Utilizing Gaussian Processes to Classify the MNIST Dataset

Ryan Honea

Abstract

Using Gaussian Process Classification in the Scikit learn package, I compare the classification accuracy of various kernels against other common techniques (SVM, Neural Networks, and QDA). With 2500 training samples used on the entire testing set for each algorithm, SVM was by far the strongest, but a Gaussian Process with a Rational Quadratic Product Kernel outperformed a neural network by 10% accuracy by achieving 90% accuracy. Computational run time however showed that Gaussian Process Classification is not the most viable method for image classification.

1 Introduction

1.1 Gaussian Processes

Historically, Machine Learning methods have been constructed by combining as much data as possible from various data sets to create the most accurate model possible. These methods would often be computationally expensive and require extensive research on the features involved. Gaussian processes have been proposed as a method for regressing with little data by utilizing the random nature of the process.

To describe a Gaussian Process (GP), I will first show a specific example of a commonly utilized GP, Brownian Motion. If we let $\{X(t), t \geq 0\}$ be a Brownian motion and define

$$X(t) = \begin{cases} X(t) - 1, & \text{coin flip is heads} \\ X(t) + 1, & \text{coin flip is tails} \end{cases}$$

or each successive case is determined to move forward with probability half or backwards with probability half, then we will observe something similar to the 25 simulated versions in Figure 1. It is clear that these processes are random, but for any given $X(t)$, the expected value will be 0 and the variance will be t.

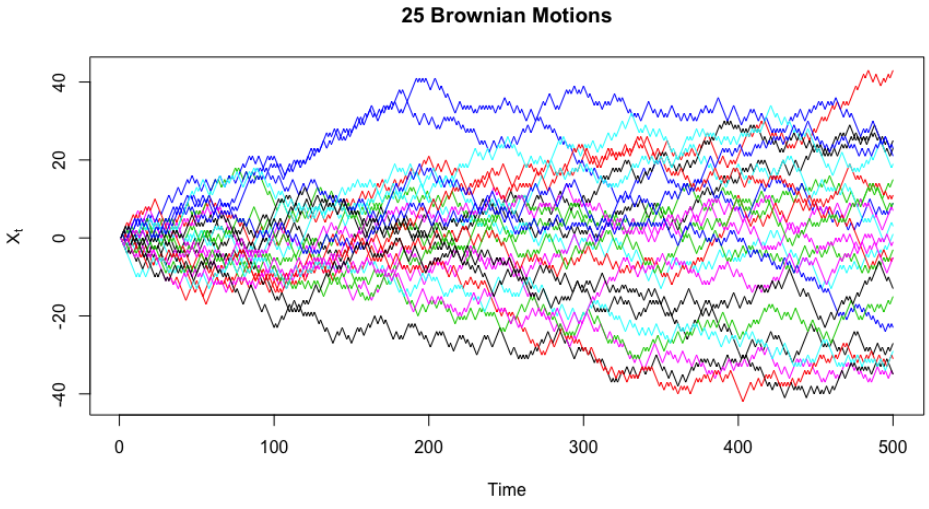

Figure 1: A simulation of 25 Simulated Brownian Motions defined in Section 1

.

This simple case of a Gaussian Process defines every order point $t_{(i)}$ to have a normal distribution with mean 0 and variance t. Now, we reach the formal definition of a Gaussian Process, chosen from Ross's Introduction to Probability Models.[Ros11]

Definition 1.1. A stochastic process $\{X(t), t \geq 0\}$ is called a *Gaussian Process* if $X(t_1), ..., X(t_n)$ has a multivariate normal distribution for all $t_1, ..., t_n$.

Now, in the case of both Brownian Motion and Gaussian Processes, their classical definitions are defined on a continuous space. However, in data observations, we will not observe a continuous space without an infinite number of observations. Therefore, we create a new definition to more accurately portray a dataset.

Definition 1.2. A Gaussian Process on a discrete set shall be defined as a collection of observations that have a multivariate normal distribution. Each observation is defined by a mean vector, $\vec{\mu}$ and a covariance matrix Σ where Σ_{ij} is calculated by some covariance function $\Sigma(\vec{x_i}, \vec{x_j})$.

The covariance can be calculated in a number of ways, but for the rest of this paper, the covariance function used will be

$$\Sigma(\vec{x_i}, \vec{x_j}) = \exp(-||\vec{x_i} - \vec{x_j}||) \tag{1}$$

where $||\vec{x_i} - \vec{x_j}||$ is the Euclidean distance between $\vec{x_i}$ and $\vec{x_j}$. The most important property of this covariance function is that the covariance between points will decay exponentially as two observations decrease in distance. Figure 1.1 shows by simulation what these processes might look like.

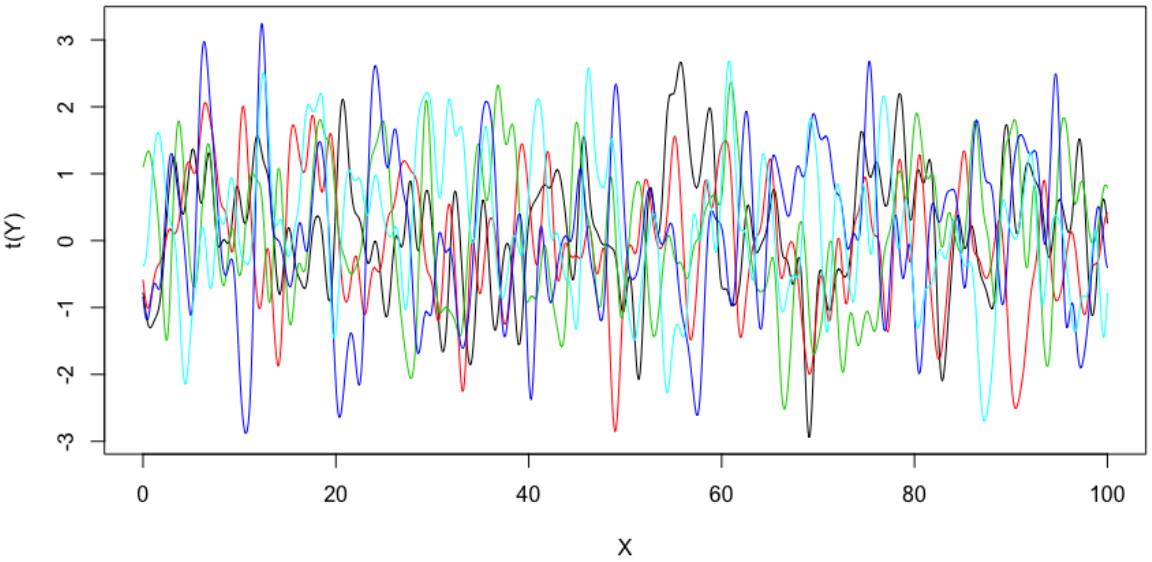

Figure 2: Five simulated Gaussian Processes

If we are to utilize these processes for regression or classification, then we typically use them in a Bayesian framework, i.e. we define a Gaussian Process prior $Y(x) \sim GP$ and then when we apply our data to this prior distribution, a posterior $Y(X) \mid \vec{\omega} = (\vec{x_1}, y_1)...(\vec{x_n}, y_n)$. In order to generate these processes, we need to be able to define that posterior distribution.

First, consider the simple case where we have some additional observation outside of the original dataset called $Y(x)$. If we want to find the conditional distribution of $Y(x)|\omega$, then we find the conditional distribution of a joint multivariate normal distribution with mean zero and some covariance function $\Sigma(x, x')$.

For a joint multivariate distribution defined this way, it is found that $Y(x)|\omega \sim Normal(\mu(x), \sigma^2(x))$ which would have mean and variance

$$\mu(x) = \Sigma(x, \omega_X)\Sigma_n^{-1}\omega_Y$$
$$\sigma^2(x) = 1 - \Sigma(x, \omega_X)\Sigma_n^{-1}\Sigma(x, \omega_X)^T$$

where Σ_n is the covariance matrix of our original data, ω_X is all the X values from the original data, and ω_Y is all the Y values from the original data.

Now, it's not very valuable to have a conditional distribution for a single additional observation if the objective is regression. Therefore, let ζ be the set of infinite observations defined on the range $(\min(\omega_X)-c, \max(\omega_X)+c)$ whose conditional distribution in respect to ω define the regression predictor. Clearly, an infinite set of data points aren't able to be handled computationally, and so let $|\zeta|$ be reasonably large for generation purposes. From the previous equations, this leads to a conditional distribution of

$$\mu(\zeta) = \Sigma(\zeta, \omega_X)\Sigma_n^{-1}\omega_Y \tag{2}$$

and

$$\Sigma(\zeta) = \Sigma(\zeta, \omega_X) - \Sigma(\zeta, \omega_X)\Sigma_n^{-1}\Sigma(\zeta, \omega_X)^T \tag{3}$$

Dr. Robert Gramacy showed the strength of the simplest form of this process on data generated from a Sin Wave[Gra18] which is shown in Figure 1.1. The black line is the mean process while the grey lines are the various other processes crafted $\mu(\zeta)$ and $\Sigma(\zeta)$. An important observation is that the further from the closest point, the larger the variance and thus the wider the Guassian Processes will be outside our data.

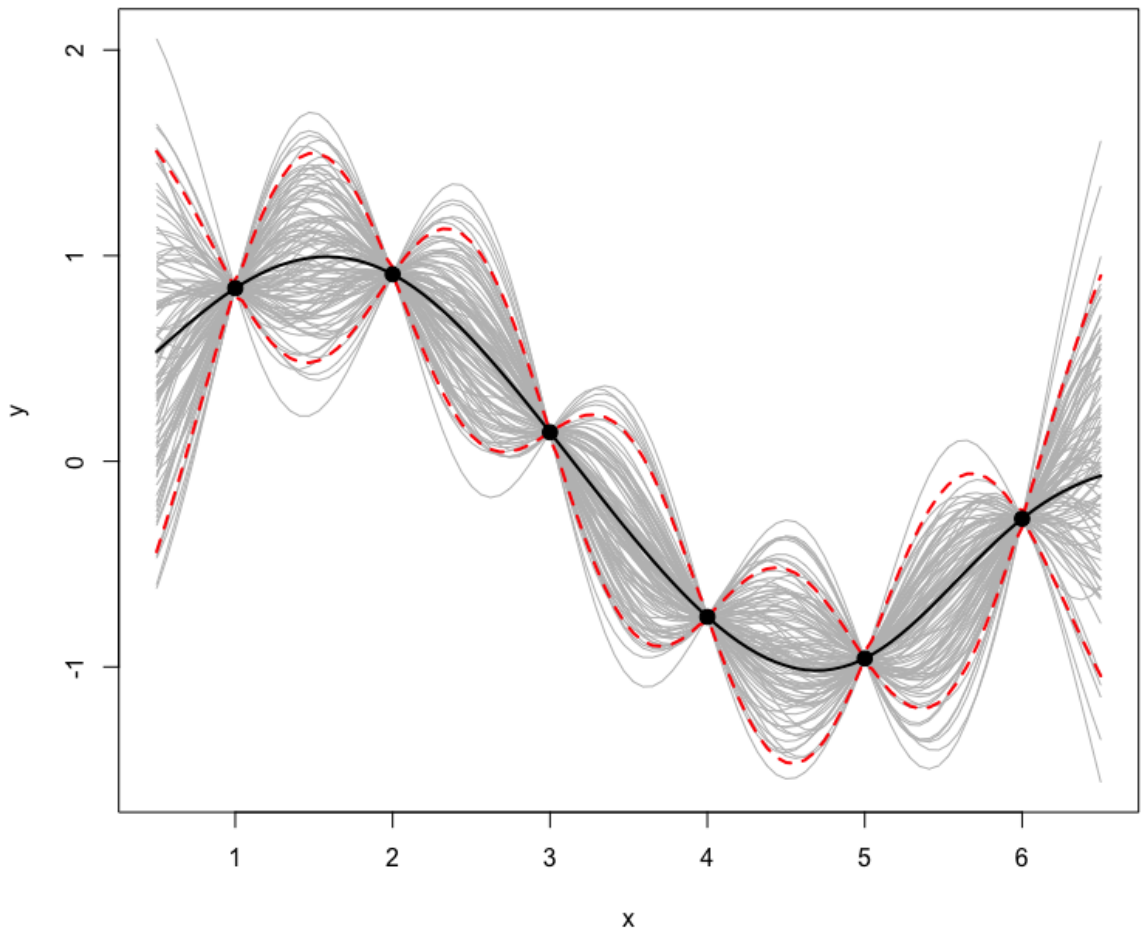

Figure 3: Gaussian Process Regression on a Sin Wave

This is all well and good, but what if we are trying to rest on data with a noise component, i.e. data that we would be expected to see in real life? We can accomplish this by redefining our covariance

matrix as

$$\Sigma\left(\vec{x_i}, \vec{x_j}\right) = \tau\left(\exp(-||\vec{x_i} - \vec{x_j}||) + \delta_{ij}\sigma^2\right) \tag{4}$$

where δ_{ij} is the Kronecker-Delta function ($\delta_{ij} = 1$ when $i = j$) and σ^2 is the variance at that specific point and τ is a constant for scale. Each of these parameters must estimated in some way (and there are several ways to do this).

1.2 The MNIST Numerical Database

The MNIST database has been called the Hello World of Machine Learning and is commonly used as an initial benchmark test for multi-dimensional classification algorithms[LeC98]. The dataset consists of 60,000 training samples and 10,000 testing samples. Each sample is composed of length 784 vector (so 784 features) that can be converted to a 28×28 matrix representing an image. Each number has an associated label between 0 and 9.

As access to a cluster was not available in time, a random subset of 2500 training samples is taken from the data set and then any trained models are tested on the entirety of the testing set.

2 Model Building

In order to classify the images, the Gaussian Process Regression model must be extended to a logistic form. The method for this is described at length in Rasmussen and Williams's text on Gaussian Processes for Machine Learning[RW06], but to put it concisely, we replace the linear component $f(x)$ of classical linear regression with a Gaussian Process and then replace the Gaussian prior on the class weights with a Gaussian Process prior. We also extend the covariance function in this case to allow for multiple covariance functions and have

$$\Sigma\left(\vec{x_i}, \vec{x_j}\right) = \tau K\left(\vec{x_i}, \vec{x_j}\right) + \delta_{ij}\sigma^2 \tag{5}$$

where $K\left(\vec{x_i}, \vec{x_j}\right)$ is some kernel on $\vec{x_i}$ and $\vec{x_j}$.

Selection of this kernel is an important step in the model building process, and we must consider additional complexity added to computation as well as any potential loss of information. Possible Kernels used will be

- Radial Basis Kernel
$$K(\vec{x_i}, \vec{x_j}) = \exp\left(-\frac{1}{2}||\vec{x_i} - \vec{x_j}||^2\right)$$
- Rational Quadratic Kernel
$$K(\vec{x_i}, \vec{x_j}) = \left(\frac{1 + ||\vec{x_i} - \vec{x_j}||^2}{2\alpha\tau^2}\right)^{\alpha}$$
- Dot Product Kernel
$$K(\vec{x_i}, \vec{x_j}) = \sigma^2 + \vec{x_i} \cdot \vec{x_j}$$

Each of these requires some degree of estimation, but for sake of time, the simplest cases will typically be used. The benefits of the radial basis kernel is that it often it used well to describe image information without much loss of information. The rational quadratic is an extension of the radial basis function by emphasizing the effects of the difference with the α value. Finally, the Dot Product Kernel can be described as similar to the polynomial kernel used in support vector machines.

The Gaussian Processes with each kernel will be compared against three other non-parametric classification models:

1. Multi-layer Perceptron Neural Network
2. Support Vector Machines
3. K-Nearest Neighbor

These will also utilize the same subset of 2500 training points for proper comparisons. Similarly, they will all be tested on the entirety of the testing set so that accuracy measures are the same for the comparison models as the Gaussian Process models accuracy metrics.

3 Results

Ultimately, the Rational Quadratic Kernel resulted in the strongest accuracy at approximately 90% which is detailed further in Table 3.

The Radial Basis Function performed very poorly which is interesting because the Rational Quadratic Kernel set with $\alpha = 1$ and $\tau = 1$ performed so well. The difference between these is a loss of the exponent which highlights the closeness of some of the numbers. Indeed, by using the Radial Basis which causes the kernel to decay exponentially with distance, we lose information that the Rational Quadratic maintains. Similarly, the Dot Product performs well as it is nearly a direct comparison that just projects the two $\vec{x}$ vectors into a smaller dimension.

Table 1: Accuracy of Gaussian Process Models and Comparison Models

Model	Accuracy
GP: Radial Basis	0.1009
GP: Rational Quadratic	0.8987
GP: Dot Product	0.8304
SVM: Polynomial	0.9240
K-Nearest Neighbor	0.9068
Multi-layer Perceptron	0.7897

An interesting point in complexity analysis arises when training these models as well. The Gaussian Processes and Support Vector Machines all train with Complexity $O(n^3)$, however, when utilizing Python's Scikit Learn toolbox, Support Vector Machine training would complete nearly ten times faster than the Gaussian Process models. This could be due to increased optimization on Support Vector Machines by the open source community, or it could be due to the added complexity from the Gaussian Processes' Kernels.

3.1 SVM:Poly vs GP:Rational Quadratic

For sake of additional analysis, 4 randomly selected images are shown from the following cases

1. Both the Support Vector Machine and Gaussian Process model predicted correctly.
2. The Support Vector Machine predicted correctly, the Gaussian Process model did not.
3. The Gaussian Process model predicted correctly, the Support Vector machine did not.
4. Neither model predicted correctly.

These two models were chosen to view as they were the best performing comparative model and Gaussian Process model. The images are shown in Figures 3.1 through 3.1.

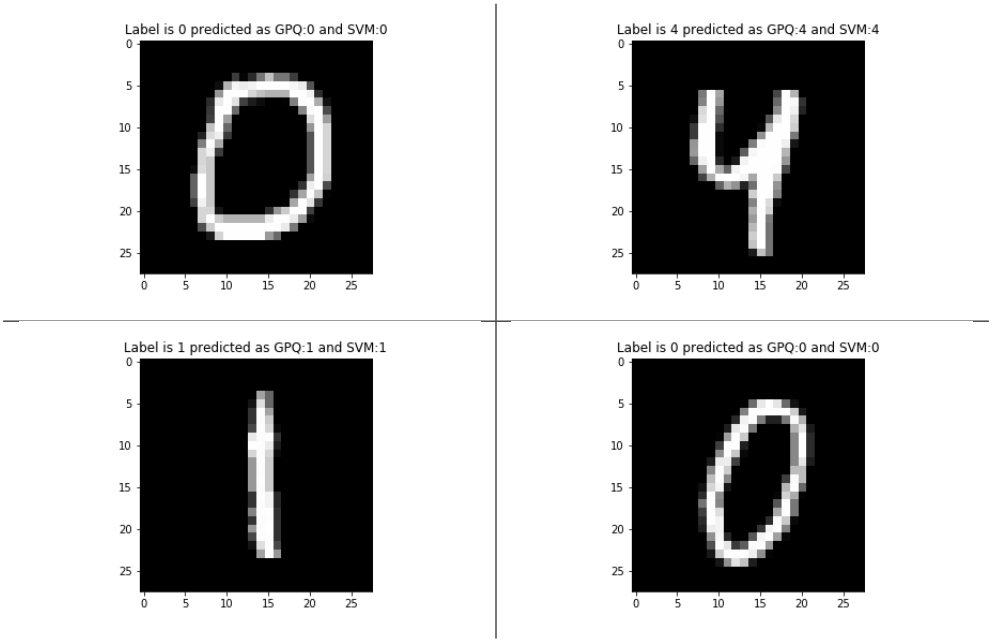

Figure 4: Both the Support Vector Machine and Gaussian Process model predicted correctly.

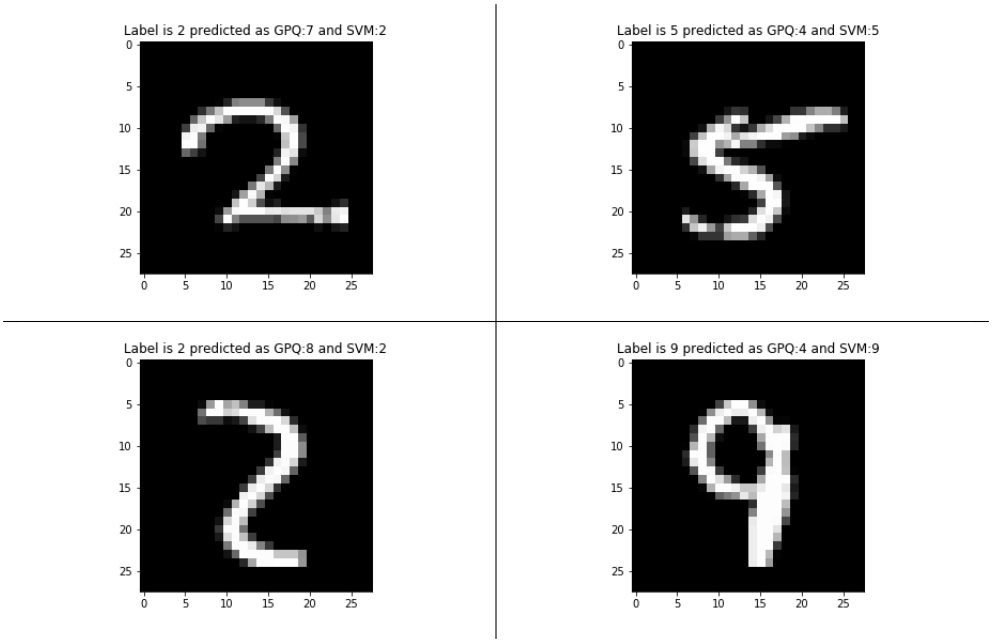

Figure 5: The Support Vector Machine predicted correctly, the Gaussian Process model did not.

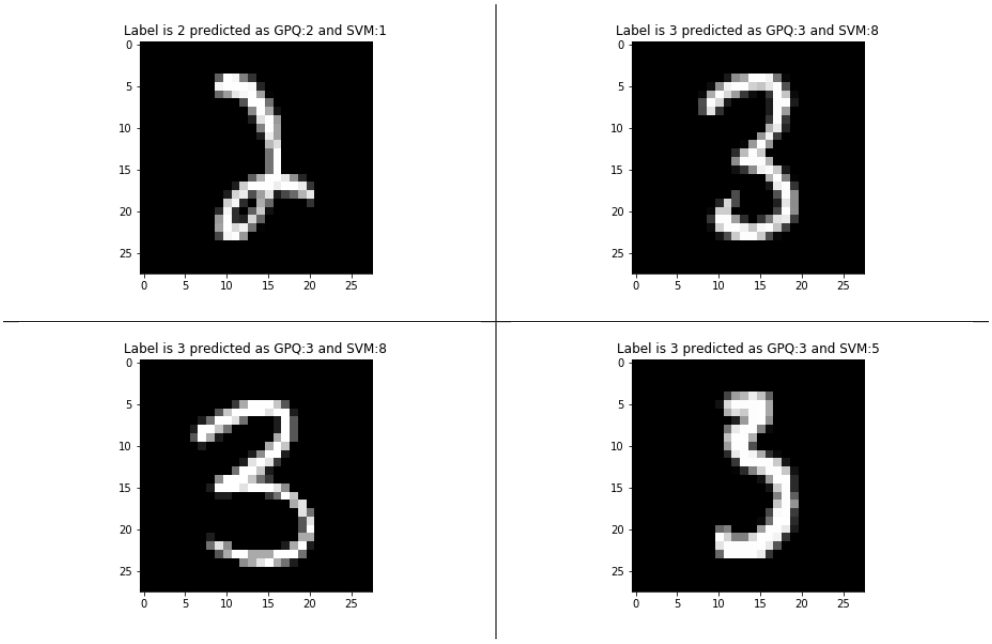

Figure 6: The Gaussian Process model predicted correctly, the Support Vector machine did not.

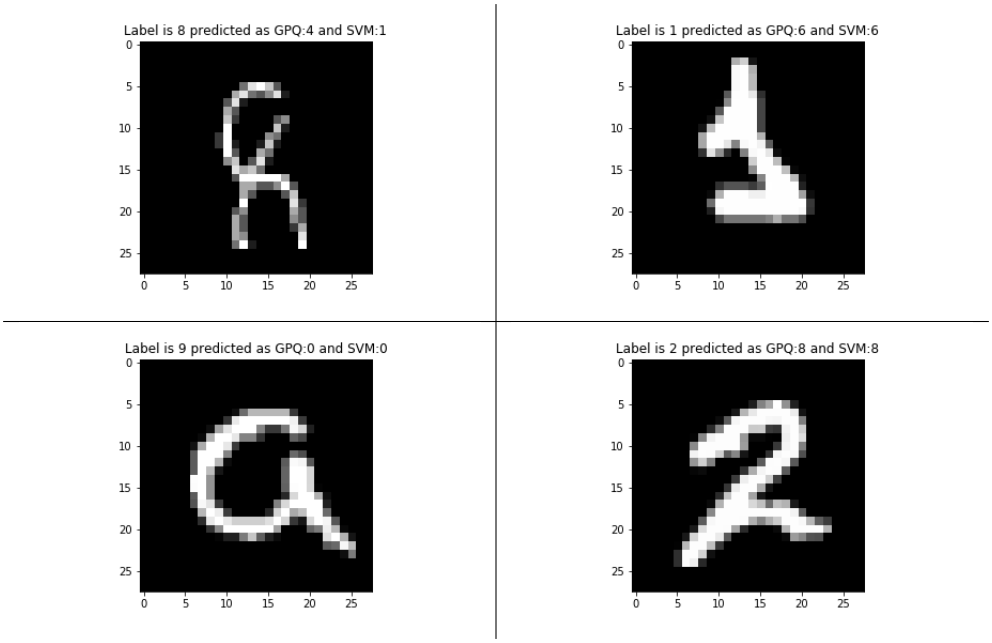

Figure 7: Neither model predicted correctly.

4 Conclusion

There is no clear pattern in the image recognition that shows why one classifier is performing better than another. That being said, the results do show some overlap in misclassification, but not much. This means that a model that utilizes both methods could potentially show additional strength and is a point of future research. Additionally, the Gaussian Processes parameters were not finely tuned due to time constraints and the computational power needed to perform the operations.

As a method for classifying images, the effort that needed to be put into the establishment of a prior over the classification lends towards not using it and opting for computationally cheaper options. Indeed, when the average run of the algorithm took 30+ minutes while the average run of Support Vector Machine implementations only took 3, it makes more sense the use the faster one that remains more accurate. Certainly, there are optimizations that could be made to the algorithm as well as tunings of the kernel that would lead to stronger results, but the strength of the comparitive models suggest that the effort might not lead an analyst anywhere except down a pathway of increased compilation time.

References

[Gra18] Bobby Gramacy. *Gaussian Processes*. 2018. URL: `https://www.youtube.com/watch?v=XxqVPzb_sGM`.

[LeC98] Yann LeCun. *The MNIST Database*. 1998. URL: `http://yann.lecun.com/exdb/mnist/`.

[Ros11] Sheldon Ross. *Introduction to Probability Models*. 2011.

[RW06] Carl Edward Rasmussen and Chris Williams. *Gaussian Processes for Machine Learning*. 2006.

Appendix

Primary Code

```
# Miscellaneous Packages
import numpy as np
from matplotlib import pyplot as plt
from matplotlib import cm
from mnist import MNIST
from sklearn.externals import joblib
import random

# ML Packages
from scipy import stats
from sklearn import svm
from sklearn.gaussian_process import GaussianProcessClassifier
from sklearn.gaussian_process.kernels import DotProduct
from sklearn.gaussian_process.kernels import RationalQuadratic
from sklearn.gaussian_process.kernels import RBF
from sklearn.neural_network import MLPClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn import datasets
from sklearn.datasets import fetch_mldata


# Fetches Data from SciKit Learn repository and converts
# to a Numpy vector
def prepare_data():
    mnist = fetch_mldata('MNIST_original', data_home='./')
    mnist.keys()

    images = mnist.data
    targets = mnist.target

    X_data = images
    Y = targets
    shuf = random.sample(range(len(X_data)), 2500)

    X_train = []
    for x in shuf: X_train.append(X_data[x])

    y_train = []
    for x in shuf: y_train.append(Y[x])

    c_shuf = set(range(len(X_data))) - set(shuf)

    X_test = []
    for x in c_shuf: X_test.append(X_data[x])

    y_test = []
    for x in c_shuf: y_test.append(Y[x])

    return X_train, X_test, y_train, y_test


# Assigns data
train_nums, test_nums, train_labels, test_labels = prepare_data()
```

```
# SVM
sv = svm.SVC(kernel = 'poly')
sv.fit(train_nums, train_labels)
svm_accuracy = sum(sv.predict(test_nums) == test_labels)/len(test_nums)
print(svm_accuracy)

# Gaussian Process Classifier
gp = GaussianProcessClassifier(kernel = RBF(length_scale = 1.0), n_jobs = 4, multi_class
gp.fit(train_nums, train_labels)
gp_accuracy = sum(gp.predict(test_nums) == test_labels)/len(test_labels)
print(gp_accuracy)

# Guassian Quadratic
kernelQuad = RationalQuadratic()
gp_Quad = GaussianProcessClassifier(multi_class = 'one_vs_rest',
                                    kernel = kernelQuad, n_jobs = -1)
gp_Quad.fit(train_nums[:2500], train_labels[:2500])
gp_Quad_accuracy = sum(gp_Quad.predict(test_nums) == test_labels)/len(test_labels)

# Guassian Dot Product
kerneldot = DotProduct()
gp_dot = GaussianProcessClassifier(multi_class = 'one_vs_rest',
                                    kernel = kerneldot, n_jobs = -1)
gp_dot.fit(train_nums[:2500], train_labels[:2500])
gp_dot_accuracy = sum(gp_dot.predict(test_nums) == test_labels)/len(test_labels)

# Multilayer Perceptron Neural Network
mlp = MLPClassifier(alpha = 1, max_iter = 10000)
mlp.fit(train_nums[:2500], train_labels[:2500])
mlp_accuracy = sum(mlp.predict(test_nums) == test_labels)/len(test_nums)

# Quadratic Discriminant Analysis
knn = KNeighborsClassifier(n_neighbors=5)
knn.fit(train_nums[:2500], train_labels[:2500])
knn_accuracy = sum(knn.predict(test_nums) == test_labels)/len(test_nums)
```

Gaussian Processes Sin Curve Regression

```
library(plgp)
library(mvtnorm, quietly=TRUE)
library(dplyr)

eps <- sqrt(.Machine$double.eps) ## defining a small number


X <- matrix(seq(0, 2*pi, length = 10), ncol = 1)
y <- sin(X)
D <- distance(X)
Sigma <- exp(-D)


XX <- matrix(seq(min(X)-1,max(X)+1, length = 200), ncol = 1)
DXX <- distance(XX)
SXX <- exp(-DXX) + diag(eps, ncol(DXX))
DX <- distance(XX, X)
SX <- exp(-DX)

Si <- solve(Sigma)
mup <- SX %*% Si %*% y
Sigmap <- SXX - SX %*% Si %*% t(SX)

YY <- rmvnorm(100, mup, Sigmap)

q1 <- mup + qnorm(0.05, 0, sqrt(diag(Sigmap)))
q2 <- mup + qnorm(0.95, 0, sqrt(diag(Sigmap)))

matplot(XX, t(YY), type = "l", col = "gray", lty = 1,
        main = "Sin Curve Regression", xlab = "X", ylab = "Sin(X)")
points(X, y, pch = 20, cex = 2)
lines(XX, mup, lwd=2);
lines(XX, q1, lty = 2, col = "red")
lines(XX, q2, lty = 2, col = "red")


fit <- GP_fit(X, y)
pred <- predict(fit, seq(-1,2*pi + 1,by=.1))
lines(seq(-1,2*pi+1,by=.1), pred$Y_hat, type = "l", col = "blue", lty = 2)
legend("topleft", c("General Model", "Optimized Model", "95% Confidence Interval"),
       lty = c(1,2,2), col = c("Black", "Blue", "Red"))
```