37	0.38	0.38	102
add	0.61	0.65	0.63	117
dec	0.85	0.89	0.87	132
div	0.59	0.63	0.61	115
eq	0.65	0.61	0.63	135
mul	0.66	0.67	0.67	110
sub	0.74	0.80	0.77	137
accuracy			0.54	1802
macro avg	0.52	0.53	0.52	1802
weighted avg	0.53	0.54	0.54	1802

We can see that the results remain relatively the same, which means that there is no bias towards certain classes.

Is there a way to make the trees less unstable?

We can try to reduce the maximum depth of the trees and make them less complex.
But will it give us better results?
25x25, 80% train, 20% test:



We can see that at max_depth=35 we get the best results

Conclusions:

- Smaller images = better results
- More training samples = better results
- Small variations in samples of the same class can cause generation of different trees in training and can cause mismatching in testing.
- Reducing the complexity of the trees by reducing their maximum depth, improves the results.
- There is Overfitting
- Fast

3. Support Vector Machine

SVM is a method in which we plot each data item as a point in n-dimensional space (where n is the number of features we have) with the value of each feature being the value of a particular coordinate. Then, we perform classification by finding the hyperplane that differentiates the two classes very well and allows misclassifications.

The problem is that natively, SVM is a method that is used for binary classifications because it only creates one hyperplane.
In order for SVM to work on a multiclass dataset, we need to create hyperplanes between every two classes.

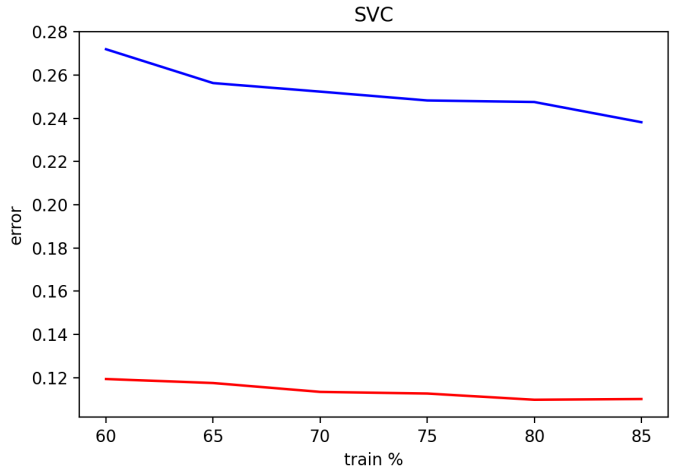
Luckily, the sklearn.svm library supports multiclass classification:

```
model = SVC(decision_function_shape='ovr')
```

ovr = one-vs-rest

How does it translate to our dataset?
Our dataset contains a total of 16 image classes and we have already seen from previous methods that we can reduce the size of the images. So for this example we will work with 25x25 images which means that we are going to create 625-dimensional hyperplanes between each two classes.

Here are results based on train and test distribution:



(Blue = test error, Red = train error)

We can see that 85% train 15% test, gives us the best results.

Let’s try to undersample the training data.
Will it impact the results?

	precision	recall	f1-score	support
0	0.74	0.68	0.71	63
1	0.65	0.83	0.73	59
2	0.74	0.74	0.74	43
3	0.76	0.63	0.69	59
4	0.68	0.55	0.61	58
5	0.74	0.73	0.74	48
6	0.57	0.69	0.63	67
7	0.62	0.84	0.71	44
8	0.71	0.68	0.69	50
9	0.56	0.65	0.60	43
add	0.85	0.86	0.85	51
dec	0.83	0.96	0.89	67
div	0.92	0.76	0.83	62
eq	0.92	0.70	0.80	67
mul	0.98	0.73	0.83	55
sub	0.76	0.85	0.80	65
accuracy			0.74	901
macro avg	0.75	0.74	0.74	901
weighted avg	0.76	0.74	0.74	901

It doesn’t.
Which means that our dataset is relatively balanced.

Conclusion:

- Relatively good results
  - More training samples = better results
  - SVM gives much better results compared to the previous methods.
  - There is Overfitting
- Slow

4. Multi-Layer Perceptron

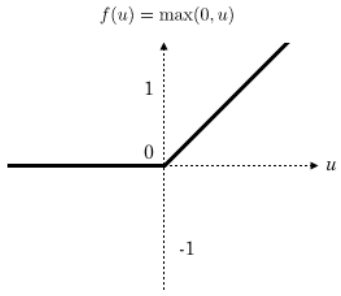
MLP is a feedforward artificial neural network method that trains data by using Backpropagation for updating weights. Backpropagation computes the gradient of the loss function with respect to the weights of the network for a single input–output example
The model consists of three types layers which contain neurons:
Input layer: This is the layer that contains the data of our sample, each neuron represents a single feature.
Hidden layers: These layers are responsible for taking the inputs from the previous layers, multiplying them by certain weights and feedforward the outputs based on an activation function.
Output layer: The final layer that makes the classification based on the calculations from the previous layers. Each neuron represents a class in the dataset.
The weights in each layer are updated after every iteration based on the error.

For my dataset I need 625 neurons in the input layer (25x25 images) and 16 neurons in the output layer (16 classes).

But after understanding the model, there are still some unknowns.

How many hidden layers do I need?
How many neurons do I need in each hidden layer?
Which activation function should I use?
How many iterations do I need to get optimal weights?

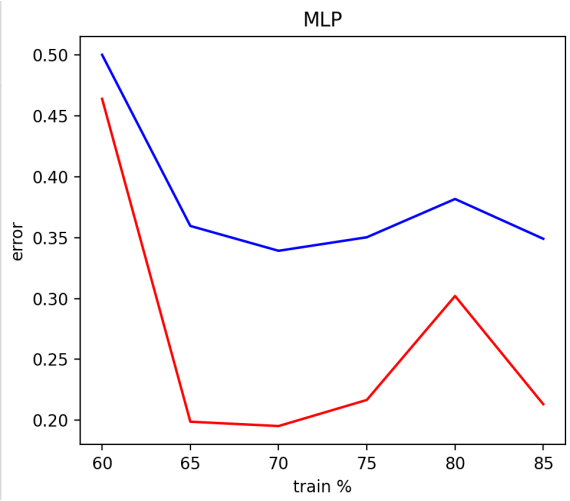
After doing some research and testing I decided to go with the ReLU activation function. The main advantage of using the ReLU function over other activation functions is that it does not activate all the neurons at the same time. For the negative input values, the result is zero, that means the neuron does not get activated. Since only a certain number of neurons are activated, the ReLU function is far more computationally efficient when compared to the sigmoid and tanh function.



As for the number of hidden layers and number of neurons in each layer, there isn't a clear answer for the right amounts and some experimentation is needed. Since my data set is quite big and there are alot of features and classes, the algorithm runs pretty slow, so experimenting with different values would have taken a lot of time. I decided to go with the default values of the sklearn's MLPClassifier which is: 2 hidden layers and 100 neurons.

As for the number of iterations, I kept increasing the number until the model converged and landed on 500 iterations.

These are the results for 25x25 images:



We can see that there's a sweet spot at 70% training and 30% testing.

Conclusion:

- Relatively good results
- Simple model but requires a lot of experimentation in order to get optimal results
- There is Overfitting

Why did some methods work better than others?

I think it's because methods like KNN and Decision Trees are very sensitive to small variances in data where KNN algorithm can compute very different distances based on the variance and Decision Trees algorithm can compute an inaccurate tree based on variance.
But most importantly, unlike KNN and Decision Trees, algorithms like Multi Layer Perceptron and Support Vector Machine allow misclassification in order to make more accurate classification boundaries in the training phase, which means that they can miss classify some samples when they build the model with the training samples but make more accurate classifications in general with the testing samples.

Libraries used:
Os
Cv2
Sklearn
matplotlib