Support Vector Machine (SVC) for Classification of Handwritten Digits

Daniel Gargiullo
CUNY Queens College
daniel.gargiullo15@qmail.cuny.edu

Abstract

This paper focuses primarily on the Support Vector Machine model (SVC), and evaluates it, along with several classifiers from the `scikit-learn` Python library (Gaussian Naive Bayes, K-Nearest Neighbors, Logistic Regression, `scikit-learn`'s SVC model, Decision Tree, Random Decision Forest, and Multi-Level Perceptron) on a normalized 16×16 grayscale dataset of hand-written digits, by analyzing and comparing their classification results. These digits are from the USPS Machine Learning Dataset (training and test sets), and have been normalized by the folks at the U.S. Postal Service. Additionally, the program allows the user to test their own handwritten digits by first using image pre-processing functions in `preprocess.py` and then testing them on the model in `main.py`. As well, a small handful of images of digits (some taken from the web, some my own bad handwriting) have been provided in a simple, organized directory structure within the project.

1 Introduction

Image classification is a fundamental feature of digital image processing and computer vision, and has revolutionized how we use and interpret data across a wide range of fields and technologies, such as healthcare, self-driving cars, security, etc. In the past, the classification of images relied mostly on manual (i.e done by a human) feature extraction and heuristic-based ("good enough" in practice, likely not so good in theory) algorithms. These methods were labor-intensive and clearly not scalable. However in the 20^{th} and 21^{st} centuries came the advent of machine learning techniques for digital image processing, and with it significant advancements in accuracy, speed, and efficiency in complex pattern recognition of visual data.

At its core, machine learning involves algorithms that "learn", iteratively, from data, to perform image classification or image recognition without explicit programming instructions. The program in this paper spotlights my own coded SVC Model to classify handwritten digits based on pixel intensity values, but also calls upon the models provided by `scikit-learn` in order to compare the classification results of all the models. And while I have incorporated the Multi-Level Perceptron from the `scikit-learn` library, this is a feed-forward network and isn't technically considered a "Deep Learning" model, but I was curious to see how it fared against the other models in terms of accuracy and performance, so I included it. I've also kept the number of layers for the MLP model low. The objective of this paper is to provide a thorough analysis of the SVC model, evaluating and comparing its accuracy and performance on the dataset provided by USPS with that of the other models. Please note that all occurrences of the `self` keyword have been removed from all code snippets in this document for sake of brevity.

2 Methodology

To gain an understanding of how the Support Vector Machine model works, let's start with an arbitrary example. In Figure 1, we have two classes (black dots and white dots), and three hyperplanes (H_1, H_2, and H_3) to choose from to separate these two classes. We can see that H_1 (green) does not fully separate the two classes. H_2 (blue) fully separates the two, but with a very narrow margin. H_3 (red) fully separates the two classes with the widest margin of the three hyperplanes, and is therefore our optimal choice in this example configuration. This is just a visual example with arbitrary points and hyperplanes, but it represents the underlying idea behind the fitting of the model.

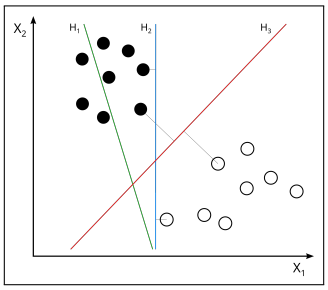


Figure 1: H_3 is the optimal hyperplane here.

2.1 SVMClassifier class

The class for the SVC model that I've coded (`SVMClassifier`) can be found at the top of `classify.py`. Defined within it are several class variables, the first three of which are initialized by parameters of names identical to those of the class variables themselves (i.e. `self.learning_rate = learning_rate`) for consistency:

- `regularization_rate`: Must be strictly > 0. Regularization strength (not explicitly defined in the program) is inversely proportional to the regularization rate:

$$regularization\_strength = \frac{1}{\texttt{regularization\_rate}}.$$

  For our purposes, we have tuned the `regularization_rate` to ≈ 10.5.

- `learning_rate`: The learning rate is the size of the update step, and by default is `1e-3`, or

$$1 \times 10^{-3}.$$

  With each step, the learning rate is subtracted from the weight gradient.

- `number_of_epochs`: An epoch is a single gradient-update step on the whole dataset. The margins are calculated, and by doing so we find what points are misclassified in the training set.

- Additionally, two class variables, `weights` and `biases` are initialized to empty lists to be used by the `SVMClassifier` functions.

2.2 Fitting the model

In the `SVMClassifier` class is the `fit(X, y)` function, which is the function to be called when training the SVC model on the dataset. In this section, we'll use x to denote parameter `X`, y to denote parameter `y`, $\mathbf{w}$ to denote our `weight_vector`, and b our bias.

The function begins by extracting the classes (digits 0 through 9) and initializing a vector called `weights` to a zero-matrix of ten rows (one row for each class) and 256 columns (one for each of the 256 values to be updated) to create a 2-D matrix, and a `biases` vector to store the biases that are to be calculated and updated with each epoch.

```
        classes     = np.unique(y)
        weights     = np.zeros((len(classes), X.shape[1]))
        biases      = np.zeros(len(classes))
```

Parameter y is the original 1-D set of integer labels (each representing a class), and for each class, we declare a variable `y_binary` that gets set to 1 if the current y is the same as the `class_label` for the current iteration, and -1 otherwise. We then create the initial 1-D array called `weight_vector` for that class, setting every weight to 0. We also initialize the bias for the current class to 0.0.

```
        y_binary        = np.where(y==class_label, 1, -1)
        weight_vector   = np.zeros(X.shape[1])
        bias            = 0.0
```

2.2.1 Computing margins, weight gradient, and bias gradient

We have a set of input vectors, x. For every class j, a weight vector $\mathbf{w}_j$ and a bias b_j are trained. The boundary of that binary classifier is defined as the hyperplane where $\mathbf{w}_j x + b_j = 0$. Points on one side of the boundary (i.e. where $\mathbf{w}_j x + b_j > 0$) are classified as being in class j, while points on the other (i.e. < 0) are classified as being not in class j. The weight gradient is obtained when `np.dot(...)` sums up $y_i x_i$ over all examples whose margin values are < 1.

```
        margins         = y_binary * (X.dot(weight_vector) + bias)
        misclassified   = margins < 1
```

Multiplying by the regularization rate gets us what is known as the *hinge-loss gradient*, which we then subtract from the weight vector, obtaining our full weight gradient.

```
    weight_gradient = weight_vector - regularization_rate*np.dot(
        y_binary[misclassified], X[misclassified])
```

Summing y_i over all "margin-violating" examples and then multiplying by the negative of the regularization rate, we get our bias gradient.

```
bias_gradient   = -regularization_rate*np.sum(y_binary[misclassified])
```

2.2.2 Updating the weight vector and bias

The weight vector **w** is moved by a factor of the learning rate against the weight gradient with each epoch.

```
    weight_vector   -= learning_rate*weight_gradient
```

As well our bias, b, steps against its gradient by a factor of the `learning_rate`.

```
    bias            -= learning_rate*bias_gradient
```

Our `weights` matrix and `bias` vector are then updated for this iteration's class, and the loop either repeats or terminates, concluding the fitting process of our SVC model.

```
            weights[class_index]    = weight_vector
            biases[class_index]     = bias
```

3 Implementation and experimentation

All the models in the `scikit-learn` toolkit are similarly structured in terms of their parameters, making it extremely convenient when comparing the performances of multiple classification models. For this implementation, I referred to `scikit`'s GitHub source code for their SVC class, and tailored my `fit(...)` function to suit the needs of this project. In this section we discuss the structure of the dataset and evaluate the performance of these models against each other.

3.1 Dataset structure

The structure of the dataset is a simple 2-D matrix, with the first column being the class labels, and the remaining 256 columns being the pixel values of the normalized grayscale (between $[-1, 1]$) 16×16 image of the handwritten digits. In short, each full row — meaning all 257 elements (the class label + the normalized 256 pixel values) — represents a labeled image. Figure 2 gives a visual representation of a small sample of the set with arbitrary values, with the class labels shown in red and the pixel values in blue.

…	…	…	…	…
4.0000	-1.0000	-0.9480	-0.5610	…
7.0000	-0.7490	0.1370	-0.3710	…
0.0000	1.0000	-1.0000	-0.8719	…
1.0000	0.1648	1.0000	0.6302	…
5.0000	0.0737	-0.9006	0.2750	…
7.0000	-0.7910	0.2175	-0.3443	…
…	…	…	…	…

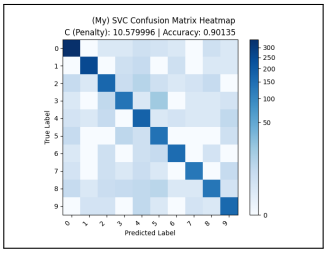
Figure 2: Dataset structure visualization.

3.2 Analysis and comparison of models

We evaluate the performance of the model primarily based on its accuracy results, looking at the deterministic and non-deterministic models separately, and then comparing the accuracy, runtime and memory usage results of all models at the end of the program. For the comparison, the deterministic models run only once, but the non-deterministic models, which use randomized seeds, run `n` times, with our `n` being set to 1, after which we take the average results (for our purposes, I've set `n` to 100).

Looking in `main.py`, one will see that each model is initialized by calling the function with its corresponding name (i.e. `model_gnb` calls the `gaussian_naive_bayes` function), which fits and returns the classifier.

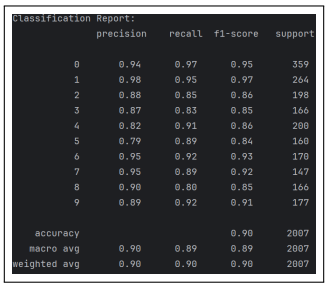
Each classifier, when trained, allows for a boolean parameter to be passed, which determines whether or not to display the heatmap visualization of that model's confusion matrix (the confusion matrix is printed to the output file `results.txt` in either case). Figure 3 shows the confusion matrix for my SVC model. The accuracy is represented by the diagonal of the matrix— the greater the difference in values outside the diagonal from those values on the diagonal, the more accurate the model. The heatmap aims to represent this with color intensities, but this same matrix can also be found in text form (for this model and all other models) in the `results` file.




```
==== (My) Support Vector Machine ====
Confusion Matrix:
[[347   0   1   1   3   2   1   0   3   1]
 [  0 251   0   3   5   0   3   0   1   1]
 [  5   1 168   4   9   3   1   2   5   0]
 [  1   0   6 137   1  17   0   1   1   2]
 [  2   1   5   0 181   1   2   1   1   6]
 [  5   0   0   7   3 142   0   0   0   3]
 [  1   0   3   0   3   5 156   0   2   0]
 [  2   0   3   1   4   1   0 131   0   5]
 [  5   1   4   5   6   8   1   1 133   2]
 [  0   2   2   0   6   1   0   2   1 163]]
```

Figure 3: Confusion matrix for my SVC model in heatmap and text form.

Analyzing the classification report in Figure 4 (also found in the `results` file) for the model, we can see that the overall accuracy (as shown by the confusion matrix) is roughly 90%. However, looking at each class individually we notice that there is a fair amount of variance in accuracies between classes. For example, we see that the number 5 has an accuracy below 80%, while the number 1 has an accuracy of 98%.



Classification Report:

	precision	recall	f1-score	support
0	0.94	0.97	0.95	359
1	0.98	0.95	0.97	264
2	0.88	0.85	0.86	198
3	0.87	0.83	0.85	166
4	0.82	0.91	0.86	200
5	0.79	0.89	0.84	160
6	0.95	0.92	0.93	170
7	0.95	0.89	0.92	147
8	0.90	0.80	0.85	166
9	0.89	0.92	0.91	177
accuracy			0.90	2007
macro avg	0.90	0.89	0.89	2007
weighted avg	0.90	0.90	0.90	2007

Figure 4: Classification report for my SVC model.

3.2.1 Comparing our SVC model to scikit's SVC classifier

As we will see later in our performance analysis, the SVC model provided by the `scikit-learn` library gives the highest accuracy rating out of all the models run on this particular dataset. It also does so with much lower variance between classes. Take a look at Figure 5, which shows the confusion matrix for `scikit`'s SVC model in both heatmap and text form, alongside the classification report.

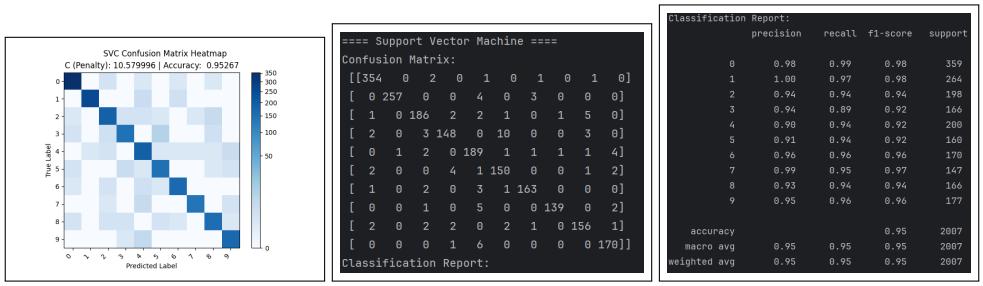


Figure 5: Confusion matrix and classification report for `scikit`'s SVC model.

Comparing `scikit`'s SVC model with my own, we can see that it is not only more accurate, but the per-class accuracies vary significantly less. For example, we see that all of the number 1's have been correctly classified, and that our lowest-accuracy class (the number 4), has a score of 90%.

3.2.2 Accuracy comparison of all models

We now compare the accuracy of all of the models. As previously stated, the deterministic models (GNB, KNN, LogReg, and both SVC models) each run once, while the non-deterministic models (DT, RDF, and MLP) each run 100 times, after which we take each of their average accuracies.

Upon inspection we find that our model actually fares decently in comparison to some of the other models, most notably to Logistic Regression.

The models from the `scikit` library are all set to their default values, as they are not the main focus of this paper. However, this could explain some of the subpar accuracies of some of the models, as some of the parameters may not be set to their optimal values.

3.2.3 Performance results and comparison of all models

Looking at Figure 7, we can see that the `scikit` SVC model (labeled "SVC" in the figure), while being the most accurate, does use the most memory by far. My custom-made model (labeled "SVM-

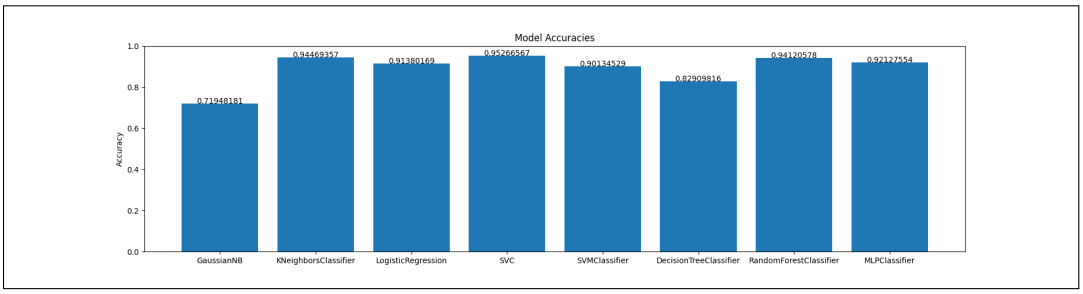


Figure 6: Graph comparison of accuracy scores for all models.

Classifier") has a slightly higher runtime than that of `scikit`'s. Compared to the rest of the models, both SVC classifiers perform well, particularly with regard to runtime, especially when we consider the high accuracy rate for the Support Vector Machine.

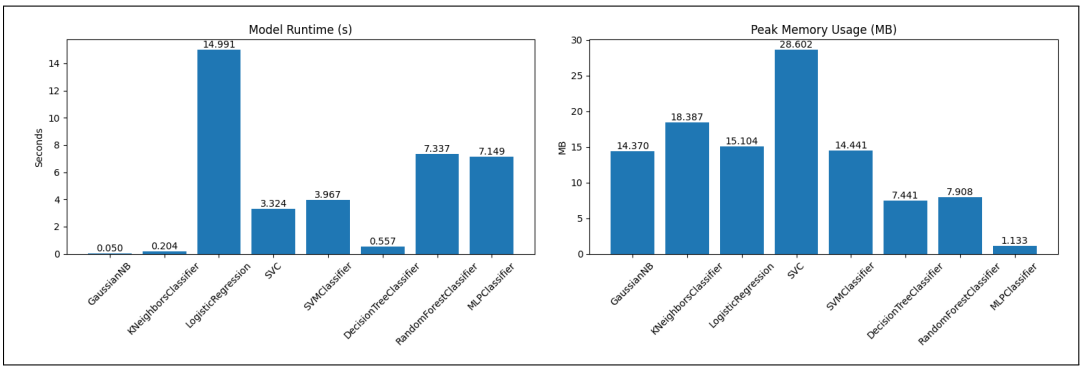


Figure 7: Graph comparison of runtime and memory usage performances for all models.

3.3 Validation

To validate that I was processing the data properly, I created several functions in the `generate.py` module that the user can use to test images against the dataset. The user is able to pass an image to the `image_to_grayscale()` function in `preprocess.py`, which normalizes and converts the image to the same format as each row of the dataset: a 1-D normalized 16×16 grayscale vector of 256 pixel values between $[-1, 1]$, with the first value of that vector being the class label, totaling 257 elements in the 1-D array. When calling this function, the user passes a boolean to either display or not display the normalized version of the image they just passed.

Additionally, the user can call the `generate.plot_random_samples` function, pass it a set of pixels, a set of labels, and an integer specifying the number of samples per set. This will display a figure of a grid, each element of which being a random image in the set, as shown in Figure 8.

Also in `generate.py` are several simple functions that I made for testing input images. These can be used to compare the mean (average), median (eliminating extreme outliers), and mode (only pixels that show up in **all** images) of any number of image sets to produce a cumulative image. The function requires an `X` and `y` parameter (for which I use `training_pixels` and `training_labels`, respectively). After that, the user can enter any number of integer values $[0 - 9]$, and the function will display the mean, median, or mode of the specified image set(s) with respect to the training set or test set. If only the `X` and `y` parameters are passed, the image displayed will be the mean, median, or mode image of the whole (training or test) set.

For example, if the function is called as follows,

```
generate.plot_mean_image(training_pixels, training_labels)
```

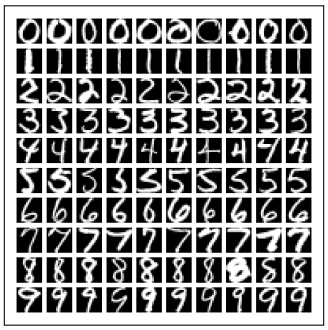


Figure 8: A random sampling of 10 images per class.

then the displayed image will be the mean image of the whole training set. If it is called, say, in the following three ways

```
generate.plot_mean_image(training_pixels, training_labels, 5)
generate.plot_mean_image(training_pixels, training_labels, 5, 2)
generate.plot_mean_image(training_pixels, training_labels, 5, 2, 5)
```

then the images displayed will be the mean image of all images in the class 5, followed by the mean image of all images in the set 5 and the set 2, and finally the mean image of all images in the set 5 and the set 2, with the set 5 factored in twice. This can be done with any number and any combination of sets of classes in the dataset. Figure 9 shows the results of these functions on the set 9.

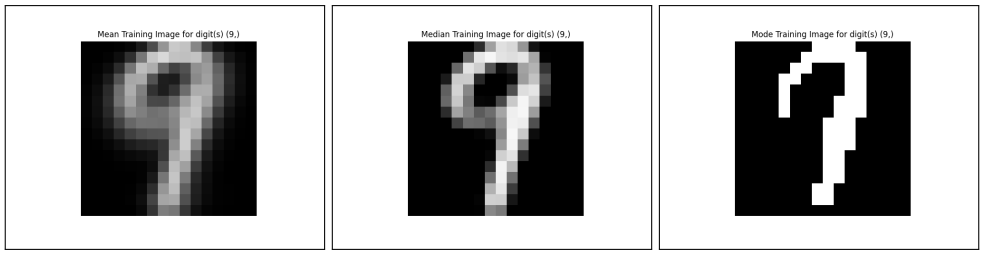


Figure 9: Mean, median, and mode images of the set of handwritten 9's.

4 Discussion and conclusion

We see that our custom SVC held up decently against some of the `scikit` classifiers. While maybe not the most sophisticated model, our machine achieved accuracy roughly competitive with logistic regression, and not falling too far behind the out-of-the-box SVC classifier. Additionally, it had a modest memory footprint compared to `sciklit`'s SVC. The non-parametric models like KNN and Decision Tree offer a nice simplicity, but fall behind in speed. Visualizing the mean, median, and mode prototype images also helps highlight different ways we can capture the essence of an image to give us a better understanding of what the model does.

As far as future modifications go, there are several ways to improve and expand upon this program. Tuning the `regularization_rate` or implementing mini-batch training could definitely help shrink the performance gap between our model and `scikit`'s. Future additions to the program could include something like adaptive learning rate, at which point I'll update this document (along with the source code, of course). All these potential improvements put together could help further highlight the trade-offs between model complexity and interpretability on handwritten digits or other characters.

References

[1] Tsai, Chia-Ling. (2025) Lecture 17: Classification. In Machine Learning (Slides), *SVM*, Slide 28. CUNY Queens College.

[2] Scikit-learn. SVC class documentation. Web. <https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html>

[3] Visually Explained. (2022) Support Vector Machine (SVM) in 2 minutes. Web. YouTube. <https://youtube.com/watch?v=$_Y PScrckx$28>.

[4] Hull, J.J. (1994) USPS Machine Learning Dataset. *Deeplake*. Web. <https://datasets.activeloop.ai/docs/ml/datasets/usps-dataset/>