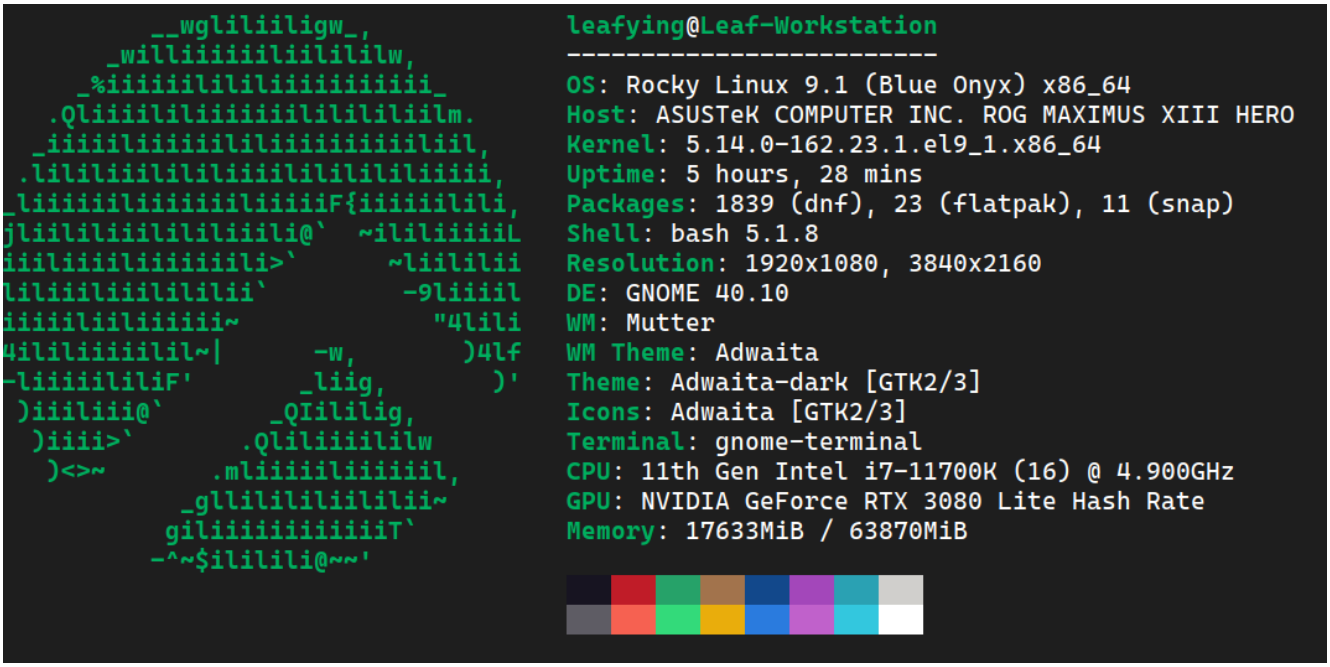
Machine Learning Homework 5

Environment

- OS



- Language: C++
- Standard: C++20

Gaussian Process

Code with detailed explanations

Libraries

Used library => Corresponding library in homework description

- Eigen => numpy
- OptimLib, autodiff => scipy.optimize
- Imgui, Implot => visualization
- Boost (for access file system only)
- OpenMP (for parallel acceleration)

Visualization

- **Setup**

  > Used to setup Imgui + OpenGL
  >
  > OpenGL is a backend for showing and drawing on window.

```cpp
static void glfwErrorCallback(int error, const char *description)
{
    std::cerr << "GLFW Error " << error << ": " << description << std::endl;
}

GLFWwindow *setUpGUI()
{
    glfwSetErrorCallback(glfwErrorCallback);
    if (!glfwInit())
    {
        return nullptr;
    }

    // Decide GL+GLSL versions
    // GL 3.3 + GLSL 330
    const char *glsl_version = "#version 330";
    glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
    glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
    glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE); // 3.2+ only
    glfwWindowHint(GLFW_OPENGL_FORWARD_COMPAT, GL_TRUE);           // 3.0+ only

    // Create window with graphics context
    GLFWwindow *window = glfwCreateWindow(1280, 720, TITLE.c_str(), nullptr, nullptr);
    if (window == nullptr)
        return nullptr;

    glfwMakeContextCurrent(window);
    glfwSwapInterval(0);

    // Setup Dear ImGui context
    IMGUI_CHECKVERSION();
    ImGui::CreateContext();
    ImPlot::CreateContext();

    ImGuiIO &io = ImGui::GetIO();

    ImFontConfig cfg;
    cfg.SizePixels = 15;
    io.Fonts->AddFontDefault(&cfg);

    // Setup Dear ImGui style
    ImGui::StyleColorsLight();
    // ImGui::StyleColorsDark();

    // Setup Platform/Renderer backends
    ImGui_ImplGlfw_InitForOpenGL(window, true);
    ImGui_ImplOpenGL3_Init(glsl_version);

    return window;
}
```

- **Draw result**

> Arguments
>
> title: plot title
>
> data: training data points
>
> f: line for Gaussian Process Regression with variance, f[0]: x, f[1]: y from means of conditional predictive distribution
>
> Setup Axes

```
ImPlot::SetupAxes("x", "y");
// set x-axis range
ImPlot::SetupAxisLimits(ImAxis_X1, -60, 60);
```

Plot training data points

```
// plot points with red color and circle
ImPlot::SetNextMarkerStyle(ImPlotMarker_Circle, IMPLOT_AUTO, COLOR_RED,
IMPLOT_AUTO, COLOR_RED);
ImPlot::PlotScatter("train data", data.col(0).data(), data.col(1).data(),
data.rows());
```

Plot the line for Gaussian Process Regression

```
// 95% confidence interval == std * 2
VectorXd variance = 2 * f.col(2);
// calculate the lower bound and upper bound of y
VectorXd upperBound = f.col(1) + variance;
VectorXd lowerBound = f.col(1) - variance;

// plot shade filled with blue color
ImPlot::SetNextFillStyle(COLOR_BLUE, 0.5f);
// Equivalent to fill_between of matplotlib
ImPlot::PlotShaded("f(x)'s 95%% confidence", f.col(0).data(),
upperBound.data(), lowerBound.data(), f.rows());

// plot line with black color
ImPlot::SetNextLineStyle(COLOR_BLACK);
ImPlot::PlotLine("f(x)'s mean", f.col(0).data(), f.col(1).data(), f.rows());
```

```
111 void drawPlot(const std::string &title, const MatrixX2d &data, const MatrixX3d &f)
112 {
113     if (ImPlot::BeginPlot(title.c_str()))
114     {
115         ImPlot::SetupAxes("x", "y");
116         ImPlot::SetupAxisLimits(ImAxis_X1, -60, 60);
117         // ImPlot::SetupAxisLimits(ImAxis_Y1, -60, 60);
118 
119         ImPlot::SetNextMarkerStyle(ImPlotMarker_Circle, IMPLOT_AUTO, COLOR_RED, IMPLOT_AUTO, COLOR_RED);
120         ImPlot::PlotScatter("train data", data.col(0).data(), data.col(1).data(), data.rows());
121 
122         VectorXd variance = 2 * f.col(2);
123         VectorXd upperBound = f.col(1) + variance;
124         VectorXd lowerBound = f.col(1) - variance;
125         ImPlot::SetNextFillStyle(COLOR_BLUE, 0.5f);
126         ImPlot::PlotShaded("f(x)'s 95%% confidence", f.col(0).data(), upperBound.data(), lowerBound.data(), f.rows());
127 
128         ImPlot::SetNextLineStyle(COLOR_BLACK);
129         ImPlot::PlotLine("f(x)'s mean", f.col(0).data(), f.col(1).data(), f.rows());
130 
131         ImPlot::EndPlot();
132     }
133 }
```

- **Show GUI**

> Main loop of Imgui

> Arguments
>
> data: training data points
>
> f: line for Gaussian Process Regression with variance, f[0]: x, f[1]: y from means of conditional predictive distribution
>
> optimizedF: line for Optimized Gaussian Process Regression with variance, f[0]: x, f[1]: y from means of conditional predictive distribution

```cpp
void showGUI(const MatrixX2d &data, const MatrixX3d &f, const MatrixX3d &optimizedF)
{
    auto window = setUpGUI();
    if (window == nullptr)
    {
        throw std::runtime_error("Cannot create window.");
    }

    // Our state
    auto clear_color = ImVec4(0.45f, 0.55f, 0.60f, 1.00f);

    // Main loop
    while (!glfwWindowShouldClose(window))
    {
        glfwPollEvents();

        // Start the Dear ImGui frame
        ImGui_ImplOpenGL3_NewFrame();
        ImGui_ImplGlfw_NewFrame();
        ImGui::NewFrame();

        {
            const auto windowSize = ImGui::GetIO().DisplaySize;

            ImGui::SetNextWindowPos(ImVec2(0, 0), ImGuiCond_Always);
            ImGui::SetNextWindowSize(ImVec2(windowSize.x, windowSize.y), ImGuiCond_Always);
            ImGui::Begin("Result", nullptr, ImGuiWindowFlags_NoDecoration);

            if (ImPlot::BeginSubplots("Result", 1, 2, ImVec2(windowSize.x, windowSize.y)))
            {
                drawPlot("Original", data, f);
                drawPlot("Optimized", data, optimizedF);
                ImPlot::EndSubplots();
            }

            ImGui::End();
        }

        // Rendering
        ImGui::Render();
        int display_w, display_h;
        glfwGetFramebufferSize(window, &display_w, &display_h);
        glViewport(0, 0, display_w, display_h);
        glClearColor(clear_color.x * clear_color.w, clear_color.y * clear_color.w, clear_color.z * clear_color.w, clear_color.w);
        glClear(GL_COLOR_BUFFER_BIT);
        ImGui_ImplOpenGL3_RenderDrawData(ImGui::GetDrawData());

        glfwSwapBuffers(window);
    }

    // Cleanup
    ImGui_ImplOpenGL3_Shutdown();
    ImGui_ImplGlfw_Shutdown();
    ImPlot::DestroyContext();
    ImGui::DestroyContext();

    glfwDestroyWindow(window);
    glfwTerminate();
}
```

Part1

- **Rational quadratic kernel**
  - Formula

    $k(x, x') = \sigma^2(1 + \frac{||x-x'||_2^2}{2\alpha\ell^2})^{-\alpha}$
  - Main kernel function (without $diff = ||x - x'||_2^2$)

> Arguments
> diff: data of $||x - x'||_2^2$
> kernelParameters: parameters of Rational quadratic kernel, σ^2, α, ℓ
>
> Part of formula
> $k(x, x') = \sigma^2(1 + \frac{diff}{2\alpha\ell^2})^{-\alpha}$

```cpp
template <typename DerivedA, typename DerivedB, typename Out = Eigen::Matrix<typename DerivedB::Scalar, DerivedA::RowsAtCompileTime, DerivedA::ColsAtCompileTime>>
Out calculateRationalQuadraticKernel(const Eigen::MatrixBase<DerivedA> &diff, const Eigen::MatrixBase<DerivedB> &kernelParameters)
{
    return (kernelParameters[0] *
            (1 + diff.array() / (2 * kernelParameters[1] * autodiff::detail::pow(kernelParameters[2], 2))).pow(-kernelParameters[1]));
}
```

- Vector `lhs` -> Vector `rhs` kernel function

> Arguments
> lhs: vector x with n instances
> rhs: vector x' with m instances
> kernelParameters: parameters of Rational quadratic kernel, σ^2, α, ℓ
>
> Part of formula
> $diff = ||\vec{x} - \vec{x}'||_2^2$

```cpp
template <typename DerivedA, typename DerivedB, typename Out = Eigen::Matrix<typename DerivedB::Scalar, Eigen::Dynamic, Eigen::Dynamic>>
Out calculateRationalQuadraticKernel(const Eigen::MatrixBase<DerivedA> &lhs, const Eigen::MatrixBase<DerivedA> &rhs, const Eigen::MatrixBase<DerivedB> &kernelParameters)
{
    Eigen::Matrix<typename DerivedA::Scalar, Eigen::Dynamic, Eigen::Dynamic> diff = (lhs.replicate(1, lhs.rows()).rowwise() - rhs.transpose()).array().pow(2);
    return calculateRationalQuadraticKernel(diff, kernelParameters);
}
```

- Vector `lhs` -> Scalar `rhs` kernel function

> Arguments
> lhs: vector x with n instances
> rhs: scalar x'
> kernelParameters: parameters of Rational quadratic kernel, σ^2, α, ℓ
>
> Part of formula
> $diff = ||\vec{x} - x'||_2^2$

```cpp
VectorXd calculateRationalQuadraticKernel(const VectorXd &lhs, double rhs, const Vector3d &kernelParameters)
{
    VectorXd diff = (lhs.array() - rhs).pow(2);
    return calculateRationalQuadraticKernel(diff, kernelParameters);
}
```

- Scalar `lhs` -> Scalar `rhs` kernel function

> Arguments
> lhs: scalar x
> rhs: scalar x'
> kernelParameters: parameters of Rational quadratic kernel, σ^2, α, ℓ
>
> Part of formula (for convenience, create a vector contained one scalar)
> $diff = ||x - x'||_2^2$

```cpp
double calculateRationalQuadraticKernel(double lhs, double rhs, const Vector3d &kernelParameters)
{
    VectorXd diff = VectorXd::Constant(1, std::pow(lhs - rhs, 2));
    return calculateRationalQuadraticKernel(diff, kernelParameters).value();
}
```

- **Gaussian Process Regression**
  - Formula

- Training with RBF kernel

$$p(\mathbf{y}|\theta) = \int p(\mathbf{y}|\mathbf{f})p(\mathbf{f})d\mathbf{f} = \mathcal{N}(\mathbf{y}|\mathbf{0}, \mathbf{C}_\theta)$$

$$\begin{aligned} \mathrm{C}_\theta(\mathbf{x_n}, \mathbf{x_m}) &= k_\theta(\mathbf{x_n}, \mathbf{x_m}) + \beta^{-1}\delta_{nm} \\ \mathrm{C}_\theta(\mathbf{x}, \mathbf{x}') &= K_\theta(\mathbf{x}, \mathbf{x}') + \beta^{-1}\mathbf{I_n} \end{aligned} \tag{2}$$

$$k_\theta(\mathbf{x}, \mathbf{x}') = \theta_0(1 + \frac{||\mathbf{x} - \mathbf{x}'||_2^2}{2\theta_1\theta_2^2})^{-\theta_1}$$

- Prediction

$$p(y^*|x^*, \mathbf{y}) = \mathcal{N}(y^*|\mu(\mathbf{x}^*), \sigma^2(\mathbf{x}^*))$$

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

- Main function

> Arguments
>
> data: training data points
>
> beta: hyperparameter for variance of error noise
>
> Check the comments in the figure to understand the whole procedure

```
314  void modelData(const MatrixX2d &data, double beta)
315  {
316      // variance, alpha, length scale
317      Vector3d kernelParameters = Vector3d::Constant(1);
318      // calculate C from p(y) = N(y|0, C)
319      MatrixXd covariance = calculateCovariance(data, beta, kernelParameters);
320
321      // Optimization
322      optim::algo_settings_t settings;
323      // settings.gd_settings.par_step_size = 1e-4;
324      settings.conv_failure_switch = 1;
325      settings.vals_bound = true;
326      settings.upper_bounds = Vector3d::Constant(1e5);
327      settings.lower_bounds = Vector3d::Constant(1e-5);
328      settings.print_level = 1;
329
330      // externel data used in loss term
331      LossArguments args{data, beta};
332      VectorXd optimizedKernelParameters = kernelParameters;
333      // optimize the kernel parameters by bfgs algorithm
334      optim::bfgs(optimizedKernelParameters, evaluateOptimFn, reinterpret_cast<void*>(&args), settings);
335
336      // calculate C after optimizing the kernel parameters
337      MatrixXd optimizedCovariance = calculateCovariance(data, beta, optimizedKernelParameters);
338
339      // sample data points from predictive distribution p(y*|y) with different covariance
340      MatrixX3d f = generatePointsOfLine(data, beta, covariance, kernelParameters);
341      MatrixX3d optimizedF = generatePointsOfLine(data, beta, optimizedCovariance, optimizedKernelParameters);
342
343      std::cout << "Optimized kernel parameters (variance, alpha, length scale)" << std::endl;
344      std::cout << optimizedKernelParameters << std::endl;
345
346      // Plot the results
347      showGUI(data, f, optimizedF);
348  }
```

- `calculateCovariance` function: calculate covariance $\mathbf{C}_\theta$ of marginal distribution $p(\mathbf{y})$

> Arguments
> data: training data points
> beta: hyperparameter for variance of error noise
> kernelParameters: parameters of Rational quadratic kernel, σ^2, α, ℓ
>
> Formula

$$\mathrm{C}_\theta(\mathbf{x}, \mathbf{x}') = K_\theta(\mathbf{x}, \mathbf{x}') + \beta^{-1}\mathbf{I_n} \tag{4}$$

```cpp
template<typename Derived, typename Out = Eigen::Matrix<typename Derived::Scalar, Eigen::Dynamic, Eigen::Dynamic>>
Out calculateCovariance(const MatrixX2d &data, double beta, const Eigen::MatrixBase<Derived> &kernelParameters)
{
    Out covariance = calculateRationalQuadraticKernel(data.col(0), data.col(0), kernelParameters);
    covariance.diagonal().array() += (1 / beta);
    return covariance;
}
```

- `generatePointsOfLine` function: sample data points from predictive distribution $p(y^*|\mathbf{y})$

> Formula

$$p(y^*|x^*, \mathbf{y}) = \mathcal{N}(y^*|\mu(\mathbf{x}^*), \sigma^2(\mathbf{x}^*))$$

$$\begin{aligned} \mu(\mathbf{x}^*) &= k(\mathbf{x}, \mathbf{x}^*)^\top \mathbf{C}^{-1}\mathbf{y} \\ \sigma^2(\mathbf{x}^*) &= k(\mathbf{x}^*, \mathbf{x}^*) + \beta^{-1} - k(\mathbf{x}, \mathbf{x}^*)^\top \mathbf{C}^{-1} k(\mathbf{x}, \mathbf{x}^*) \end{aligned} \tag{5}$$

> Note: $k(\mathbf{x}, \mathbf{x}^*)^\top \mathbf{C}^{-1}$ is a common part.

```cpp
MatrixX3d generatePointsOfLine(const MatrixX2d &data, double beta, const MatrixXd &covariance, const Vector3d& kernelParameters)
{
    MatrixXd inv_covariance = covariance.inverse();
    double inv_beta = 1 / beta;
    MatrixX3d f(250, 3);
    f.col(0) = VectorXd::LinSpaced(250, -60, 60);

    for (auto row : f.rowwise())
    {
        double k = calculateRationalQuadraticKernel(row[0], row[0], kernelParameters) + inv_beta;
        VectorXd kernel = calculateRationalQuadraticKernel(data.col(0), row[0], kernelParameters);

        Eigen::Matrix<double, 1, Eigen::Dynamic> common = kernel.transpose() * inv_covariance;
        row[1] = (common * data.col(1)).value();
        row[2] = k - (common * kernel).value();
    }

    return f;
}
```

Part2

- Main function (check the red rectangle)

> Arguments
> data: training data points
> beta: hyperparameter for variance of error noise
>
> Check the comments in the figure to understand the whole procedure

```cpp
void modelData(const MatrixX2d &data, double beta)
{
    // variance, alpha, length scale
    Vector3d kernelParameters = Vector3d::Constant(1);
    // calculate C from p(y) = N(y|0, C)
    MatrixXd covariance = calculateCovariance(data, beta, kernelParameters);

    // Optimization
    optim::algo_settings_t settings;
    // settings.gd_settings.par_step_size = 1e-4;
    settings.conv_failure_switch = 1;
    settings.vals_bound = true;
    settings.upper_bounds = Vector3d::Constant(1e5);
    settings.lower_bounds = Vector3d::Constant(1e-5);
    settings.print_level = 1;

    // externel data used in loss term
    LossArguments args{data, beta};
    VectorXd optimizedKernelParameters = kernelParameters;
    // optimize the kernel parameters by bfgs algorithm
    optim::bfgs(optimizedKernelParameters, evaluateOptimFn, reinterpret_cast<void*>(&args), settings);

    // calculate C after optimizing the kernel parameters
    MatrixXd optimizedCovariance = calculateCovariance(data, beta, optimizedKernelParameters);

    // sample data points from predictive distribution p(y*|y) with different covariance
    MatrixX3d f = generatePointsOfLine(data, beta, covariance, kernelParameters);
    MatrixX3d optimizedF = generatePointsOfLine(data, beta, optimizedCovariance, optimizedKernelParameters);

    std::cout << "Optimized kernel parameters (variance, alpha, length scale)" << std::endl;
    std::cout << optimizedKernelParameters << std::endl;

    // Plot the results
    showGUI(data, f, optimizedF);
}
```

- `evaluateOptimFn` function: calculate the gradients by using autodiff library

> Arguments
>
> parameters: parameters of Rational quadratic kernel, σ^2, α, ℓ
>
> grad_out: used to store the gradients
>
> args: external data

```cpp
double evaluateOptimFn(const VectorXd &parameters, VectorXd *grad_out, void *args)
{
    auto lossArguments = reinterpret_cast<LossArguments*>(args);
    autodiff::real u;
    autodiff::Vector3real paramtersd = parameters.eval();

    if (grad_out != nullptr)
    {
        // in order to use external data in autodiff library
        // create a lambda function to wrap the loss function
        auto lossFn = [lossArguments](const autodiff::Vector3real &_paramtersd)
        {
            return calculateLoss(_paramtersd, lossArguments->data, lossArguments->beta);
        };

        *grad_out = autodiff::gradient(lossFn, autodiff::wrt(paramtersd), autodiff::at(paramtersd), u);
    }
    else
    {
        u = calculateLoss(paramtersd, lossArguments->data, lossArguments->beta);
    }

    return u.val();
}
```

- `calculateLoss` function: calculate the loss value by negative log marginal likelihood

  > Arguments
  > parameters: parameters of Rational quadratic kernel, σ^2, α, ℓ
  > data: training data points
  > beta: hyperparameter for variance of error noise
  >
  > Formula
  >
  > $$\ln p(\mathbf{y}|\theta) = \frac{1}{2}\ln\det\mathbf{C}_\theta + \frac{1}{2}\mathbf{y}^\top\mathbf{C}_\theta{}^{-1}\mathbf{y} + \frac{N}{2}\ln(2\pi) \tag{6}$$

```
259  autodiff::real calculateLoss(const autodiff::Vector3real &parameters, const MatrixX2d &data, double beta)
260  {
261      autodiff::MatrixXreal covariance = calculateCovariance(data, beta, parameters);
262      return 0.5 * autodiff::detail::log(covariance.determinant())
263          + 0.5 * (data.col(1).transpose() * covariance.inverse() * data.col(1)).value()
264          + 0.5 * static_cast<double>(data.rows()) * autodiff::detail::log(2 * M_PI);
265  }
```

Experiments settings and results

Settings

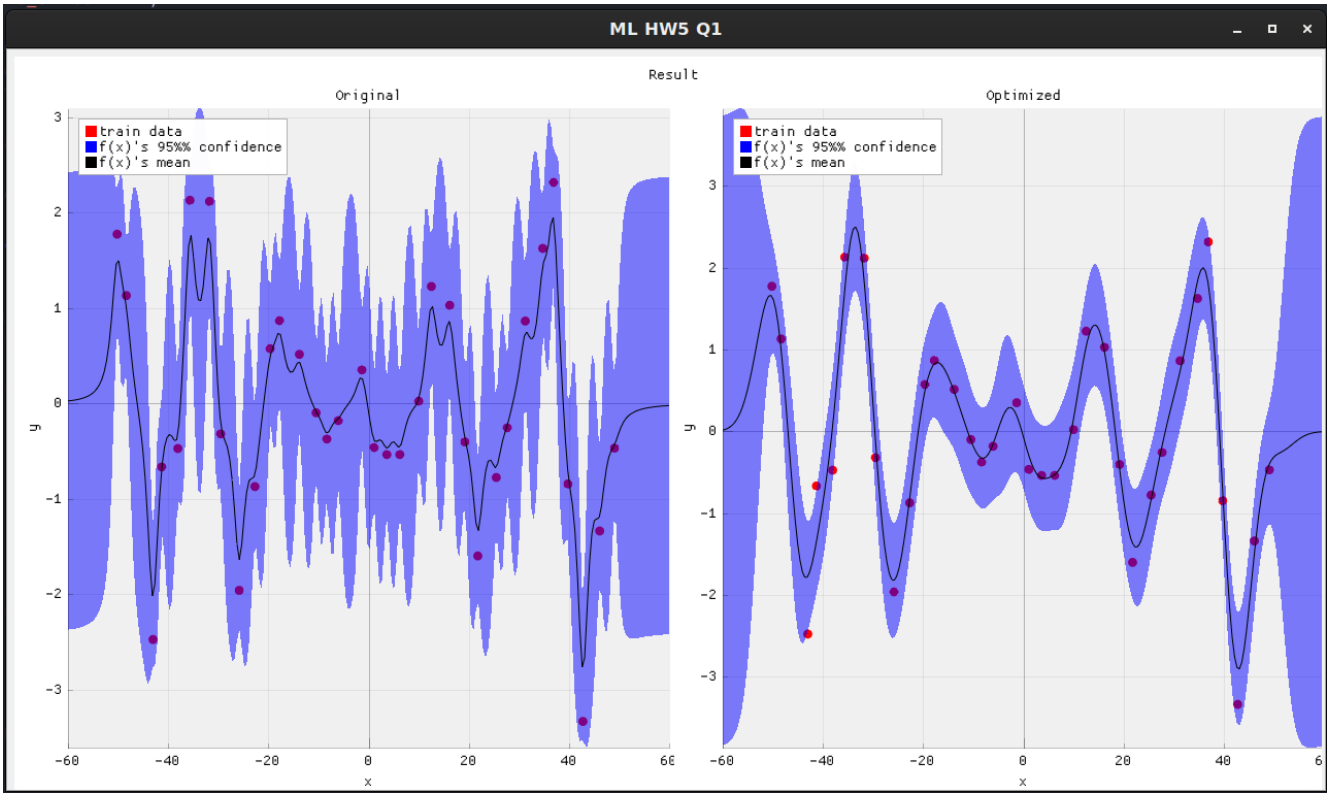
- β: 5

Result

- Optimized parameters

```
Optimized kernel parameters (variance, alpha, length scale)
1.72477
99999.6
3.31874
```

- Figure



Observations and discussion

- Before optimizing the kernel parameters of RBF function
  - the variance for each x is unstable.
  - some training data points are not fitted well, especially in the edge of curve.
- After optimizing the kernel parameters of RBF function
  - the problems above are relieved.
  - but the variances in intervals which do not have training data points are super large.
    - It is quite reasonable because we rely on kernel function to calculate the similarity between training data and x.
- If we adjust hyperparameter β, what it will happen.

SVM

Code with detailed explanations

Libraries

- LIBSVM
- Boost (for access file system only)
- OpenMP (for parallel acceleration)

Part1

- **Kernel functions**
  - Linear

$$k(\mathbf{x}, \mathbf{x}') = \mathbf{x}\mathbf{x}'^{\top} \tag{7}$$

  - RBF

$$k(\mathbf{x}, \mathbf{x}') = \exp^{-\gamma||\mathbf{x}-\mathbf{x}'||_2^2} \tag{8}$$

  - Polynomial

$$k(\mathbf{x}, \mathbf{x}') = (\gamma\mathbf{x}\mathbf{x}'^{\top} + coef0)^{degree} \tag{9}$$

- Entrypoint

```
switch (mode)
{
case 1:
{
    auto trainProblem = makeProblem(trainXData, trainYData);
    auto testProblem = makeProblem(testXData, testYData);
    solvePart1(trainProblem, testProblem, numberOfFeatures);
    releaseProblem(trainProblem);
    releaseProblem(testProblem);
    break;
}
case 2:
{
    auto trainProblem = makeProblem(trainXData, trainYData);
    auto testProblem = makeProblem(testXData, testYData);
    solvePart2(trainProblem, testProblem, numberOfFeatures);
    releaseProblem(trainProblem);
    releaseProblem(testProblem);
    break;
}
case 3:
    solvePart3(trainXData, trainYData, testXData, testYData, numberOfFeatures);
    break;
default:
    std::cerr << "Unknown mode." << std::endl;
    return 1;
}
```

- `makeProblem` function: convert data into `struct svm_problem` format

> Arguments
>
> x: data features
>
> y: data labels
>
> LIBSVM data structures

```
For example, if we have the following training data:

LABEL    ATTR1    ATTR2    ATTR3    ATTR4    ATTR5
-----    -----    -----    -----    -----    -----
 1         0        0.1      0.2      0        0
 2         0        0.1      0.3     -1.2      0
 1         0.4      0        0        0        0
 2         0        0.1      0        1.4      0.5
 3        -0.1     -0.2      0.1      1.1      0.1

then the components of svm_problem are:

l = 5

y -> 1 2 1 2 3

x -> [ ] -> (2,0.1) (3,0.2) (-1,?)
     [ ] -> (2,0.1) (3,0.3) (4,-1.2) (-1,?)
     [ ] -> (1,0.4) (-1,?)
     [ ] -> (2,0.1) (4,1.4) (5,0.5) (-1,?)
     [ ] -> (1,-0.1) (2,-0.2) (3,0.1) (4,1.1) (5,0.1) (-1,?)

where (index,value) is stored in the structure svm_node
*index = -1 indicates the end of one vector. Note that indices must be in
ASCENDING order.

struct svm_node
{
  int index;
  double value;
};

struct svm_problem
{
  int l;
  double *y;
  struct svm_node **x;
};
```

```cpp
svm_problem makeProblem(const std::vector<std::vector<double>> &x, std::vector<double> &y)
{
    auto featureSize = static_cast<int>(x[0].size());
    svm_problem problem;

    // number of data instances
    problem.l = static_cast<int>(x.size());
    // data labels
    problem.y = y.data();

    // data features: [N, pixels]
    problem.x = new svm_node *[problem.l];
    for (int i = 0; i < problem.l; i++)
    {
        problem.x[i] = new svm_node[featureSize];
    }

    // data conversion
#pragma omp parallel for
    for (int i = 0; i < problem.l; i++)
    {
        std::vector<svm_node> features;

        for (int j = 0; j < featureSize; j++)
        {
            double feature = x[i][j];
            if (feature == 0)
            {
                continue;
            }

            features.push_back(svm_node{j + 1, feature});
        }

        // insert end of features
        features.push_back(svm_node{-1, 0});

        std::move(features.begin(), features.end(), problem.x[i]);
    }

    return problem;
}
```

- `solvePart1` function: main function of Part 1

> Arguments
>
> trainProblem: training data represented by `struct svm_problem`
>
> testProblem: test data represented by `struct svm_problem`
>
> numberOfFeatures: number of pixels for each row (image)

```cpp
void solvePart1(const svm_problem &trainProblem, const svm_problem &testProblem, int numberOfFeatures)
{
    auto linearParameter = createSVMParameter(numberOfFeatures);

    auto polyParameter = createSVMParameter(numberOfFeatures);
    polyParameter.kernel_type = POLY;

    auto rbfParameter = createSVMParameter(numberOfFeatures);
    rbfParameter.kernel_type = RBF;

    // Part 1 Defaults
    std::cout << "Part 1" << std::endl;

    std::cout << "Linear Model" << std::endl;
    train_evaluate(trainProblem, testProblem, linearParameter);

    std::cout << "————————————————————————————————————————————————————————————" << std::endl;

    std::cout << "Polynomial Model" << std::endl;
    train_evaluate(trainProblem, testProblem, polyParameter);

    std::cout << "————————————————————————————————————————————————————————————" << std::endl;

    std::cout << "RBF Model" << std::endl;
    train_evaluate(trainProblem, testProblem, rbfParameter);

    std::cout << "============================================================" << std::endl;
}
```

- `createSVMParameter` function: create a default `struct svm_parameter` structure

> LIBSVM data structure
>
> ```cpp
> enum { C_SVC, NU_SVC, ONE_CLASS, EPSILON_SVR, NU_SVR }; /* svm_type */
> enum { LINEAR, POLY, RBF, SIGMOID, PRECOMPUTED }; /* kernel_type */
>
> struct svm_parameter
> {
>   int svm_type;
>   int kernel_type;
>   int degree; /* for poly */
>   double gamma; /* for poly/rbf/sigmoid */
>   double coef0; /* for poly/sigmoid */
>
>   /* these are for training only */
>   double cache_size; /* in MB */
>   double eps; /* stopping criteria */
>   double C; /* for C_SVC, EPSILON_SVR and NU_SVR */
>   int nr_weight;    /* for C_SVC */
>   int *weight_label;  /* for C_SVC */
>   double* weight;   /* for C_SVC */
>   double nu;  /* for NU_SVC, ONE_CLASS, and NU_SVR */
>   double p; /* for EPSILON_SVR */
>   int shrinking;  /* use the shrinking heuristics */
>   int probability; /* do probability estimates */
> };
> ```

> The initial value is copied from their svm program, named `svm-train.c`.
>
> We only focus on `kernel_type`, `C`, `gamma` (poly/RBF), `degree` (poly), `coef0` (poly).

```cpp
svm_parameter createSVMParameter(int numOfFeatures)
{
    svm_parameter parameter;
    parameter.svm_type = C_SVC;
    parameter.kernel_type = LINEAR;
    parameter.degree = 3;
    parameter.gamma = 1 / static_cast<double>(numOfFeatures); // 1/num_features
    parameter.coef0 = 0;
    parameter.nu = 0.5;
    parameter.cache_size = 100;
    parameter.C = 1;
    parameter.eps = 1e-3;
    parameter.p = 0.1;
    parameter.shrinking = 1;
    parameter.probability = 0;
    parameter.nr_weight = 0;
    parameter.weight_label = nullptr;
    parameter.weight = nullptr;
    return parameter;
}
```

- `train_evaluate` function: train the SVM by training data & evaluate the performance by test data

> Arguments
> trainProblem: training data represented by `struct svm_problem`
> testProblem: test data represented by `struct svm_problem`
> parameter: SVM parameters

```cpp
void train_evaluate(const svm_problem &trainProblem, const svm_problem &testProblem, const svm_parameter &parameter)
{
    auto model = train(trainProblem, parameter);
    auto predictions = predict(*model, testProblem);
    evaluate(testProblem, predictions);
    svm_free_and_destroy_model(&model);
}
```

- `train` function: train the SVM by training data & get the trained model

```cpp
svm_model *train(const svm_problem &problem, const svm_parameter &parameter)
{
    if (auto error = svm_check_parameter(&problem, &parameter); error != nullptr)
    {
        std::cerr << error << std::endl;
        throw std::runtime_error(error);
    }

    std::cout << "Start training..." << std::endl;

    return svm_train(&problem, &parameter);
}
```

- `predict` function: predict the test data by the trained model

```cpp
std::vector<double> predict(const svm_model &model, const svm_problem &problem)
{
    std::vector<double> predictions(problem.l);

    std::cout << "Start predicting..." << std::endl;
#pragma omp parallel for
    for (int i = 0; i < problem.l; i++)
    {
        predictions[i] = svm_predict(&model, problem.x[i]);
    }
    return predictions;
}
```

- `evaluate` function: evaluate the accuracy between predictions and ground truths.

```cpp
double evaluate(const svm_problem &problem, const std::vector<double> &predictions)
{
    int correctCount = 0;

    std::cout << "Start evaluating..." << std::endl;
#pragma omp parallel for reduction(+ : correctCount)
    for (int i = 0; i < problem.l; i++)
    {
        if (problem.y[i] == predictions[i])
        {
            correctCount++;
        }
    }

    double accuracy = static_cast<double>(correctCount) / problem.l;
    std::cout << "Accuracy: " << accuracy << std::endl;
    return accuracy;
}
```

Part2

- Entrypoint

```
switch (mode)
{
case 1:
{
    auto trainProblem = makeProblem(trainXData, trainYData);
    auto testProblem = makeProblem(testXData, testYData);
    solvePart1(trainProblem, testProblem, numberOfFeatures);
    releaseProblem(trainProblem);
    releaseProblem(testProblem);
    break;
}
case 2:
{
    auto trainProblem = makeProblem(trainXData, trainYData);
    auto testProblem = makeProblem(testXData, testYData);
    solvePart2(trainProblem, testProblem, numberOfFeatures);
    releaseProblem(trainProblem);
    releaseProblem(testProblem);
    break;
}
case 3:
    solvePart3(trainXData, trainYData, testXData, testYData, numberOfFeatures);
    break;
default:
    std::cerr << "Unknown mode." << std::endl;
    return 1;
}
```

- `solvePart2` function: main function of Part 2

> Arguments
>
> trainProblem: training data represented by `struct svm_problem`
>
> testProblem: test data represented by `struct svm_problem`
>
> numberOfFeatures: number of pixels for each row (image)
>
> Data structure for grid search

```
template <typename T>
struct GridSearchRange
{
 T start = -1;
 T end = -1;
 T step = 1;
};

struct GridSearchSettings
{
 /* for poly */
 GridSearchRange<int> degree;
 GridSearchRange<double> coef0;
```

```
    /* for poly/rbf */
    GridSearchRange<double> gamma;

    /* for C_SVC */
    GridSearchRange<double> C;

    int kFold = 3;
};
```

```
void solvePart2(const svm_problem &trainProblem, const svm_problem &testProblem, int numberOfFeatures)
{
    auto linearParameter = createSVMParameter(numberOfFeatures);

    auto polyParameter = createSVMParameter(numberOfFeatures);
    polyParameter.kernel_type = POLY;

    auto rbfParameter = createSVMParameter(numberOfFeatures);
    rbfParameter.kernel_type = RBF;

    // Part 2 Grid Search
    std::cout << "Part 2" << std::endl;

    GridSearchSettings settings;
    settings.degree.start = 1;
    settings.degree.end = 10;
    settings.coef0.start = -10;
    settings.coef0.end = 11;
    settings.gamma.start = -10;
    settings.gamma.end = 11;
    settings.C.start = -10;
    settings.C.end = 11;
    settings.kFold = 5;

    std::cout << "Linear Model" << std::endl;
    linearParameter = findCSVCParametersByGridSearch(trainProblem, linearParameter, settings);
    train_evaluate(trainProblem, testProblem, linearParameter);

    std::cout << "----------------------------------------------------------------------" << std::endl;

    std::cout << "Polynomial Model" << std::endl;
    polyParameter = findCSVCParametersByGridSearch(trainProblem, polyParameter, settings);
    train_evaluate(trainProblem, testProblem, polyParameter);

    std::cout << "----------------------------------------------------------------------" << std::endl;

    std::cout << "RBF Model" << std::endl;
    rbfParameter = findCSVCParametersByGridSearch(trainProblem, polyParameter, settings);
    train_evaluate(trainProblem, testProblem, rbfParameter);

    std::cout << "======================================================================" << std::endl;
}
```

- `findCSVCParametersByGridSearch` function: perform grid search & find the best parameters by cross-validation

> Arguments
>
> problem: training data represented by `struct svm_problem`
>
> defaultParameter: original SVM parameter
>
> settings: settings for grid search
>
> We only focus on `C`, `gamma` (poly/RBF), `degree` (poly), `coef0` (poly).

```cpp
svm_parameter findCSVCParametersByGridSearch(const svm_problem &problem, const svm_parameter &defaultParameter, const GridSearchSettings &settings)
{
    GridSearchSettings _settings = settings;

    if (defaultParameter.kernel_type != PRECOMPUTED && defaultParameter.kernel_type != POLY)
    {
        _settings.degree.start = defaultParameter.degree;
        _settings.degree.end = defaultParameter.degree + 1;
        _settings.degree.step = 1;

        _settings.coef0.start = defaultParameter.coef0;
        _settings.coef0.end = defaultParameter.coef0 + 1;
        _settings.coef0.step = 1;

        if (defaultParameter.kernel_type != RBF)
        {
            _settings.gamma.start = defaultParameter.gamma;
            _settings.gamma.end = defaultParameter.gamma + 1;
            _settings.gamma.step = 1;
        }
    }

    auto degreeList = generateSequence(_settings.degree);
    auto coef0List = generateSequence(_settings.coef0);
    auto gammaList = generateSequence(_settings.gamma);
    auto CList = generateSequence(_settings.C);

    double bestAccuracy = 0;
    svm_parameter bestParameter = defaultParameter;
#pragma omp parallel for collapse(4)
    for (auto degree : degreeList)
    {
        for (auto coef0 : coef0List)
        {
            for (auto gamma : gammaList)
            {
                for (auto C : CList)
                {
                    svm_parameter parameter = defaultParameter;
                    parameter.degree = degree;
                    parameter.coef0 = std::pow(2, coef0);
                    parameter.gamma = std::pow(2, gamma);
                    parameter.C = std::pow(2, C);

                    std::vector<double> targets(problem.l);
                    svm_cross_validation(&problem, &parameter, _settings.kFold, targets.data());

                    auto accuracy = evaluate(problem, targets);
#pragma omp critical
                    if (accuracy > bestAccuracy)
                    {
                        bestParameter = parameter;
                        bestAccuracy = accuracy;
                    }
                }
            }
        }
    }

    std::cout << "Best Cross Validation Accuracy: " << bestAccuracy << std::endl;
    std::cout << "Degree: " << bestParameter.degree << std::endl;
    std::cout << "Coef0: " << bestParameter.coef0 << std::endl;
    std::cout << "Gamma: " << bestParameter.gamma << std::endl;
    std::cout << "C: " << bestParameter.C << std::endl;

    return bestParameter;
}
```

Part3

- Custom kernel function (RBF + Linear kernel)

$$k(\mathbf{x}, \mathbf{x}') = \mathbf{x}\mathbf{x}'^{\top} + \exp^{-\gamma||\mathbf{x}-\mathbf{x}'||_2^2} \tag{10}$$

- Entrypoint

```
switch (mode)
{
case 1:
{
    auto trainProblem = makeProblem(trainXData, trainYData);
    auto testProblem = makeProblem(testXData, testYData);
    solvePart1(trainProblem, testProblem, numberOfFeatures);
    releaseProblem(trainProblem);
    releaseProblem(testProblem);
    break;
}
case 2:
{
    auto trainProblem = makeProblem(trainXData, trainYData);
    auto testProblem = makeProblem(testXData, testYData);
    solvePart2(trainProblem, testProblem, numberOfFeatures);
    releaseProblem(trainProblem);
    releaseProblem(testProblem);
    break;
}
case 3:
    solvePart3(trainXData, trainYData, testXData, testYData, numberOfFeatures);
    break;
default:
    std::cerr << "Unknown mode." << std::endl;
    return 1;
}
```

- `solvePart3` function: main function of Part 3

> Arguments
> trainX: x of training data, shape [N, pixels]
> trainY: y of training data, shape [N]
> testX: y of test data, shape [N, pixels]
> testY: y of test data, shape [N]
> numberOfFeatures: number of pixels for each row (image)

```cpp
void solvePart3(const std::vector<std::vector<double>> &trainX, std::vector<double> &trainY, const std::vector<std::vector<double>> &testX, std::vector<double> &testY, int numberOfFeatures)
{
    auto parameter = createSVMParameter(numberOfFeatures);
    parameter.kernel_type = PRECOMPUTED;

    // Part 3 RBF + Linear kernel
    std::cout << "Part 3" << std::endl;

    GridSearchSettings settings;
    settings.degree.start = parameter.degree;
    settings.degree.end = parameter.degree + 1;
    settings.degree.step = 1;

    settings.coef0.start = parameter.coef0;
    settings.coef0.end = parameter.coef0 + 1;
    settings.coef0.step = 1;

    settings.gamma.start = -10;
    settings.gamma.end = 11;
    settings.C.start = -10;
    settings.C.end = 11;
    settings.kFold = 5;

    parameter = findCSVCParametersByGridSearch(trainX, trainY, parameter, settings);

    auto trainProblem = makeKernel(trainX, trainX, trainY, parameter);
    auto testProblem = makeKernel(testX, trainX, testY, parameter);
    train_evaluate(trainProblem, testProblem, parameter);

    std::cout << "=============================================================" << std::endl;
    releaseProblem(trainProblem);
    releaseProblem(testProblem);
}
```

- `makeKernel` function: calculate precomputed kernel values with custom kernel function $k(\mathbf{x}, \mathbf{x}')$ & convert data into `struct svm_problem` format

> Arguments
>
> x1: x, shape [N, pixels]
>
> x2: x', shape [N]
>
> y: y, shape [N]
>
> parameter: SVM parameter

```cpp
svm_problem makeKernel(const std::vector<std::vector<double>> &x1, const std::vector<std::vector<double>> &x2, std::vector<double> &y, const svm_parameter &parameter)
{
    svm_problem problem;
    auto numberOfX1Data = static_cast<int>(x1.size());
    auto numberOfX2Data = static_cast<int>(x2.size());

    problem.l = numberOfX1Data;
    problem.y = y.data();

    // data features: [N, pixels]
    problem.x = new svm_node *[numberOfX1Data];
    for (int i = 0; i < numberOfX1Data; i++)
    {
        problem.x[i] = new svm_node[numberOfX2Data + 2];
    }

#pragma omp parallel for
    for (int i = 0; i < numberOfX1Data; i++)
    {
        std::vector<svm_node> features(numberOfX2Data + 2);

        auto xi = std::valarray<double>(x1[i].data(), x1[i].size());

        features[0] = svm_node{0, static_cast<double>(i + 1)};

#pragma omp simd
        for (int j = 0; j < numberOfX2Data; j++)
        {
            features[j + 1] = svm_node{j + 1, calculateKernel(xi, std::valarray<double>(x2[j].data(), x2[j].size()), parameter)};
        }

        features[numberOfX2Data + 1] = svm_node{-1, 0};

        std::move(features.begin(), features.end(), problem.x[i]);
    }

    return problem;
}
```

- `calculateKernel` function: calculate kernel value with custom kernel function (RBF + Linear kernel)

> Arguments
>
> x1: x, shape [N, pixels]
>
> x2: x', shape [N]
>
> parameter: SVM parameter
>
> Formula

$$k(\mathbf{x}, \mathbf{x}') = \mathbf{x}\mathbf{x}'^{\top} + \exp^{-\gamma||\mathbf{x}-\mathbf{x}'||_2^2} \tag{11}$$

```cpp
double calculateKernel(const std::valarray<double> &x1, const std::valarray<double> &x2, const svm_parameter &parameter)
{
    auto linear = (x1 * x2).sum();
    auto rbf = std::exp(-parameter.gamma * std::pow((x1 - x2), 2).sum());
    return linear + rbf;
}
```

- `findCSVCParametersByGridSearch` function: perform grid search & find the best parameters by cross-validation with custom kernel function

> Arguments
>
> x: x of training data, shape [N, pixels]
>
> y: y of training data, shape [N]
>
> defaultParameter: original SVM parameter
>
> settings: settings for grid search
>
> We only focus on `C`, `gamma` (RBF).
>
> Once we choose a set of SVM parameter, we have to recalculate the precomputed kernel.

```cpp
svm_parameter findCSVCParametersByGridSearch(const std::vector<std::vector<double>> &x, std::vector<double> &y, const svm_parameter &defaultParameter, const GridSearchSettings &settings)
{
    auto degreeList = generateSequence(settings.degree);
    auto coef0List = generateSequence(settings.coef0);
    auto gammaList = generateSequence(settings.gamma);
    auto CList = generateSequence(settings.C);

    double bestAccuracy = 0;
    svm_parameter bestParameter = defaultParameter;
#pragma omp parallel for collapse(4)
    for (auto degree : degreeList)
    {
        for (auto coef0 : coef0List)
        {
            for (auto gamma : gammaList)
            {
                for (auto C : CList)
                {
                    svm_parameter parameter = defaultParameter;
                    parameter.degree = degree;
                    parameter.coef0 = std::pow(2, coef0);
                    parameter.gamma = std::pow(2, gamma);
                    parameter.C = std::pow(2, C);

                    auto problem = makeKernel(x, x, y, parameter);

                    std::vector<double> targets(problem.l);
                    svm_cross_validation(&problem, &parameter, settings.kFold, targets.data());

                    auto accuracy = evaluate(problem, targets);
                    releaseProblem(problem);

#pragma omp critical
                    if (accuracy > bestAccuracy)
                    {
                        bestParameter = parameter;
                        bestAccuracy = accuracy;
                    }
                }
            }
        }
    }

```

Experiments settings and results

Default Settings

```
svm_parameter createSVMParameter(int numOfFeatures)
{
    svm_parameter parameter;
    parameter.svm_type = C_SVC;
    parameter.kernel_type = LINEAR;
    parameter.degree = 3;
    parameter.gamma = 1 / static_cast<double>(numOfFeatures); // 1/num_features
    parameter.coef0 = 0;
    parameter.nu = 0.5;
    parameter.cache_size = 100;
    parameter.C = 1;
    parameter.eps = 1e-3;
    parameter.p = 0.1;
    parameter.shrinking = 1;
    parameter.probability = 0;
    parameter.nr_weight = 0;
    parameter.weight_label = nullptr;
    parameter.weight = nullptr;
    return parameter;
}
```

Part1

> Settings: only change the kernel type to the corresponding kernel.

```
Part 1
Linear Model
Start training ...
Start predicting ...
Start evaluating ...
Accuracy: 0.9508
--------------------------------------------------------------------------------
Polynomial Model
Start training ...
Start predicting ...
Start evaluating ...
Accuracy: 0.3452
--------------------------------------------------------------------------------
RBF Model
Start training ...
Start predicting ...
Start evaluating ...
Accuracy: 0.9532
================================================================================
```

Part2

> Grid Search settings: `coef0`, `gamma`, `C` are exponential terms based on 2.

```
GridSearchSettings settings;
settings.degree.start = 1;
settings.degree.end = 11;
settings.degree.step = 2;
settings.coef0.start = -5;
settings.coef0.end = 6;
settings.coef0.step = 2;
settings.gamma.start = -5;
settings.gamma.end = 6;
settings.gamma.step = 2;
settings.C.start = -5;
settings.C.end = 6;
settings.C.step = 2;
settings.kFold = 5;
```

- Linear model

  > Accuracy of test data is 0.96.

  ```
  Best Cross Validation Accuracy: 0.9714
  Degree: 3
  Coef0: 1
  Gamma: 1.00088
  C: 0.03125
  Start training ...
  Start predicting ...
  Start evaluating ...
  Accuracy: 0.96
  ```

- Polynomial model

  > Accuracy of test data is 0.9804.

```
2186    Best Cross Validation Accuracy: 0.982
2187    Degree: 5
2188    Coef0: 32
2189    Gamma: 0.5
2190    C: 0.125
2191    Start training ...
2192    Start predicting ...
2193    Start evaluating ...
2194    Accuracy: 0.9804
```

- RBF model

> Accuracy of test data is 0.9792.

```
4357    Best Cross Validation Accuracy: 0.9824
4358    Degree: 3
4359    Coef0: 8
4360    Gamma: 0.5
4361    C: 2
4362    Start training ...
4363    Start predicting ...
4364    Start evaluating ...
4365    Accuracy: 0.9792
```

Part3

> RBF + Linear kernel
>
> Grid Search settings: `gamma`, `C` are exponential terms based on 2. (ignore `degree`, `coef0`)

```
GridSearchSettings settings;
settings.degree.start = parameter.degree;
settings.degree.end = parameter.degree + 1;
settings.degree.step = 1;

settings.coef0.start = parameter.coef0;
settings.coef0.end = parameter.coef0 + 1;
settings.coef0.step = 1;

settings.gamma.start = -10;
settings.gamma.end = 11;
settings.gamma.step = 2;
settings.C.start = -10;
settings.C.end = 11;
settings.C.step = 2;
settings.kFold = 5;
```

Accuracy of test data is 0.9604.

```
244    Best Cross Validation Accuracy: 0.9716
245    Degree: 3
246    Coef0: 1
247    Gamma: 0.00390625
248    C: 0.0625
249    Start training ...
250    Start predicting ...
251    Start evaluating ...
252    Accuracy: 0.9604
```

Observations and discussion

- Custom kernel function comparisons

> Grid Search settings: only change `gamma` and `C`
>
> Highest accuracy: 0.9844
>
> RBF * Linear kernel & RBF * Poly kernel
>
> The difference of these two kernels are `C` cost parameter.
>
> `gamma` parameter is same.

  - RBF + Linear kernel

```
Best Cross Validation Accuracy: 0.9716
Degree: 3
Coef0: 1
Gamma: 0.00390625
C: 0.0625
Start training ...
Start predicting ...
Start evaluating ...
Accuracy: 0.9604
```

  - RBF * Linear kernel

```
Best Cross Validation Accuracy: 0.9856
Degree: 3
Coef0: 1
Gamma: 0.015625
C: 256
Start training ...
Start predicting ...
Start evaluating ...
Accuracy: 0.9844
```

  - Linear + Poly kernel

```
244    Best Cross Validation Accuracy: 0.9816
245    Degree: 3
246    Coef0: 1
247    Gamma: 0.25
248    C: 0.000976562
249    Start training ...
250    Start predicting ...
251    Start evaluating ...
252    Accuracy: 0.9772
```

- Linear * Poly kernel

```
244    Best Cross Validation Accuracy: 0.9818
245    Degree: 3
246    Coef0: 1
247    Gamma: 0.015625
248    C: 0.25
249    Start training ...
250    Start predicting ...
251    Start evaluating ...
252    Accuracy: 0.98
```

- RBF + Poly kernel

```
Best Cross Validation Accuracy: 0.983
Degree: 3
Coef0: 1
Gamma: 0.015625
C: 1
Start training ...
Start predicting ...
Start evaluating ...
Accuracy: 0.98
```

- RBF * Poly kernel

```
244    Best Cross Validation Accuracy: 0.986
245    Degree: 3
246    Coef0: 1
247    Gamma: 0.015625
248    C: 4
249    Start training ...
250    Start predicting ...
251    Start evaluating ...
252    Accuracy: 0.9844
```