# Homework 2 report

## Machine Learning

Institute of Computer Science and Engineering

13/05/2024

***Produced by :***
PAULY Alexandre
alexandre.pauly@cy-tech.fr

# Contents

# 1 Introduction

**Context:**

As part of the Machine Learning module, an assignment focusing on Gaussian Process Regression (GPR) and Support Vector Machines (SVM) has been initiated. This assignment aims to deepen understanding and practical implementation skills in these specific areas of machine learning. Gaussian Process Regression offers a probabilistic approach to regression tasks, allowing for the estimation of not only the function values but also their uncertainties. On the other hand, SVM is a powerful supervised learning algorithm used for classification and regression tasks, particularly effective in high-dimensional spaces.

**Objective:**

In this report, we focus on understanding and implementing GPR from scratch and SVM from the library *libsvm*. The main objective is to explain in detail each step of the implementation of these two techniques, including the underlying mathematical principles, computational procedures, and visualization techniques. Moreover, we will demonstrate the application of the SVM on the MNIST dataset, a collection of handwritten digits.

# 2 Implementation

## 2.1 Gaussian process regression (GPR)

Gaussian Process Regression is a powerful non-parametric probabilistic regression technique used to model complex relationships between input and output variables. It offers a flexible framework for regression tasks by providing not only point estimates but also measures of uncertainty. In this section, we delve into the implementation of GPR to predict the distribution of a latent function given noisy observations.

To implement Gaussian Process Regression, we start by loading and pre-processing the training data, which consists of input-output pairs. The input data is represented as a 2D matrix where each row corresponds to a data point, and the output data is the noisy observation Yi associated with each input point Xi. We then proceed with the following steps:

**1st step : Covariance matrix**

To apply a Gaussian distribution and predict its parameters, we need to calculate its covariance for each element as follows with $\beta$ as regulation parameter and $\delta_{ij}$, which is 1 if i=j and 0 otherwise :

$$C(x_i, x_j) = k(x_i, x_j) + \beta^{-1}\delta_{ij} \tag{1}$$

We compute the covariance matrix using a rational quadratic kernel function **(2)**. This kernel function measures the similarity between data points based on their Euclidean distances and is characterized by hyperparameters such as $\ell$, $\sigma^2$ and $\alpha$ where :

- $\alpha$ : The scale-mixture.
- $\ell$ : The lengthscale of the kernel.
- $\sigma^2$ : The kernel variance.
- $(x_i - x_j)^2$: The euclidean distance between $x_i$ and $x_j$.

$$k(x_i, x_j) = \sigma^2 \left(1 + \frac{(x_i - x_j)^2}{2\alpha\ell^2}\right)^{-\alpha} \tag{2}$$

```
1 def rational_quadratic_kernel(xi, xj, alpha, lengthscale,
     kernel_variance):
2     """! Function to calculate the rational quadratic kernel.
         """
3     k = kernel_variance * (1 + ((xi-xj) ** 2) / (2 * alpha *
         lengthscale**2)) ** (-alpha)
4     return k
```

***Code 1 :*** *Rational quadratic kernel*

```python
def covariance_matrix(data, beta, alpha, lengthscale, kernel_variance):
    """! Function to calculate the covariance matrix of the sample data."""

    # Initialization of variable
    n = len(data)                  # Number of data
    cov_matrix = np.zeros((n, n)) # Covariance matrix

    # For each row
    for i in range(n):
        # For each column
        for j in range(n):
            cov_matrix[i][j] = rational_quadratic_kernel(data[i], data[j], alpha, lengthscale, kernel_variance)

            # On the diagonal, add 1/beta
            if i == j:
                cov_matrix[i][j] += 1 / beta

    return cov_matrix
```

***Code 2 :*** *Covariance matrix*

**2nd step : Estimate parameters**

With the covariance matrix computed, we use it to estimate the mean and variance of the Gaussian process by the below formula with :

- $x^*$ : The input for which you want to predict the output.
- k(x, $x^*$) : The covariance vector between the training data x and the new input $k^*$
- C : The covariance matrix created in step 1, it references relationship between training data.
- $\mu(x^*)$ : The mean of the posterior distribution for the new input $x^*$, representing the output prediction.
- y : The vector of target values for training data.
- $\sigma^2(x^*)$ : The variance of the posterior distribution for the new input $x^*$, representing the uncertainty of the prediction.
- $\beta$ : The regulation parameter.

$$\begin{aligned} \mu(x^*) &= k(x, x^*)^T C^{-1} y \\ \sigma^2(x^*) &= k(x, x^*)^T C^{-1} k(x, x^*) \\ k^* &= k(x, x^*) + \beta^{-1} \end{aligned} \tag{3}$$

```python
def gaussian_process_parameters(data, target, cov_matrix, beta, predict_sample_size, alpha, lengthscale, kernel_variance):
    """! Function to calculate the parameters of the gaussian process method."""

    # Initialization of variable
    n = len(data)                                          # Number of data
    mean = np.zeros(predict_sample_size)                   # Mean
    variance = np.zeros(predict_sample_size)               # Variance
    x_sample = np.linspace(-60, 60, predict_sample_size)   # Datapoints creation

    for sample in range(predict_sample_size):
        # Kernel initialization
        kernel = np.zeros((n, 1))
        for i in range(n):
            kernel[i][0] = rational_quadratic_kernel(data[i], x_sample[sample], alpha, lengthscale, kernel_variance)

        kernel_star = rational_quadratic_kernel(x_sample[sample], x_sample[sample], alpha, lengthscale, kernel_variance) + 1 / beta

        # Parameters calculation
        mean[sample] = np.dot(np.dot(kernel.T, inv(cov_matrix)), target)
        variance[sample] = kernel_star - np.dot(np.dot(kernel.T, inv(cov_matrix)), kernel)

    return mean, variance
```

***Code 3 :*** *Parameters estimation*

### 3rd step : Kernel parameters optimization

To optimize kernel parameters, we used the following formula to minimize the nagative marginal loglikelihood like :

$$ln(p(y|\theta)) = -\frac{1}{2}ln|C| - \frac{1}{2}y^T C^{-1} y - \frac{N}{2}ln(2\pi) \tag{4}$$

```
    """! Function to calculate the optimized parameters of
       the gaussian process method. """

    # Initialization of variables
    alpha = theta[0]            # Alpha
    lengthscale = theta[1]      # Lengthscale
    kernel_variance = theta[2] # Kernel variance
    n = len(data)               # Number of data

    # Negative marginal log likelihood
    cov_matrix_theta = covariance_matrix(data, beta, alpha,
       lengthscale, kernel_variance)
    likelihood = np.log(np.linalg.det(cov_matrix_theta)) / 2
    likelihood += np.dot(np.dot(target.T, inv(
       cov_matrix_theta)), target) / 2
    likelihood += n / 2 * np.log(2 * np.pi)

    return likelihood
```

***Code 4 :*** *Plotting results*

### 4th and last step : Plotting results

Finally, we visualize the results by plotting the mean function along with its 95% confidence interval. This visualization allows us to assess the quality of the regression model and the uncertainty associated with the predictions. For better visualization, we display a 95% confidence interval based on the data, so for a normal distribution, we'll have a quantile of 1.96 because it's a symmetrical interval.

```
def plot_gaussian_process(x, y, mean, variance,
   predict_sample_size):
    """! Function to display the plot of Gaussian process
       method. """

    # Line to represent mean of f in range [-60,60]
    x_sample = np.linspace(-60, 60, predict_sample_size)

    # Confidence interval of f
```

```
    interval = 1.96 * (variance ** 0.5) # 1.96 by reading law table of normal law

    # Plot display
    plt.scatter(x, y, color = 'k')
    plt.plot(x_sample, mean, color = 'b')
    plt.plot(x_sample, mean + interval, color = 'r')
    plt.plot(x_sample, mean - interval, color = 'r')
    plt.fill_between(x_sample, mean + interval, mean - interval, color = 'pink', alpha = 0.3)
    plt.show()
```

***Code 5 :*** *Plotting results*

## 2.2 Support Vector Machines (SVM)

*"Support Vector Machines (SVMs) are a type of supervised machine learning algorithm used for classification and regression tasks. They are widely used in various fields, including pattern recognition, image analysis, and natural language processing."*

**1st case : Linear, polynomial and RBF kernels**

To carry out this implementation, we used the libsvm library, in particular the functions svm_train and svm_predict where -q is the quiet mode (only final output) -t represent the type of kernel like :

- 0 : linear kernel
- 1 : polynomial kernel
- 2 : RBF kernel

```
model = svm_train(Y_train, X_train, f'-t {i} -q')
result = svm_predict(Y_test, X_test, model)
```

***Code 6 :*** *Use of svm_train and svm_predict*

So, with the following code we can compare easily the three different methods :

```
for i in range(3):
    if i == 0:
        print("Linear kernel :")
    elif i == 1:
        print("Polynomial kernel :")
    elif i == 2:
        print("RBF kernel :")
    model = svm_train(Y_train, X_train, f'-t {i} -q')
```

```
9     result = svm_predict(Y_test, X_test, model)
```

***Code 7 :*** *Comparison with different kernels*

**2nd case : C-SVC**

In every type of kernel, the cost for C-SVC search in [0.001, 0.01, 0.1, 1, 10, 100] and do 3-fold cross validation. The parameters in svm_train are :

- -s : set type of SVM
- -c : set the parameter C of C-SVC
- -v : n-fold cross validation mode

So, now, we need a function to compare current option with previous optimal option like the cross validation :

```
1  def best_option(data, target, option, optimal_option,
       optimal_accuracy):
2      """! Function to choose the best option. """
3
4      # SVM algo
5      accuracy = svm_train(target, data, option)
6
7      # Search for greater accuracy
8      if accuracy > optimal_accuracy:
9          return option, accuracy
10     else:
11         return optimal_option, optimal_accuracy
```

***Code 8 :*** *Choose best option*

But, to compare different options, we need to initialize parameters as following :

- cost = [0.001, 0.01, 0.1, 1, 10, 100] for each kernels. We will search C in cost.
- Polynomial kernel :
  - gamma = [0.0001, 1/784, 0.01, 0.1, 1, 10] in pair with -g option
  - coefficient = [-10, -5, 0, 5, 10] in pair with -r option
  - degree = [1, 2, 3, 4] in pair with -d option
- RBF kernel :
  - gamma = [0.0001, 1/784, 0.01, 0.1, 1, 10] in pair with -g option
- Sigmoid kernel :
  - gamma = [0.0001, 1/784, 0.01, 0.1, 1, 10] in pair with -g option

– coefficient = [-10, -5, 0, 5, 10] in pair with -r option

```
def grid_search(data, target, kernel):
    """! Search function for finding parameters of best performing model."""

    # Initialization of variables
    cost = [0.001, 0.01, 0.1, 1, 10, 100] # Costs value to test
    optimal_option = f'-s 0 -v 3 -q'      # Options
    optimal_accuracy = 0                  # Optimal accuracy

    # If it's linear
    if kernel == 0:
        print('\nLinear kernel:')
        for c in cost:
            # Options initialization for svm_train
            option = f'-s 0 -t 0 -c {c} -q -v 3'
            print(option)

            # Cross validation
            optimal_option, optimal_accuracy = cross_validation(data, target, option, optimal_option, optimal_accuracy)
    # If it's polynomial
    elif kernel == 1:
        # Initialization of variables
        gamma = [0.0001, 1/784, 0.01, 0.1, 1, 10] # Gamma values to test
        coefficient = [-10, -5, 0, 5, 10]         # Coefficients in kernel
        degree = [1, 2, 3, 4]                     # Degree for polynomial kernel

        print('\nPolynomial kernel:')
        for c in cost:
            for d in degree:
                for g in gamma:
                    for r in coefficient:
                        # Options initialization for svm_train
                        option = f'-s 0 -t 1 -c {c} -d {d} -g {g} -r {r} -q -v 3'
                        print(option)
```

```
34
35                          # Cross validation
36                          optimal_option, optimal_accuracy =
                               cross_validation(data, target,
                               option, optimal_option,
                               optimal_accuracy)
37      # If it's RBF
38      elif kernel == 2:
39          # Initialization of variables
40          gamma = [0.0001, 1/784, 0.01, 0.1, 1, 10] # Gamma
                values to test
41
42          print('\nRBF kernel:')
43          for c in cost:
44              for g in gamma:
45                  # Options initialization for svm_train
46                  option = f'-s 0 -t 2 -c {c} -g {g} -q -v 3'
47                  print(option)
48
49                  # Cross validation
50                  optimal_option, optimal_accuracy =
                       cross_validation(data, target, option,
                       optimal_option, optimal_accuracy)
51      # If it's sigmoid
52      elif kernel == 3:
53          # Initialization of variables
54          gamma = [0.0001, 1/784, 0.01, 0.1, 1, 10] # Gamma
                values to test
55          coefficient = [-10, -5, 0, 5, 10]         #
                Coefficients in kernel
56
57          print('\nSigmoid kernel:')
58          for c in cost:
59              for g in gamma:
60                  for r in coefficient:
61                      # Options initialization for svm_train
62                      option = f'-s 0 -t 3 -c {c} -g {g} -r {r}
                            -q -v 3'
63                      print(option)
64
65                      # Cross validation
66                      optimal_option, optimal_accuracy =
                           cross_validation(data, target, option,
                            optimal_option, optimal_accuracy)
```

```
67
68      optimal_option = optimal_option[:-5]
69
70      # Results display
71      print(optimal_accuracy)
72      print(optimal_option)
73
74      return optimal_option
```

***Code 9 :*** *Search best performing model function*

**3rd case : Linear kernel + RBF kernel**

On the same principle, when we want to combine the use of two kernels, we need to prepare the kernel calculation for the svm_train and svm_predict functions where $\gamma = \frac{1}{number\ of\ data}$.

$$\begin{aligned} Linear\ kernel &: u^T * v \\ RBF\ kernel &: e^{-\gamma||u-v||^2} \end{aligned} \tag{5}$$

```
1  def linear_kernel(u, v):
2      """! Function to calculate the linear kernel."""
3
4      return u @ v.T
5
6  def RBFkernel(u, v):
7      """! Function to calculate the RBF kernel."""
8
9      gamma = 1 / u.shape[1]
10     dist = np.sum(u ** 2, axis=1).reshape(-1, 1) + np.sum(v
           ** 2, axis=1) - 2 * u @ v.T
11
12     return np.exp(-gamma * dist)
```

***Code 10 :*** *Linear and RBF kernels*

Using these functions, we can add these two kernels to calculate predictions with a pre-calculated kernel (i.e. -t = 4). And as with any modelling method, the model is adjusted on the training data before calculating predictions on the test set.

```
1  # Initialization of variables
2  n_train = len(X_train) # Number of training images
3  n_test = len(X_test)   # Number of test images
4
5  # Kernel calculation of training and test images
6  train_kernel = linear_kernel(X_train, X_train) + RBFkernel(
       X_train, X_train)
```

```
7  test_kernel = linear_kernel(X_test, X_train) + RBFkernel(
       X_test, X_train)
8
9  # Add index in front of kernel (Required for compatibility
       with the library)
10 train_kernel = np.hstack((np.arange(1,n_train + 1).reshape
       (-1, 1), train_kernel))
11 test_kernel = np.hstack((np.arange(1,n_test + 1).reshape(-1,
       1), test_kernel))
12
13 model = svm_train(Y_train, train_kernel, '-t 4 -q')
14 result = svm_predict(Y_test, test_kernel, model)
```

***Code 11 :*** *Prediction*

# 3 Results

## 3.1 Gaussian process regression (GPR)

**Dataset :**

The data used in this study come from the *input.data* file supplied and are as follows:

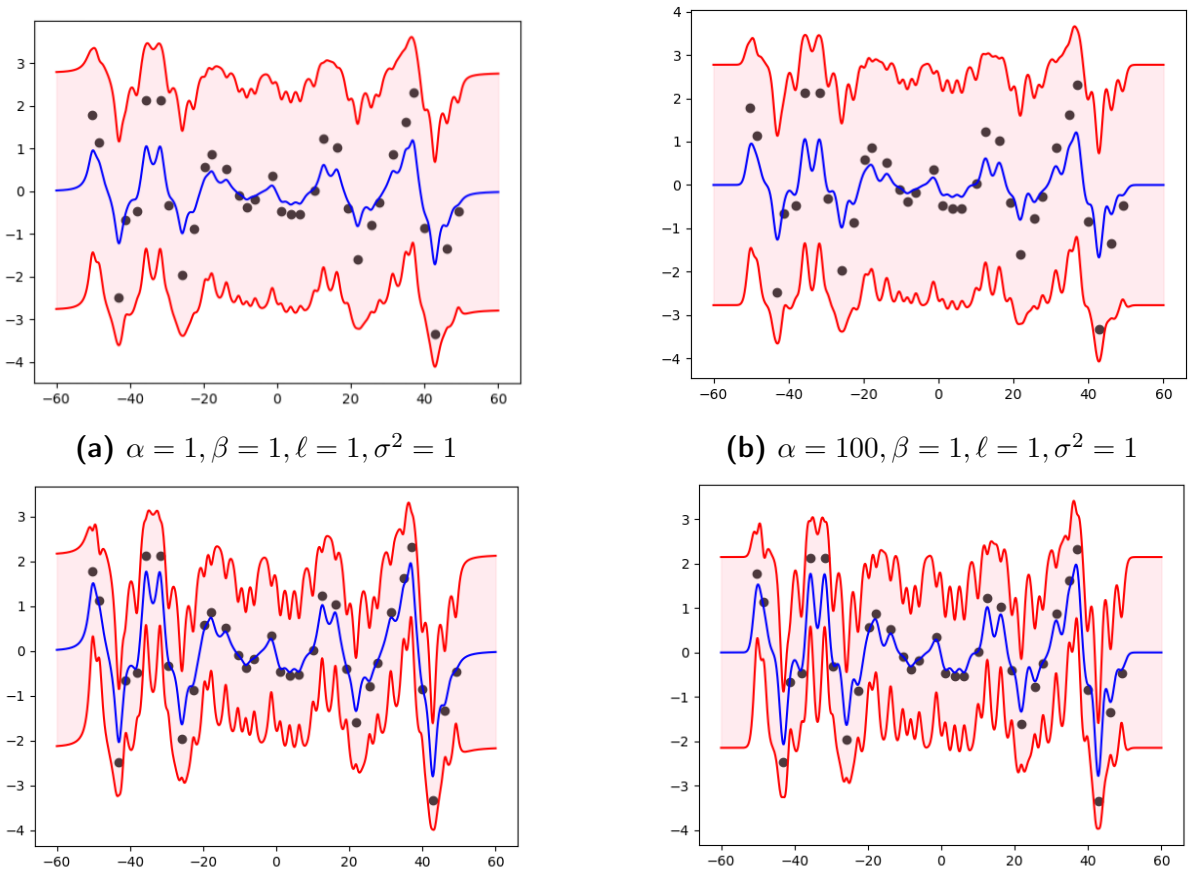
```
-5.018038795695388643e+01 1.774746809131112046e+00
-4.833784599279169925e+01 1.130536934559104534e+00
-4.302028859716906339e+01 -2.475911069911202134e+00
-4.129017073572108387e+01 -6.658953170536108246e-01
-3.806586017284342915e+01 -4.729198308342458246e-01
-3.565718050254557170e+01 2.130641129617679486e+00
-3.175855024358983414e+01 2.118813811282026283e+00
-2.951857591431968331e+01 -3.196856711841976617e-01
-2.581803824914714340e+01 -1.960597635454770504e+00
-2.268316859590379053e+01 -8.724173954600775716e-01
-1.964445405169167103e+01 5.740047765630208465e-01
-1.776974942243885991e+01 8.676390358505385869e-01
-1.380265473516695351e+01 5.141365067427436930e-01
-1.046513416251222317e+01 -9.810668860749949260e-02
-8.325354601305097191e+00 -3.731931361113974832e-01
-6.041351180348173422e+00 -1.808535325581353270e-01
-1.329031538332497764e+00 3.516710819870855209e-01
1.097877623658328305e+00 -4.632926363360667654e-01
3.652659243622094820e+00 -5.371530550462246811e-01
6.248188739533526714e+00 -5.364893276186940563e-01
1.001130876357145105e+01 2.314668661886454260e-02
1.254132798246287095e+01 1.226304302981155558e+00
1.620040364712260583e+01 1.029662116642863001e+00
1.921715213051720283e+01 -4.033065347311380888e-01
2.180883040391566041e+01 -1.599366609761119662e+00
2.549834124894415410e+01 -7.767816641190832261e-01
2.769579332199591093e+01 -2.555539188359608471e-01
3.128894536032172269e+01 8.641761457612352482e-01
3.477796621355683016e+01 1.625672793134455008e+00
3.692803505082112991e+01 2.317634400344302126e+00
3.979619017618449561e+01 -8.454503575823562045e-01
4.278911824608752568e+01 -3.335429342355324511e+00
4.613749437395509290e+01 -1.338092015158637071e+00
4.910408077094962209e+01 -4.687881992375469986e-01
```

**Figure 1:** Dataset for the Gaussian procession regression method

**Prediction quality :**

We can see from the following figures that despite the difference in value for $\alpha$ ($\alpha$ 100 times higher), it has no influence on the model. This can be explained by the fact that $\alpha$ is only a scaling factor. As it tends to infinity, the rational quadratic kernel converges to

the exponential quadratic kernel.



**(a)** $\alpha = 1, \beta = 1, \ell = 1, \sigma^2 = 1$

**(b)** $\alpha = 100, \beta = 1, \ell = 1, \sigma^2 = 1$

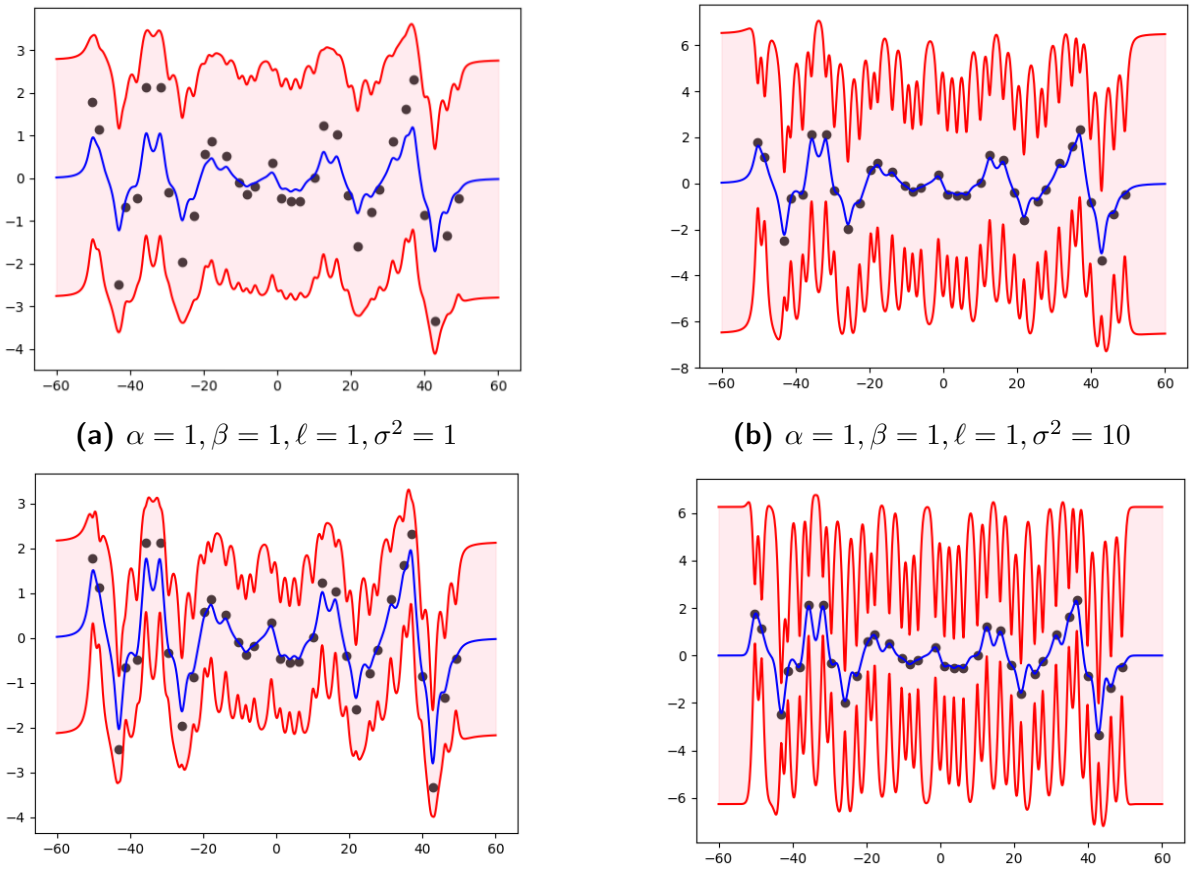
**(c)** $\alpha = 1, \beta = 5, \ell = 1, \sigma^2 = 1$

**(d)** $\alpha = 100, \beta = 5, \ell = 1, \sigma^2 = 1$

**Figure 2:** Comparison of GPR with different configurations of $\alpha$

Varying the variance has a much greater impact on the results, which was predictable. Where increasing $\beta$ reduces the importance of noise in the model, and as a result, the predictions seem more smoothed. Moreover, a lower value of $\beta$ leads to wider confidence intervals around the model's predictions, indicating lower confidence in these predictions.

In the following figures, we can see that when $\sigma^2$ is too large, it leads to uncertainty in the model's predictions. This means that the model becomes less confident in its predictions and the confidence intervals around the predictions become wider, even at $\beta > 1$.



**(a)** $\alpha = 1, \beta = 1, \ell = 1, \sigma^2 = 1$

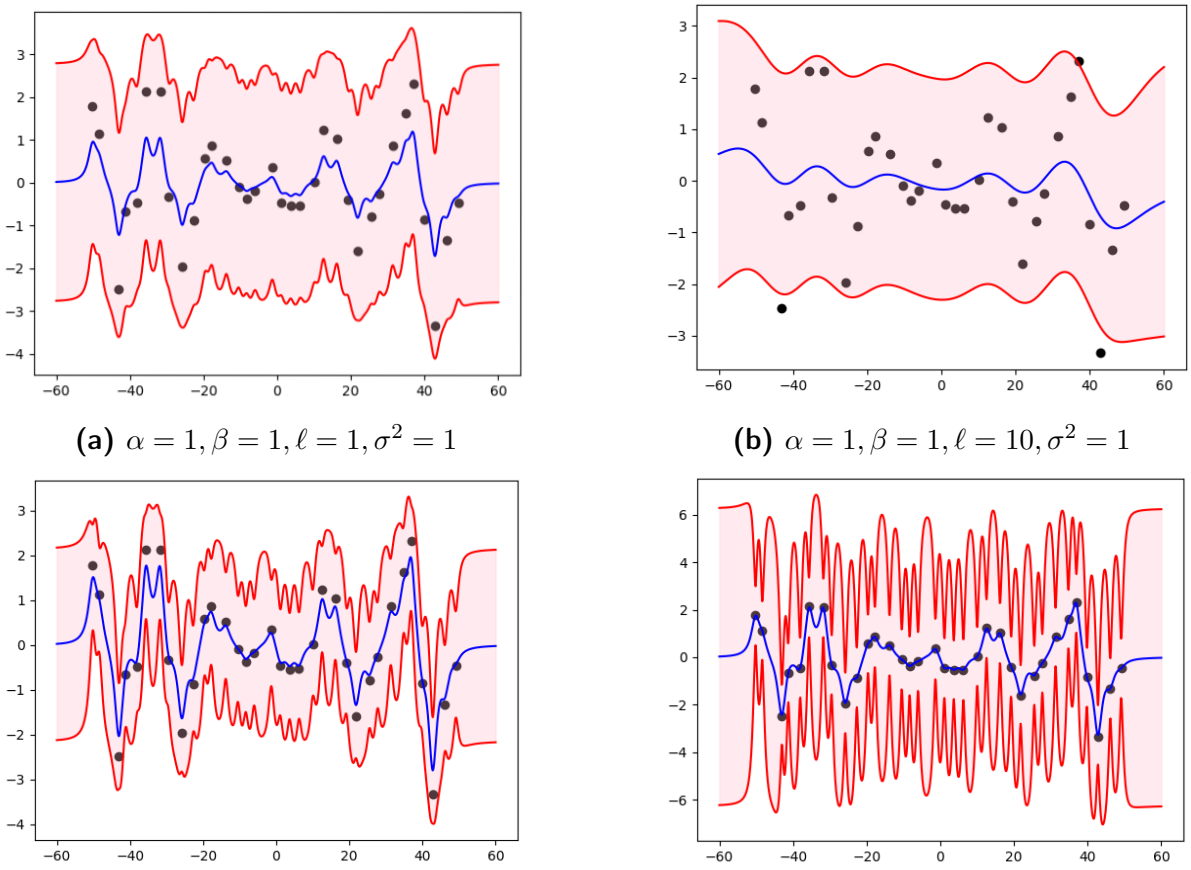
**(b)** $\alpha = 1, \beta = 1, \ell = 1, \sigma^2 = 10$

**(c)** $\alpha = 1, \beta = 5, \ell = 1, \sigma^2 = 1$

**(d)** $\alpha = 1, \beta = 5, \ell = 1, \sigma^2 = 10$

**Figure 3:** Comparison of GPR with different configurations of $\sigma^2$

A larger length scale ($\ell = 10$) makes the covariance function flatter and smoother, which smooths the model's predictions. This means that the model considers more distant points as more similar, reducing sensitivity to local data fluctuations and thus becoming less adaptive.



**(a)** $\alpha = 1, \beta = 1, \ell = 1, \sigma^2 = 1$

**(b)** $\alpha = 1, \beta = 1, \ell = 10, \sigma^2 = 1$

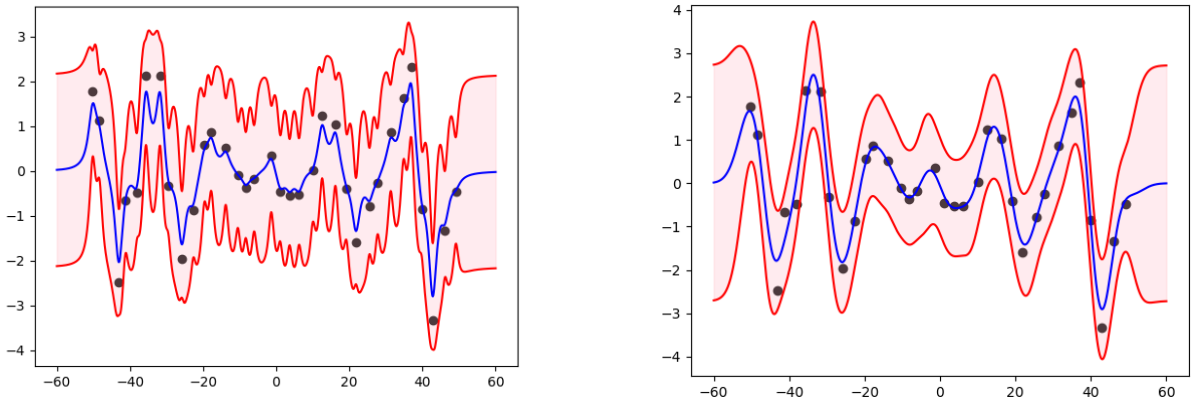
**(c)** $\alpha = 1, \beta = 5, \ell = 1, \sigma^2 = 1$

**(d)** $\alpha = 1, \beta = 5, \ell = 10, \sigma^2 = 1$

**Figure 4:** Comparison of GPR with different configurations of $\ell$

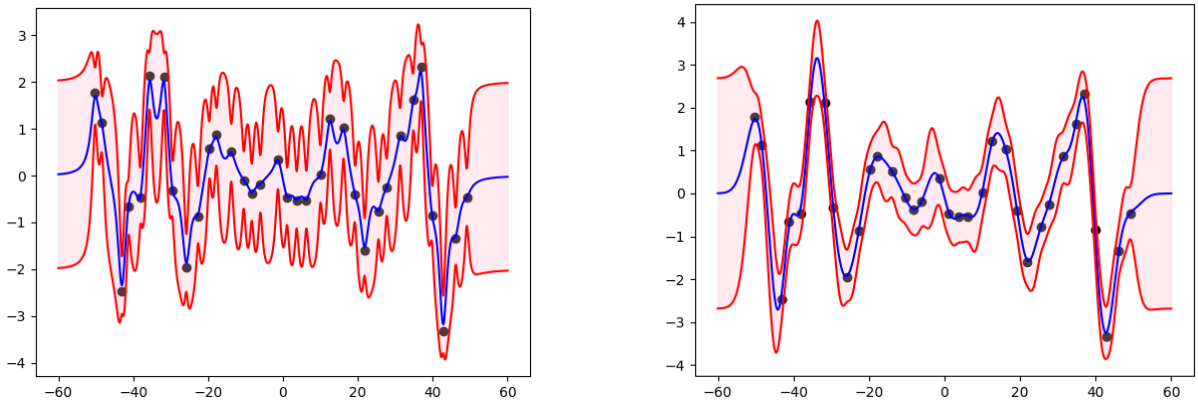
Now that we've realised that some parameters have more influence than others, let's take a look at their impact once they've been optimised by minimizing negative marginal log-likelihood.

By adjusting the model parameters, we can see a clear improvement in the fit of the model to the training data, leading to more accurate predictions on new data. This is especially true for the confidence interval. As for the blue line, i.e. the prediction mean, the difference is minimal. And greater is $\beta$, better is the prediction with optimised parameters.



**(a)** $\alpha = 1, \beta = 5, \ell = 1, \sigma^2 = 1$ **(b)** $\alpha = 508, \beta = 5, \ell = 3.32, \sigma^2 = 1.73$

**Figure 5:** Comparison of GPR with different with and without optimization



**(a)** $\alpha = 1, \beta = 20, \ell = 1, \sigma^2 = 1$ **(b)** $\alpha = 508, \beta = 5, \ell = 3.32, \sigma^2 = 1.73$

**Figure 6:** Comparison of GPR with different with and without optimization

**Training time :**

Regarding training time, GPR requires a high investment. Obviously, the required time depends of the number of data and the value of parameters, mainly the number of value to predict. Here, we've only used 1000 points, which is very fast, but beyond that the cost becomes considerable.

## 3.2 SVM on MNIST dataset

**Dataset :**

The data used in this study is derived from the MNIST dataset, which is extensively employed for handwritten character recognition and image classification tasks. This dataset comprises images of handwritten digits from zero to nine, each represented as a

matrix of grayscale pixels. These images serve as digital representations of handwritten digits captured under various conditions, including diverse writing styles, different sizes, and varying levels of noise. In our case, we have only used digits from 0 to 4.

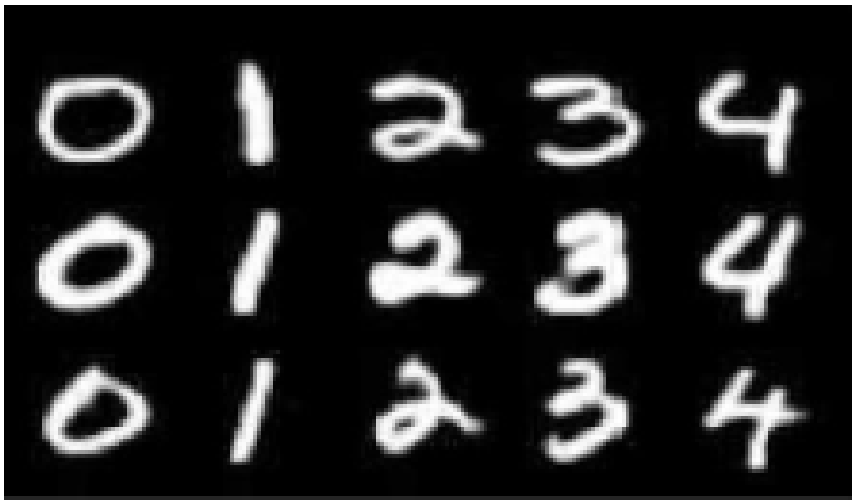


**Figure 7:** Dataset for the Support Vector Machines method

**1st case : Linear, polynomial and RBF kernels**

When we compute the 1st case, we can compare easily the different results. And as the table shows, with the default parameters, the results of the linear method and the RBF are very good, unlike the polynomial method. This is why it will be interesting to apply the C-SVC algorithm to optimise its parameters.

| Kernel type | Default Hyper-parameters | Test Accuracy |
|---|---|---|
| Linear | | 95.08% |
| Polynomial | -d 3 -g 1/784 -r 0 | 34.68% |
| RBF | -g 1/784 | 95.32% |

**2nd case : C-SVC**

As expected, the results with parameter optimisation are much better than without. Although it's minimal for the linear and RBF methods, but their results were already very good (>95%), the results for the polynomial kernel are just as good, if not better than the other kernels. This means that this type of kernel is sensitive to hyper-parameters, unlike the others. What's more, the prediction accuracy on the test set is even worse than the Cross-Validation predictions.

| Kernel type | Optimal Hyper-parameters | C-V Accuracy | Test Accuracy |
|---|---|---|---|
| Linear | -c 0.01 | 96.84% | 95.96% |
| Polynomial | -c 0.1 -d 3 -g 1 -r 10 | 98.18% | 97.92% |
| RBF | -c 10 -g 0.01 | 98.18% | 98.2% |
| Sigmoid | -c 10 -g 1/784 -r 0 | 97.04% | 95.32% |

**3rd case : Linear kernel + RBF kernel**

Having defined a new kernel ourselves (linear kernel + RBF kernel), the results are much better than each one separately without parameter optimisation. For this reason, it would be interesting to try optimising its parameters to see what results we could obtain.

| Kernel type | Default Hyper-parameters | Test Accuracy |
|---|---|---|
| Linear + RBF | -g 1/784 | 95.96% |

Overall, we find that the model using C-SVC with polynomial and RBF kernels and optimised hyperparameters performs best in terms of test accuracy. The linear kernel also performs reasonably well with optimised hyperparameters. But, the polynomial kernel with default hyperparameters performs worst.

**Training time :**

However, regarding training time, C-SVC requires a high investment. The more cases there are to process and compare with Cross-Validation to find the optimal parameters, the more time it will take, whereas simply calling the svm_train and svm_predict functions requires very little investment. So, obviously, linear kernel is much faster than other kernels because it is the simplest one.

# 4 Conclusion

The implementation and comparative analysis of GPR and SVM have highlighted the significance of optimized hyper-parameters in machine learning methods. We observed that the better or worse fitting of model parameters will significantly influence its performance, even in cases where the initial results seemed satisfactory.

SVMs have proven to be particularly effective for classifying the MNIST dataset. However, the computational cost of hyper-parameters optimization is substantial when using methods like grid search.

On the other hand, GPRs have been highly useful for regression in cases where the data distribution is complex. It provides a probabilistic estimation of predictions, allowing for quantification of uncertainty associated with each prediction. However, GPRs can be also sensitive to the choice of hyper-parameters, especially the length scale and kernel variance.

In summary, the choice between SVMs and GPRs largely depends on the specific nature of the problem, data distribution, and computational constraints. In many cases, SVMs are a good choice for classification, while GPRs are better suited for regression.

# References

[1] SVM regression with libsvm – Xu Cui while(alive)learn;. (s. d.). Xu Cui while(alive)learn; – while(alive)learn;, https://www.alivelearn.net/?p=1083.

[2] Wang, K. (2022, 4 octobre). Introduction to Gaussian process regression, Part 1 : The basics. Medium, https://medium.com/data-science-at-microsoft/introduction-to-gaussian-process-regression-part-1-the-basics-3cb79d9f155f.

[3] Tibrewal, T. P. (2023, 1 juillet). Support Vector Machines (SVM) : An Intuitive Explanation. Medium, https://medium.com/low-code-for-advanced-data-science/support-vector-machines-svm-an-intuitive-explanation-b084d6238106.

[4] SVM with grid search parameters... (s. d.). Federico Nutarelli, https://www.federiconutarelliphd.com/post/svm-with-grid-search-parameters.

[5] libsvm/README at master · cjlin1/libsvm. (s. d.). GitHub, https://github.com/cjlin1/libsvm/blob/master/README.