\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Contents:

Preparations

1. Gaussian Process
2. Code

* Code for Task 1:
* Code for Task 2:

1. Experiments

* Experiment for Task 1
* Experiment for Task 2

1. Observations and discussion

* Observations and discussion for Task 1
* Observations and discussion for Task 2

1. SVM on MNIST
2. Code

* Code for Task 1:
* Code for Task 2:
* Code for Task 3:

1. Experiments

* Experiment for Task 1
* Experiment for Task 2
* Experiment for Task 3

1. Observations and discussion

* Observations and discussion for Task 1
* Observations and discussion for Task 2
* Observations and discussion for Task 3

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Preparations **(The code is in python)**

First step is to mount the drive from where I will access all the input files for my code:

0001.

from google.colab import drive

drive.mount("/content/drive")

Then I added a custom print function, preprint(), to save the outputs of print() in a txt file:

0002.

def perprint(message, end = None, file = rtf\_file):

    print(message, end = end)

    print(message, end = end, file = file)

1. Gaussian Process
2. Code

First thing is to import the packages needed for Task 1 and Task 2:

0003.

import numpy as np

from scipy.optimize import minimize

import matplotlib.pyplot as plt

import scipy.spatial.distance as ssd

We then load the **training** file “input.data” with numpy, and split it into X (Nx1) and Y (Nx1), each of shape 34x1:

0004.

""" Input path for I. Gaussian Process"""

input\_file\_path\_gaussian\_process = '/content/drive/…/input.data'

# Training data

input\_data = np.loadtxt(input\_file\_path\_gaussian\_process)

X = input\_data[:, 0].reshape(-1, 1)  # Input features

Y = input\_data[:, 1].reshape(-1, 1)  # Observed values

* Code for Task 1:

We need to first define the function to compute the rational quadratic kernel. This function will take as inputs: 2 column vectors X1 (Nx1) and X2 (Mx1), and the kernel parameters sigma, alpha and the length scale *l*.

It will be used to compute the similarity between the values within X1 and X2.

is the squared distance between .

The kernel , will be translated into code that will **return** a kernel (NxM) as:

0005.

def Rational\_Quadratic\_Kernel(X1, X2, sigma, alpha, length\_scale):

    square\_distance = ssd.cdist(X1, X2, 'sqeuclidean') # (x\_i-x\_j)^2

    kernel = (sigma \*\* 2) \* np.power( 1 + square\_distance/( 2\*alpha\*length\_scale\*\*2), -alpha )

    return kernel

Then we can define the function to perform Gaussian Process Regression using the defined kernel above.

This function takes as inputs: column vectors X and Y, a new set of values named new\_X (a column vector Mx1, M => 1), beta (precision of the noises), and the parameters of the kernel.

, where .

, where C (NxN) *is the covariance of* Y(N\*1). mean: Mx1

, where . : MxM

, where Covariance : MxM

The function will **return** the mean and the covariance matrices of f on the new data points new\_X as:

0006.

def Gaussian\_Process\_Regression(X, Y, new\_X, beta, sigma, alpha, length\_scale):

    """ Gaussian process regression with rational quadratic kernel. """

    K\_train\_train = Rational\_Quadratic\_Kernel(X, X, sigma, alpha, length\_scale) # Kernel(X, X)

    C = K\_train\_train + (1 / beta) \* np.eye(len(X)) # Covariance matrix

    inv\_C = np.linalg.inv(C) # Inverse of covariance matrix

    K\_train\_new = Rational\_Quadratic\_Kernel(X, new\_X, sigma, alpha, length\_scale) # Kernel(X, new\_X)

    K\_new\_new = Rational\_Quadratic\_Kernel(new\_X, new\_X, sigma, alpha, length\_scale) # Kernel(new\_X, new\_X)

    K\_new\_new += (1 / beta) \* np.eye(len(new\_X)) # Kernel(new\_X, new\_X) with 1/beta added its diagonal elements

    mean\_new = np.dot(K\_train\_new.T, np.dot(inv\_C, Y)) # Mean of new\_X

    cov\_new = K\_new\_new - np.dot(K\_train\_new.T, np.dot(inv\_C, K\_train\_new)) # Covariance of new\_X

    return mean\_new, cov\_new

To visualize the Gaussian Process Regression in the range of [-60, 60], which is the range for the values in new\_X, we will define a function that takes as inputs: X and Y, new\_X, its mean and covariance matrices, as well as the parameters of the kernel function.

Since new\_X and mean(new\_X) are each Mx1, the plotting is straightforward with (new\_X, mean) as (Xs, Ys).

For the variance of each element of new\_X, we take the diagonal of covariance(new\_X), which will have M elements that will match each an element of new\_X.

So, variances(new\_X) has M elements:

Then for the 95% confidence interval, we need to compute an interval for M elements (will match with M mean values, or M new\_X’s elements in other words):

, which will have M elements.

The 95% confidence interval of f is the space between (mean – interval, mean + interval) over new\_X.

This function has **no return** and uses matplotlib.pyplot (plt) for the visualization.

0007.

def Visualize\_Gaussian\_Process(X\_train, Y\_train, new\_X, mean\_new, cov\_new, sigma, alpha, length\_scale):

    plt.figure(figsize=(10, 6))

    # Plot training data points

    plt.scatter(X\_train, Y\_train, color='k', label='Training Data')

    # Draw a line to represent mean of f in range [-60,60].

    plt.plot(new\_X, mean\_new, color='b', label='Mean of f')

    interval = 1.96 \* np.sqrt(np.diag(cov\_new))

    new\_X = new\_X.ravel()

    mean\_new = mean\_new.ravel()

    # Mark the 95% confidence interval of f.

    plt.fill\_between(new\_X, mean\_new + interval, mean\_new - interval, color='r', alpha=0.3, label='95% Confidence Interval of f')

    plt.legend()

    plt.xlim(-60, 60)

    plt.xlabel('X')

    plt.ylabel('f(X)')

    plt.title('Gaussian Process Regression with Rational Quadratic Kernel\n'

          f'sigma: {sigma:.5f}, alpha: {alpha:.5f}, length scale: {length\_scale:.5f}')

    #plt.grid(True)

    plt.show()

* Code for Task 2:

In this Task 2, we first need the function to compute the Negative Marginal Log-Likelihood.

Task 2 will make use of all the functions in Task1.

The function to define will take as inputs: params (list of parameters of the kernel function in code block 000.5), the column vectors X and Y, as well as the precision beta of the noises.

The formula for the Negative Marginal Log-Likelihood is:

The function will **return** the computed Negative Marginal Log-Likelihood.

0008.

def Negative\_Marginal\_Log\_Likelihood(params, X, Y, beta):

    sigma, alpha, length\_scale = params

    K\_train\_train = Rational\_Quadratic\_Kernel(X, X, sigma, alpha, length\_scale)  # Kernel(X, X)

    C = K\_train\_train + (1 / beta) \* np.eye(len(X)) # Covariance matrix

    inv\_C = np.linalg.inv(C) # Inverse of covariance matrix

    neg\_marg\_log\_lik = 0.5 \* len(X) \* np.log(2 \* np.pi)  # (k/2) \* log(2\*pi)

    neg\_marg\_log\_lik += np.sum(np.log(np.diagonal(np.linalg.cholesky(C)))) # 0.5\*log(det(C)), more  stable

    neg\_marg\_log\_lik += 0.5 \* np.dot(Y.T, np.dot(inv\_C, Y)) # (1/2)\*(Y.T).(inv\_C).Y

    return neg\_marg\_log\_lik

“”” Brief discussion of Task 2, commenting on the code block above:

I used “ 0.5\*log(det\_C) = np.sum(np.log(np.diagonal(np.linalg.cholesky(C))))”

to compute the log of the determinant of C because it is a more stable method compared to the alternative “0.5\*log(det\_C) = 0.5\*np.log(np.linalg.det(C))”.

Cholesky decomposition

The determinant of a positive definite matrix C can be calculated as follows:

, where , are the elements of the diagonal of ,

and is the lower triangular matrix obtained from the Cholesky decomposition C:

“””

**Back to the code for Task 2**

Then we can optimize the kernel parameters by using the minimize function from scipy.optimize.

minimize() requires the function to be minimized (Negative Marginal Log-Likelihood) in its inputs.

We also need to set bounds for each of the kernel parameters. Here, I set the lower bound for each of them to 1e-7 and the upper bound to None for each of the kernel parameters.

The minimize function **returns an object**, and we can access its x attribute to get the optimized parameters.

That function will **return** optimized, which is a 1-D numpy array with elements in the same position as our defined initial\_params list, which is a list of initial kernel parameters.

0009.

    initial\_params = [sigma, alpha, length\_scale]

    # Optimize kernel parameters

    optimized = minimize(Negative\_Marginal\_Log\_Likelihood, initial\_params,

                        bounds = ((1e-7, None), (1e-7, None), (1e-7, None)),

                        args = (X, Y, beta)).x  # Don't forget .x

    sigma = optimized[0]

    alpha = optimized[1]

    length\_scale = optimized[2]

1. Experiments

* Experiment for Task 1

Now that we have:

-the training data X and Y from code 0004,

-the rational quadratic kernel function from code 0005,

-the Gaussian Process Regression function from code 0006,

-and the visualization function for our Gaussian Regression Process from code 0007,

we can proceed to run Task 1 and visualize the results.

In the following code block:

* we first create the set of new data points new\_X, which will contain M = 1000 elements equally spaced in the range of [-60.0, 60.0].
* We then set the value of beta to 5 and each of the 3 kernel parameters are set to 1.
* Then we run Gaussian\_Process\_Regression() to get the mean and covariance matrices for new\_X, and we run Visualize\_Gaussian\_Process() to visualize the mean of f on new\_X as well as the 95% confidence interval of f.

0010.

    perprint("   Task 1: \n")

    # Get new points in range [-60,60]

    new\_X = np.linspace(-60.0, 60.0, 1000).reshape(-1, 1)

    # Parameter beta

    beta = 5

    # Kernel parameters

    sigma = 1

    alpha = 1

    length\_scale = 1

    # Perform Gaussian Process Regression

    mean\_new, cov\_new = Gaussian\_Process\_Regression(X, Y, new\_X, beta, sigma, alpha, length\_scale)

    # Visualize the result

    Visualize\_Gaussian\_Process(X, Y, new\_X, mean\_new, cov\_new, sigma, alpha, length\_scale)

Results of Task 1:

We can see the mean of f, the blue curve, the training data points in black, and the 95% confidence interval of f in light red.

Figure\_Task\_1

A graph with red and blue lines

Description automatically generated

* Experiment for Task 2

We have seen previously in "Code for Task 2":

- how to compute the negative marginal log-likelihood in code block 0008,

- and how to minimize it with respect to the kernel parameters in code block 0009.

We can now proceed to optimize the kernel parameters by minimizing the negative marginal log-likelihood.

In the following code block:

* I create a list containing the initial kernel parameters, which are all set to 1, same values as in Task 1.
* I then run the minimize function, setting the lower bound for each of the kernel parameters to 1e-7, and the **upper bound to None for each of them**.
* Then I access the “x” attribute of the object returned by minimize and assign the elements it contains to the kernel parameters.
* The optimized kernel parameters are:
* gamma = 1.31276
* alpha = 1735.74117
* length\_scale = 3.32055
* Finally, I perform Gaussian Process Regression again with the optimized kernel parameters and visualize the results.

0011.

    ###

    perprint("\n   Task 2: \n")

    # Initial kernel parameters

    initial\_params = [sigma, alpha, length\_scale]

    # Optimize kernel parameters

    optimized = minimize(Negative\_Marginal\_Log\_Likelihood, initial\_params,

                        bounds = ((1e-7, None), (1e-7, None), (1e-7, None)),

                        args = (X, Y, beta)).x  # Don't forget .x

    sigma = optimized[0]

    alpha = optimized[1]

    length\_scale = optimized[2]

    # Perform Gaussian Process Regression

    mean\_new, cov\_new = Gaussian\_Process\_Regression(X, Y, new\_X, beta, \*optimized)

    # Visualize the result

    Visualize\_Gaussian\_Process(X, Y, new\_X, mean\_new, cov\_new, sigma, alpha, length\_scale)

Results of Task 2:

We can see the mean of f, the blue curve, the training data points in black, and the 95% confidence interval of f in light red.

We can also notice the optimized parameters displayed in the title, each different from 1:

* gamma = 1.31276
* alpha = 1735.74117
* length\_scale = 3.32055

Figure\_Task\_2

A graph with red and blue lines

Description automatically generated

1. Observations and discussion

* Observations and discussion for Task 1

Observations for Task 1

In the results of Task 1, Figure\_Task\_1:

* We see that the mean of f (blue curve) is not very smooth along the training data points.
* We notice that the curve of the mean of f doesn’t pass through a certain number of the training data points, but in general it seems to be overfitting the training data.
* We also notice that neighboring points have variances that tend to sometimes be very different and higher , especially near regions where the mean of f (blue curve) changes direction even if slightly.
* We can tell by the fact that the upper bound line and the lower bound line of the 95% confidence interval of f is an exaggerated reflection of the mean of f around those regions where the mean of f changes direction, which regions also happen to have a wider 95% confidence interval compared to neighboring regions without a change in the direction of the mean of f.

Discussions for Task 1

Implications of the observations:

* The non-smoothness of the mean curve along training data points as well as the fact that it does not go through a certain number of training data points could indicate that the chosen kernel parameters produce a kernel, and thus a Gaussian Process Regression model, that does not have a good level of complexity to best fit and capture the underlying structure of the training data.
* The variability in the predicted variances might indicate that the Gaussian Process Regression model, through its kernel parameters, is more uncertain (less confident) about predictions in regions where the mean of f changes direction. It suggests that the model is having difficulty generalizing or making confident predictions in these specific areas.

These observations and discussions prompt the need for optimizing the Gaussian Process Regression model, which can mean optimizing the kernel parameters.

In the computation of the kernel, in code block 0005 (function for the rational quadratic kernel), we used

“ square\_distance = scipy.spatial.distance .cdist(X1, X2, 'sqeuclidean')” to compute [(x\_i- x\_j)]^2, which is the squared distance between x\_i and x\_j.

There are other methods for computing that squared distance with numpy:

0012.

square\_distance = np.sum(X1 \*\* 2, axis = 1).reshape(-1, 1) + np.sum(X2 \*\* 2, axis=1) - 2 \* X1 @ X2.T

This alternative method is **actually faster**, which can be noticed when using other kernels like the radial basis function kernel.

* Observations and discussion for Task 2

Observations for Task 2

In the results of Task 2, Figure\_Task\_2:

* After optimizing for the kernel parameters, we went from 1 as the value of each in Task 1, to
* gamma = 1.31276
* alpha = 1735.74117
* length\_scale = 3.32055
* By rerunning the code several times, the optimal parameters remain the same.
* alpha especially is very different from its original value of 1, and that is significant on the model as it is at the denominator position in the rational quadratic kernel.
* We see that the mean of f (blue curve) is a lot smoother along the training data points, compared to the results of Task 1.
* We notice that the curve of the mean of f doesn’t pass through a certain number of the training data points, but it does not seem to be overfitting the training data points like in Task 1.
* We also notice that neighboring points have variances that are closer to each other over most of the graph, except in the last section in the range of [45.0, 55.0] where there was a momentary loss of relativeness in the bottom part of the confidence interval.
* Overall, the 95% confidence interval in Task 2 is mirroring the mean of f more closely than in Task 1, and is also much smoother, like the mean of f.
* One more observation is that the confidence interval is a lot smaller in regions when the trend of the mean of f remains constant, meaning that the mean of f is going up or down on a relatively extended range of x values.

Discussion for Task 2

Implications of the observations:

* Stability of Optimal Kernel Parameters: The fact that the optimal parameters remain the same upon multiple runs indicates stability in the optimization process.
* The smoothness of the mean of f and of the 95% confidence interval in Task 2 suggests that not delimiting the upper bound for the kernel parameters while using the minimize function was the right course of action. The parameters had to be positive, so they all got a small enough lower bound value close to zero, 1e-7, but not too close to zero to avoid computation instabilities or errors.
* The hyperparameter alpha in the rational quadratic kernel controls the inverse scale of the length scale parameter. A large alpha implies that the model becomes less sensitive to variations in the input space. In other words, the model becomes smoother and more influenced by the overall trends in the data rather than individual data points.
* The change in alpha indicates that, after optimization, the model is favoring a smoother and more general pattern, mean of f, in the data rather than capturing finer details.
* The fact that neighboring points have variances that are more consistent over most of the graph indicates that the model is providing more uniform uncertainty estimates. This suggests better uncertainty calibration and a more reliable representation of model confidence.
* The momentary loss of relativeness in the variances in the last section of the graph could be indicative of a localized anomaly or uncertainty in that region. This might be an area where the model struggles to make accurate predictions.
* The closer mirroring of the confidence interval to the mean of f in Task 2 compared to Task 1 implies that the uncertainty estimates are more aligned with the actual variability in the data. **This is a positive indication of improved modeling.**
* The smaller 95% confidence intervals in regions where the trend of the mean of f remains constant suggest that the model is more certain about its predictions in stable regions. This reflects a better understanding of the data in areas where the trend is consistent.

In summary, the optimized parameters lead to a more stable and better-performing Gaussian Process Regression model. The smoothing effect from the adjusted alpha parameter contributes to a more reliable and generalized representation of the underlying function, reducing overfitting. The improved calibration of uncertainty estimates is evident in the closer alignment of the confidence interval with the mean of f. Overall, these observations suggest that the model is providing more accurate and stable predictions across different regions of the input space.

1. SVM on MNIST

“””Brief observation for Task 1 to Task 3:

The first thing before starting is to install libsvm to make it available on the virtual machine:

0013.

pip install libsvm-official

“””

1. Code

* Code for Task 1:

We begin Task I by importing all the packages that will be needed throughout Task 1 to task 3.

In addition to the packages imported in code block 0003, we will also need to import libsvm to work with support vector machines.

0014.

from libsvm.svmutil import \*

The next step is to load the training and testing data:

X\_train is 5000x784, Y\_train is 5000x1, X\_test is 2500x784 and Y\_test is 2500x1.

I make sure to convert Y\_train and Y\_test into integer type, even if they already are.

“””Brief discussion for Task 1:

The reason for converting to integer type explicitly is that they can be loaded as floating points.

“””

Below is the code.

0015.

""" Input paths for II. SVM on MNIST dataset"""

X\_train\_file\_path = "/content/drive/…/X\_train.csv"

Y\_train\_file\_path = "/content/drive/…/Y\_train.csv"

X\_test\_file\_path = "/content/drive/…/X\_test.csv"

Y\_test\_file\_path = "/content/drive/…/Y\_test.csv"

# Load training and testing data

X\_train = np.loadtxt(X\_train\_file\_path, delimiter=',')

Y\_train = np.loadtxt(Y\_train\_file\_path)

X\_test = np.loadtxt(X\_test\_file\_path, delimiter=',')

Y\_test = np.loadtxt(Y\_test\_file\_path)

# Convert Y\_train and Y\_test to integers (if not already)

Y\_train = Y\_train.astype(int)

Y\_test = Y\_test.astype(int)

After loading the data, we will define a dictionary that will contain information for the three types of kernels that will be used in Task 1.

The variables ‘-t variable\_number’ is a parameter that is used along with the libsvm library. ‘t’ is for ‘kernel type’ and ‘variable\_number is the specific number that indicates which kernel type will be used in the selected libsvm function.

So, the dictionary below shows the association.

0016.

# Define kernel types

kernel\_types = {'linear':'-t 0', 'polynomial':'-t 1', 'radial basis function':'-t 2'}

As mentioned above, when using libsvm there are a number of parameters aside from the training and testing data that must be defined and selected.

- s 0 is for the SVM (Support Vector Machine) type being set to C-SVM (soft-margin SVM) and comes with a cost c that is used to control how much misclassification are penalized.

param will be extracted from our kernel\_types dictionary, the ‘-t variable\_number’ variables.

0 -- linear:

1 -- polynomial:

2 -- radial basis function:

In our use case in Task 1, we will use the parameters defined in the below code block.

0017.

    # Set SVM parameters

    parameters = f'-s 0 {param} -c 10 -q'

    """ (-s default = 0) is for the SVM type, and we will set to -s 0 for C-SVM

        -q for quite mode.

        (-c value = large\_value) to use hard margin

        (-d default = 3) degree in polynomial kernel

        (-r coef0: default 0) coefficient in polynomial kernel

        (-g default = 1/num\_features) gamma in rbf and polynomial kernels

        (-c default = 1.0)

    """

The libsvm library has a function called svm\_train() that is used to train a SVM model. The function takes the training data and the parameters defined above as inputs, and **returns** a model.

The library also has a function svm\_predict() that is used to make predictions by taking the testing data, the model from svm\_train() and other parameters like ‘-q’ which is optional to avoid seeing what is happening inside svm\_predict() as it runs. It will return:

* **p\_label:** list of predicted labels
* **p\_acc:** tuple containing accuracy (for classification), mean-squared error, and squared correlation coefficient (for regression), **in this order**. The data type of each is a float.

p\_acc[0] is the accuracy value of the classification model (that is what we need here).

* **p\_vals:** list of decision values or probability estimates. If '-b 1' is specified, the list contains decision values for binary-class SVMs. If '-b 0' is specified, the list contains probability estimates for multi-class SVMs. The data type of each is a float.

In our case, b is not specified because we will not make use of p\_vals.

“””Brief observation for Task 1:

You must make sure to place the target data Y before the features data X in both functions.

“””

Below is the code block showing how to create the trained model and make predictions.

0018.

    # Train the SVM model: Y comes before X

    model = svm\_train(Y\_train, X\_train, parameters)

    # Make predictions and get the accuracy

    p\_label, p\_acc, p\_vals = svm\_predict(Y\_test, X\_test, model, '-q')

By running code blocks 0017 and 0018 in a loop, we will store the accuracy of the SVM based on the kernel type used in a list.

Here is how it will run:

0019.

accuracies = []

for k, param in kernel\_types.items():

# 0017 and 0018 code blocks

accuracies.append(p\_acc[0])

For the comparison of the performance of each kernel type in the SVM, we can simply extract the indexes from the accuracies list, and use the indexes to find out the corresponding kernel in the kernel\_types dictionary.

The code is as follows:

0020.

perprint("\n\tRanking all the kernels in Task 1:")

# Create a list of tuples containing kernel type and its accuracy

kernel\_accuracies = list(zip(kernel\_types.keys(), accuracies))

# Sort the list of tuples based on accuracy in descending order

sorted\_kernel\_accuracies = sorted(kernel\_accuracies, key=lambda x: x[1], reverse=True)

# Print the sorted kernels with ranking numbers and their accuracies

for rank, (kernel, accuracy) in enumerate(sorted\_kernel\_accuracies, start=1):

    perprint(f"\t\tRank {rank}: {kernel}: Accuracy - {accuracy:.2f} %")

* Code for Task 2:

The main part of Task 2 is the Grid\_Search function, which will allow us to find the best values for the parameters of our SVM, based on the type of kernel chosen.

The svm\_train() function enables grid search.

The -v parameter in variable “parameters” specifies the number of folds to use for cross-validation. We will use -v 3 for Task 2.

When -v is used, the svm\_train() function only returns the accuracy value, which I will saved in an accuracies array for each combination of parameters.

* The optimal parameters will be obtained by extracting the index of the maximum value inside the accuracies array of each case and link that index to the corresponding parameter values.

In the Grid\_Search() function, we need to input variables costs, gammas, degrees and coefficients.

Those variables are lists of the set of values that we want to be used for the cross-validation and determine which combination of their elements is optimal for each kernel type’s SVM.

The parameters in the variable “parameters” that goes into svm\_train() **are different based** **on** the kernel chosen.

So, we will have different parameters for each kernel type used in cross-validation.

The common parameters will be -s 0, -v 3, -q and -c (cost).

* Parameters specific to the ‘linear’ kernel: -c
* Parameters specific to the ‘radial basis function’ kernel: -c and -g.
* Parameters specific to the ‘polynomial’ kernel: -c, -g, -d and -r

Note that the cross-validation is only performed on the training data, so we will need to compute the accuracy of the optimized parameters on the testing data later.

Based on that understanding of the parameters to optimize and how to use the svm\_train() function for cross-validations, the following function was created to handle the three cases with if statements.

For each case, it will return the optimized parameters specific to each case and the optimal accuracy associated with the case during cross-validation.

Now, below is the complete Grid\_Search() function with explicit variable naming:

0021.

# Define Grid Search

def Grid\_Search(costs, gammas, degrees, coefficients, kernel, param, X\_train, Y\_train):

    if kernel == 'linear':

        accuracies = np.zeros( len(costs) )

        for i in range(len(costs)):

            parameters = f'-s 0 {param} -v 3 -c {costs[i]} -q'

            accuracies[i] = svm\_train(Y\_train, X\_train, parameters)

        accuracy\_optimal = np.max(accuracies)

        argmax\_index = np.argmax(accuracies)

        cost\_optimal = costs[argmax\_index]

        return cost\_optimal, accuracy\_optimal

    elif kernel == 'radial basis function':

        accuracies = np.zeros(( len(costs), len(gammas) ))

        for i in range(len(costs)):

            for j in range(len(gammas)):

                parameters = f'-s 0 {param} -v 3 -c {costs[i]} -g {gammas[j]} -q'

                accuracies[i, j] = svm\_train(Y\_train, X\_train, parameters)

        accuracy\_optimal = np.max(accuracies)

        argmax\_indices = np.unravel\_index(np.argmax(accuracies), accuracies.shape)

        c\_index, g\_index = argmax\_indices

        cost\_optimal = costs[c\_index]

        gamma\_optimal = costs[g\_index]

        return cost\_optimal, gamma\_optimal, accuracy\_optimal

    elif kernel == 'polynomial':

        accuracies = np.zeros(( len(costs), len(gammas), len(degrees), len(coefficients) ))

        for i in range(len(costs)):

            for j in range(len(gammas)):

                for k in range(len(degrees)):

                    for p in range(len(coefficients)):

                        parameters = f'-s 0 {param} -v 3 -c {costs[i]} -g {gammas[j]} -d {int(degrees[k])} -r {coefficients[p]} -q'

                        accuracies[i, j, k, p] = svm\_train(Y\_train, X\_train, parameters)

        accuracy\_optimal = np.max(accuracies)

        argmax\_indices = np.unravel\_index(np.argmax(accuracies), accuracies.shape)

        c\_index, g\_index, d\_index, r\_index= argmax\_indices

        cost\_optimal = costs[c\_index]

        gamma\_optimal = costs[g\_index]

        degree\_optimal = int(degrees[d\_index])

        coef\_optimal = coefficients[r\_index]

        return cost\_optimal, gamma\_optimal, degree\_optimal, coef\_optimal, accuracy\_optimal

    else:

      perprint("\n\tYou didn't give a valid input. \n")

Before running the Grid\_Search function, we will define a nested dictionary to store the optimal SVM parameters and the corresponding accuracy associated with each kernel type:

0022.

results = {

    'linear': {'accuracy': None, 'cost': None},

    'polynomial': {'accuracy': None, 'cost': None, 'gamma': None, 'degree': None, 'coefficient': None},

    'radial basis function': {'accuracy': None, 'cost': None, 'gamma': None}

}

Finally we will run the Grid\_Search() function in a for loop for all 3 types of kernels and update our nested dictionary. **Note that the final updated accuracies in the nested dictionary are computed on the testing data.**

Here is the code for:

0023.

for kernel, param in kernel\_types.items():

    if kernel == 'linear':

        cost\_optimal, accuracy\_optimal = Grid\_Search(costs, gammas, degrees, coefficients, kernel, param, X\_train, Y\_train)

        new\_values = {'accuracy': accuracy\_optimal, 'cost': cost\_optimal}

        results[kernel].update(new\_values)

        parameters = f'-s 0 {param} -c {cost\_optimal} -q'

        model = svm\_train(Y\_train, X\_train, parameters)

        \_, p\_acc, \_\_ = svm\_predict(Y\_test, X\_test, model, '-q')

        results[kernel]['accuracy'] = p\_acc[0]

    elif kernel == 'radial basis function':

        cost\_optimal, gamma\_optimal, accuracy\_optimal = Grid\_Search(costs, gammas, degrees, coefficients, kernel, param, X\_train, Y\_train)

        new\_values = {'accuracy': accuracy\_optimal, 'cost': cost\_optimal, 'gamma': gamma\_optimal}

        results[kernel].update(new\_values)

        parameters = f'-s 0 {param} -c {cost\_optimal} -g {gamma\_optimal} -q'

        model = svm\_train(Y\_train, X\_train, parameters)

        \_, p\_acc, \_\_ = svm\_predict(Y\_test, X\_test, model, '-q')

        results[kernel]['accuracy'] = p\_acc[0]

    elif kernel == 'polynomial':

        cost\_optimal, gamma\_optimal, degree\_optimal, coef\_optimal, accuracy\_optimal = Grid\_Search(costs, gammas, degrees, coefficients, kernel, param, X\_train, Y\_train)

        new\_values = {'accuracy': accuracy\_optimal, 'cost': cost\_optimal, 'gamma': gamma\_optimal, 'degree': degree\_optimal, 'coefficient': coef\_optimal}

        results[kernel].update(new\_values)

        parameters = f'-s 0 {param} -c {coef\_optimal} -g {gamma\_optimal} -d {degree\_optimal} -r {coef\_optimal} -q'

        model = svm\_train(Y\_train, X\_train, parameters)

        \_, p\_acc, \_\_ = svm\_predict(Y\_test, X\_test, model, '-q')

        results[kernel]['accuracy'] = p\_acc[0]

In this block, we use the nested dictionary to rank the kernels based on their performance on the testing data after grid search.

0024.

perprint("\tRanking all the kernels after grid search: ")

# Sort the kernels based on accuracy

sorted\_kernels = sorted(results.keys(), key = lambda k: results[k]['accuracy'], reverse=True)

# Print the sorted kernels with ranking numbers and their accuracies

for rank, kernel in enumerate(sorted\_kernels, start=1):

    perprint(f"\t\tRank {rank}: {kernel}: Accuracy - {results[kernel]['accuracy']:.2f} % ")

One last thing before ending Task 2 is to save the nested dictionary named results. It will avoid having to redo the grid search every time we run the code.

“””Brief observation for Task 2:

This is highly recommended.

“””

Back to code for Task 2.

0025.

# Numpy file path

numpy\_path = "/content/drive/…/grid\_search\_results.npy"

# Save the dictionary to a file

np.save(numpy\_path, results)

# Load the dictionary from the file

loaded\_results = np.load(numpy\_path, allow\_pickle = True).item()

* Code for Task 3:

In Task 3, we must start with defining the function for computing the linear kernel.

Linear kernel:

0026.

def linearKernel(X1, X2):

    kernel = np.dot( X1, X2.T )

    return kernel

We also define the function for computing the radial basis function (RBF) kernel.

radial basis function kernel:

I used scipy.spatial.distance.cdist() function to compute the square of the distance between *u* and *v*.

0027.

def RBFKernel(X1, X2, gamma):

    square\_distance = ssd.cdist(X1, X2, 'sqeuclidean')

    kernel = np.exp(-1 \* gamma \* square\_distance)

    return kernel

The next step after defining the functions in 0026 and 0027 is to create a precomputed kernel to be used with libsvm.

We will first create a new valid kernel that is the sum of the 2 above valid kernels in 0026 and 0027.

The precomputed (user-defined) kernel will then be created by adding a column of indexes (from 1 to length of X1 since the resulting sum of kernels is also of length “length of X1”) to the above new kernel.

The column of indexes will be in the first position.

The column of indexes will not be used for training, but it is required for running the svm\_train() function with a precomputed (custom) kernel.

Below is the code of that function.

0028.

def precomputed\_kernel(X1, X2, gamma):

    kernel\_linear = linearKernel(X1, X2)

    kernel\_RBF = RBFKernel(X1, X2, gamma)

    kernel = kernel\_linear + kernel\_RBF

    kernel = np.hstack(( np.arange(1, len(X1)+1).reshape(-1,1), kernel ))

    return kernel

Now it is time to train the SVM model.

We first compute the precomputed kernel (kernel\_train) based on the training features data X\_train.

Then we create an svm\_problem, which is a data structure that represents a Support Vector Machine (SVM) problem. It contains the training data and labels necessary for training an SVM model.

Svm\_problem() is used to create an object (prob in this case) that encapsulates the data and parameters needed for training an SVM model with the svm.train() function just like before in Task 1 and 2.

If isKernel is set to True, it indicates that kernel\_train is already a precomputed kernel matrix.

We must set ‘- t 4’ because we are using a precomputed kernel.

The cost -c and gamma -g don’t have to be specified in the svm\_train() parameters because they have no effect here (I tried).

0029.

# Train SVM with new kernel

kernel\_train = precomputed\_kernel(X\_train, X\_train, gamma)

prob = svm\_problem(Y\_train, kernel\_train, isKernel = True)

parameters = f'-s 0 -t 4 -q' # 4 for precomputed kernels

model = svm\_train(prob, parameters)

Then we test the trained model on the testing datasets.

kernel\_test is the precomputed kernel based on the testing input features and the training input features (SVM is a memory-based method).

It is not necessary to explicitly set isKernel during testing because the information about whether the kernel is precomputed or not was already conveyed during the training phase when the SVM model was built using the svm\_problem object.

We will obtain the accuracy of the model by accessing p\_acc[0] just like in the previous Task 1 and Task 2.

0030.

# Test SVM with new kernel

kernel\_test = precomputed\_kernel(X\_test, X\_train, gamma)

p\_label, p\_acc, p\_vals = svm\_predict(Y\_test, kernel\_test, model, '-q')

Finally, we add the new kernel named ‘linear + RBF’ to our existing “results”, a nested dictionary. That will make it easier to sort and rank the new kernel and the optimized 3 kernels of Task 2. So, we do the ranking as follows and print it out.

0031.

# Adding a new kernel in my dictionary

new\_kernel = 'linear + RBF'

results[new\_kernel] = {'accuracy': p\_acc[0], 'gamma': gamma}

perprint("\tRanking all the kernels: ")

# Sort the kernels based on accuracy

sorted\_kernels = sorted(results.keys(), key = lambda k: results[k]['accuracy'], reverse=True)

# Print the sorted kernels with ranking numbers and their accuracies

for rank, kernel in enumerate(sorted\_kernels, start=1):

    perprint(f"\t\tRank {rank}: {kernel}: Accuracy - {results[kernel]['accuracy']:.2f} %")

“””Brief observation for Task 3:

I run cross validation with different values of gamma (gammas), to see if the new kernel can be further optimized and then compared it again to the other optimized kernels.

“””

In the following code block, we optimized gamma of the new kernel using this method:

0032.

accuracies = np.zeros(len(gammas))

for i in range(len(gammas)):

    kernel\_train = precomputed\_kernel(X\_train, X\_train, gammas[i])

    prob = svm\_problem(Y\_train, kernel\_train, isKernel = True)

    parameters = f'-s 0 -t 4 -v 3 -q' # 4 for precomputed kernels

    accuracies[i] = svm\_train(prob, parameters)

argmax\_index = np.argmax(accuracies)

gamma\_optimal = costs[argmax\_index]

Then we compare the optimized new kernel with the other optimized three kernels after updating the nested results dictionary.

Here is the code block:

0033.

# Update gamma and the new accuracy for the new kernel

results[new\_kernel] = {'accuracy': p\_acc[0], 'gamma': gamma\_optimal}

perprint("\tRanking the optimized new kernel along with the kernels of Task 2: ")

# Sort the kernels based on accuracy

sorted\_kernels = sorted(results.keys(), key = lambda k: results[k]['accuracy'], reverse=True)

# Print the sorted kernels with ranking numbers and their accuracies

for rank, kernel in enumerate(sorted\_kernels, start=1):

    perprint(f"\t\tRank {rank}: {kernel}: Accuracy - {results[kernel]['accuracy']:.2f} % ")

**And that ends Task 3.**

1. Experiments

* Experiment for Task 1

Note:

We have seen in code for Task 1:

* code block 0015: How to load the training data.
* code block 0018: How to train a model and make predictions.
* code block 0020: How to rank the different kernels.

Using the defined kernel types of code block 0016, we run all the steps noted above.

Here are the results of running the code for Task 1: We can see how the kernels rank using hard-SVM.

A computer screen shot of a black screen

Description automatically generated

* Experiment for Task 2

Note:

We have seen in code for Task 2:

* code block 0021: How the Grid\_Search() function works.
* code block 0023: How to run the Grid\_Search() function with all the kernels
* code block 0024: How to rank the optimized kernels.

Here are the parameters costs, gammas, degrees, and coefficients used for the Grid Search.

0034.

costs = [0.001, 0.01, 0.1, 1.0, 10]

gammas = [0.001, 0.01, 0.1, 1.0]

degrees = [2, 3, 4]

coefficients = [0, 1.0, 2.0]

Here are the results of running the code for Task 2: We can see updated parameters for each kernel and the new ranking of the kernels.

A screenshot of a computer program

Description automatically generated

* Experiment for Task 3

Note:

We have seen in code for Task 3:

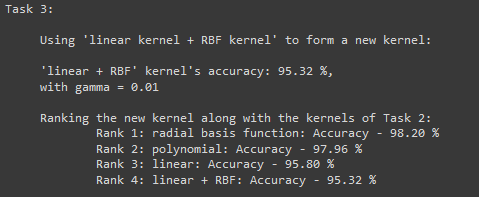
* code blocks 0026 ~ 0028: How to get a valid precomputed kernel, which is a 'linear + RBF' kernel formatted for using with libsvm.
* code blocks 0029 0030 0031: How to train, test and rank the precomputed kernel.
* code blocks 0032 0033: How to optimize the new kernel and rank it.

Based on the above note, we set the parameter gamma of the new kernel to the same value as the optimized gamma for the ‘radial basis function’ kernel in Task 2, using the nested results dictionary:

0035.

gamma = results['radial basis function']['gamma'] # 0.01

Here are the results of running the code for Task 3 (1): Train and rank the new kernel.



Here are the results of running the code for Task 3 (2): Optimize and rank the new kernel.



A screen shot of a computer

Description automatically generated

1. Observations and discussion

* Observations and discussion for Task 1

Observations for Task 1

Looking at code for Task 1 and the “results of running the code for Task 1”, I observe the followings:

* In Task 1, we used default values to train the SVM model based on different kernels. We also set a high cost '-c 10' to get results of a hard-margin SVM.
* Using such parameters for the kernels, we find that the polynomial kernel performs the worst (79.72 %), and the radial basis function (RBF) kernel performs the best (96.32 %). The linear kernel gives 95.00 % accuracy, closely in second place.
* Overall, only the polynomial kernel gives a not so favorable performance, but still kind of okay.

Discussion for Task 1

Implications of the observations of Task 1:

* The RBF kernel achieving the highest accuracy at 96.32% indicates that it captures the underlying patterns in the data effectively. This kernel is known for its ability to model complex relationships, making it a strong choice for a variety of problems.
* The linear kernel achieving 95.00% accuracy suggests that the data might have a somewhat linear or simple structure. Linear kernels work well when the decision boundary is relatively simple and can be effectively modeled with a linear function.

In conclusion, different hyperparameters for each kernel could potentially improve their performance.

* Observations and discussion for Task 2

Observations for Task 2

Looking at code for Task 2 and the “results of running the code for Task 2”, I observe the followings:

* The first observation is that due to the nature of cross validation, with multiple runs, Grid Search doesn't always give the same optimal kernel parameters for the kernels' optimization.

But the resulting accuracies are very similar, and the kernels’ ranking is the same.

* With the optimized kernel parameters, the performance of all the kernels improved, even if slightly (linear kernel).
* I notice that the RBF kernel still performs the best (98.20 %). But now the polynomial kernel is second (97.96) and the linear kernel is now last with slightly improved performance (95.96 % now vs 95.00 % before).
* Overall, the performance of the optimized kernels is very good.

Discussion for Task 2

Implications of the observations of Task 2:

* The observed variability in optimal kernel parameters in different runs of cross-validation is not uncommon. It suggests that the kernels' performance might be sensitive to the choice of the kernel parameters’ values, and small variations in the data or cross-validation splits can influence the optimal value.

Further investigation or experimentation could help determine the robustness of the model to different kernel parameters’ values.

* The fact that the rankings do not change much means that the variability in cross-validation splits is not that important for the overall accuracies.
* The fact that all kernels, including the linear one, show improved performance with optimized parameters underscores the importance of fine-tuning hyperparameters.
* The change in the ranking of kernel performances after optimization suggests that the dataset is more complex than simpler, meaning that the linear kernel cannot capture the underlying structure of the data as well as the other kernels.
* The overall performance of the optimized kernels is very good is encouraging.

In conclusion, the observations emphasize the importance of careful hyperparameter tuning and the impact it can have on the performance of SVM models with different kernels.

* Observations and discussion for Task 3

Observations for Task 3

Looking at code for Task 3 and the “results of running the code for Task 3 (1)”, I observe the followings:

* The trained dataset with the new kernel with parameter gamma equal to the optimized gamma of the RBF kernel, got a performance that is less than that of the other kernels (95.32 %).
* But the new kernel performs less than the optimized linear kernel.

Looking at code for Task 3 and the “results of running the code for Task 3 (2)”, I observe the followings:

* The optimized parameter gamma of the new kernel through cross-validation and the retrained dataset with the new kernel, achieves an improved performance, a slight improvement (95.64 % now vs 95.32 % before).
* But the optimal new kernel still performs less than all the other kernels.

Discussion for Task 3

Implications of the observations of Task 3 (1):

* This observation suggests that this setting might be a good one for the dataset, since the performance is pretty good, even if less than that of each individual kernel comprising it.

Implications of the observations of Task 3 (2):

* The slight improvement in performance with the optimized gamma for the new kernel indicates that careful parameter tuning can positively impact model accuracy. However, the fact that it still performs less than all the other kernels raises questions about the inherent suitability of this new kernel for the given dataset.
* The challenges in directly generalizing parameters between kernels underscore the unique characteristics of each kernel function, and that it is crucial to understand the behavior of the chosen kernel and experiment with parameters specifically tailored to its characteristics.

In conclusion, while hyperparameter tuning can lead to improvements, the choice of the kernel itself remains a critical factor in SVM performance. It's essential to understand the nuances of each kernel and how well it aligns with the data characteristics.