

Machine Learning Homework 5

Machine Learning Homework 5

Environment

Gaussian Process

Code with detailed explanations

Libraries

Visualization

Part1

Part2

Experiments settings and results

Settings

Result

Observations and discussion

SVM

Code with detailed explanations

Libraries

Part1

Part2

Part3

Experiments settings and results

Default Settings

Part1

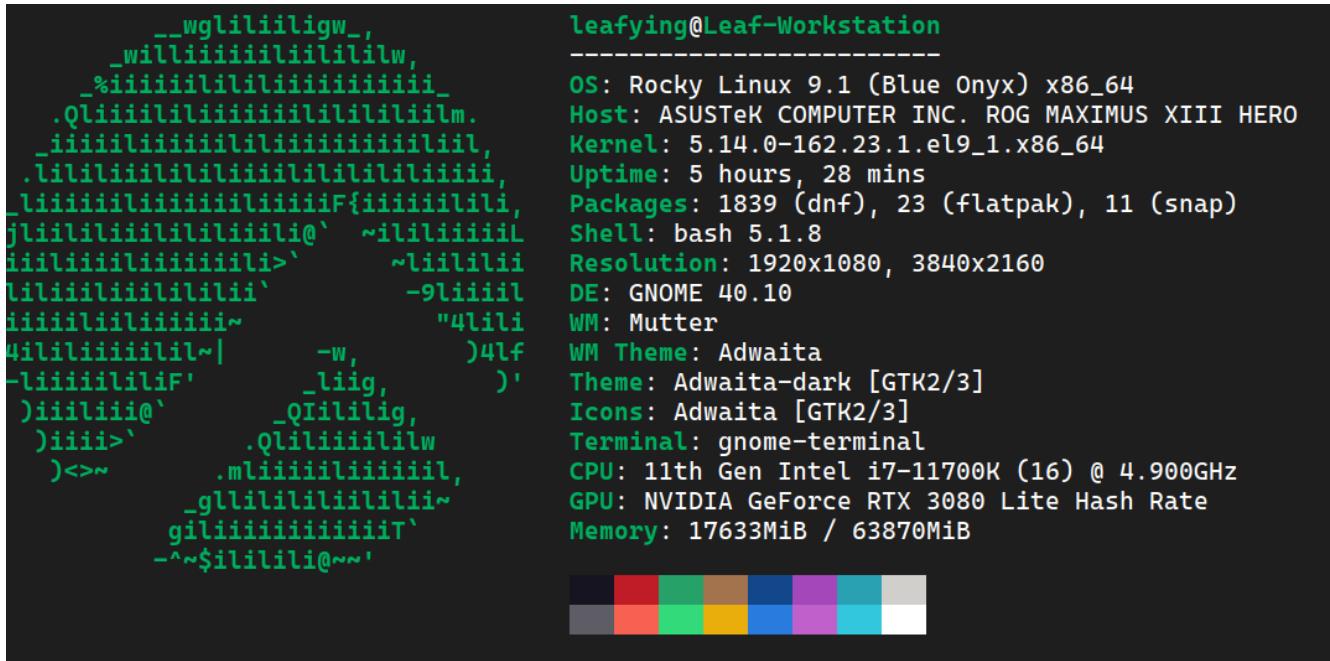
Part2

Part3

Observations and discussion

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.

```

60     static void glfwErrorCallback(int error, const char *description)
61     {
62         std::cerr << "GLFW Error " << error << ": " << description << std::endl;
63     }
64
65     GLFWwindow *setUpGUI()
66     {
67         glfwSetErrorCallback(glfwErrorCallback);
68         if (!glfwInit())
69         {
70             return nullptr;
71         }
72
73         // Decide GL+GLSL versions
74         // GL 3.3 + GLSL 330
75         const char *glsl_version = "#version 330";
76         glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
77         glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
78         glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE); // 3.2+ only
79         glfwWindowHint(GLFW_OPENGL_FORWARD_COMPAT, GL_TRUE); // 3.0+ only
80
81         // Create window with graphics context
82         GLFWwindow *window = glfwCreateWindow(1280, 720, TITLE.c_str(), nullptr, nullptr);
83         if (window == nullptr)
84             return nullptr;
85
86         glfwMakeContextCurrent(window);
87         glfwSwapInterval(0);
88
89         // Setup Dear ImGui context
90         IMGUI_CHECKVERSION();
91         ImGui::CreateContext();
92         ImPlot::CreateContext();
93
94         ImGuiIO &io = ImGui::GetIO();
95
96         ImGuiConfig cfg;
97         cfg.SizePixels = 15;
98         io.Fnts->AddFontDefault(&cfg);
99
100        // Setup Dear ImGui style
101        ImGui::StyleColorsLight();
102        // ImGui::StyleColorsDark();
103
104        // Setup Platform/Renderer backends
105        ImGui_ImplGlfw_InitForOpenGL(window, true);
106        ImGui_ImplOpenGL3_Init(glsl_version);
107
108        return window;
109    }

```

- **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(ImGuiAxis_X1, -60, 60);

```

Plot training data points

```

// plot points with red color and circle
ImPlot::SetNextMarkerStyle(ImGuiMarker_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 MatrixXd &data, const Matrix3d &f)
112 {
113     if (ImPlot::BeginPlot(title.c_str()))
114     {
115         ImPlot::SetupAxes("x", "y");
116         ImPlot::SetupAxisLimits(ImGuiAxis_X1, -60, 60);
117         // ImPlot::SetupAxisLimits(ImGuiAxis_Y1, -60, 60);
118
119         ImPlot::SetNextMarkerStyle(ImGuiMarker_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     }
132     ImPlot::EndPlot();
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

```
135 void showGUI(const MatrixX2d &data, const MatrixX3d &f, const MatrixX3d &optimizedF)
136 {
137     auto window = setUpGUI();
138     if (window == nullptr)
139     {
140         throw std::runtime_error("Cannot create window.");
141     }
142
143     // Our state
144     auto clear_color = ImVec4(0.45f, 0.55f, 0.60f, 1.00f);
145
146     // Main loop
147     while (!glfwWindowShouldClose(window))
148     {
149         glfwPollEvents();
150
151         // Start the Dear ImGui frame
152         ImGui_ImplOpenGL3_NewFrame();
153         ImGui_ImplGlfw_NewFrame();
154         ImGui::NewFrame();
155
156         {
157             const auto windowSize = ImGui::GetIO().DisplaySize;
158
159             ImGui::SetNextWindowPos(ImVec2(0, 0), ImGuiCond_Always);
160             ImGui::SetNextWindowSize(ImVec2(windowSize.x, windowSize.y), ImGuiCond_Always);
161             ImGui::Begin("Result", nullptr, ImGuiWindowFlags_NoDecoration);
162
163             if (ImGuiPlot::BeginSubplots("Result", 1, 2, ImVec2(windowSize.x, windowSize.y)))
164             {
165                 drawPlot("Original", data, f);
166                 drawPlot("Optimized", data, optimizedF);
167                 ImGuiPlot::EndSubplots();
168             }
169
170             ImGui::End();
171         }
172
173         // Rendering
174         ImGui::Render();
175         int display_w, display_h;
176         glfwGetFramebufferSize(window, &display_w, &display_h);
177         glViewport(0, 0, display_w, display_h);
178         glClearColor(clear_color.x * clear_color.w, clear_color.y * clear_color.w, clear_color.z * clear_color.w, clear_color.w);
179         glClear(GL_COLOR_BUFFER_BIT);
180         ImGui_ImplOpenGL3_RenderDrawData(ImGui::GetDrawData());
181
182         glfwSwapBuffers(window);
183     }
184
185     // Cleanup
186     ImGui_ImplOpenGL3_Shutdown();
187     ImGui_ImplGlfw_Shutdown();
188     ImGuiPlot::DestroyContext();
189     ImGui::DestroyContext();
190
191     glfwDestroyWindow(window);
192     glfwTerminate();
193 }
```

Part1

- Rational quadratic kernel

- Formula

$$k(x, x') = \sigma^2 \left(1 + \frac{\|x - x'\|_2^2}{2\alpha\ell^2}\right)^{-\alpha}$$

- 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

$$k(x, x') = \sigma^2 \left(1 + \frac{\text{diff}}{2\alpha\ell^2}\right)^{-\alpha}$$

```
220 template <typename DerivedA, typename DerivedB, typename Out = Eigen::Matrix<typename DerivedB::Scalar, DerivedA::RowsAtCompileTime, DerivedA::ColsAtCompileTime>>
221 Out calculateRationalQuadraticKernel(const Eigen::MatrixBase<DerivedA> &diff, const Eigen::MatrixBase<DerivedB> &kernelParameters)
222 {
223     return (kernelParameters[0] *
224             (1 + diff.array() / (2 * kernelParameters[1] * autodiff::detail::pow(kernelParameters[2], 2))).pow(-kernelParameters[1]));
225 }
```

- 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

$$\text{diff} = \|\vec{x} - \vec{x}'\|_2^2$$

```
226 template <typename DerivedA, typename DerivedB, typename Out = Eigen::Matrix<typename DerivedB::Scalar, Eigen::Dynamic, Eigen::Dynamic>>
227 Out calculateRationalQuadraticKernel(const Eigen::MatrixBase<DerivedA> &lhs, const Eigen::MatrixBase<DerivedA> &rhs, const Eigen::MatrixBase<DerivedB> &kernelParameters)
228 {
229     Eigen::Matrix<typename DerivedA::Scalar, Eigen::Dynamic, Eigen::Dynamic> diff = (lhs.replicate(1, lhs.rows()).rowwise() - rhs.transpose().array().pow(2));
230     return calculateRationalQuadraticKernel(diff, kernelParameters);
231 }
232 }
```

- 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

$$\text{diff} = \|\vec{x} - \vec{x}'\|_2^2$$

```
240 VectorXd calculateRationalQuadraticKernel(const VectorXd &lhs, double rhs, const Vector3d &kernelParameters)
241 {
242     VectorXd diff = (lhs.array() - rhs).pow(2);
243     return calculateRationalQuadraticKernel(diff, kernelParameters);
244 }
```

- 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)

$$\text{diff} = \|x - x'\|_2^2$$

```
234 double calculateRationalQuadraticKernel(double lhs, double rhs, const Vector3d &kernelParameters)
235 {
236     VectorXd diff = VectorXd::Constant(1, std::pow(lhs - rhs, 2));
237     return calculateRationalQuadraticKernel(diff, kernelParameters).value();
238 }
```

• Gaussian Process Regression

- Formula

- Training with RBF kernel

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

$$\begin{aligned} \mathbf{C}_\theta(\mathbf{x}_n, \mathbf{x}_m) &= k_\theta(\mathbf{x}_n, \mathbf{x}_m) + \beta^{-1}\delta_{nm} \\ \mathbf{C}_\theta(\mathbf{x}, \mathbf{x}') &= K_\theta(\mathbf{x}, \mathbf{x}') + \beta^{-1}\mathbf{I}_n \end{aligned} \quad (1)$$

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

- Prediction

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

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

- 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     // external 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}_θ 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

$$\mathbf{C}_\theta(\mathbf{x}, \mathbf{x}') = K_\theta(\mathbf{x}, \mathbf{x}') + \beta^{-1} \mathbf{I}_n \quad (3)$$

```

246 template<typename Derived, typename Out = Eigen::Matrix<typename Derived::Scalar, Eigen::Dynamic, Eigen::Dynamic>>
247 Out calculateCovariance(const MatrixX2d &data, double beta, const Eigen::MatrixBase<Derived> &kernelParameters)
248 {
249     Out covariance = calculateRationalQuadraticKernel(data.col(0), data.col(0), kernelParameters);
250     covariance.diagonal().array() += (1 / beta);
251     return covariance;
252 }
```

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

Formula

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

$$\mu(\mathbf{x}^*) = k(\mathbf{x}, \mathbf{x}^*)^\top \mathbf{C}^{-1} \mathbf{y} \quad (4)$$

$$\sigma^2(\mathbf{x}^*) = k(\mathbf{x}^*, \mathbf{x}^*) + \beta^{-1} - k(\mathbf{x}, \mathbf{x}^*)^\top \mathbf{C}^{-1} k(\mathbf{x}, \mathbf{x}^*)$$

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

```

296 MatrixX3d generatePointsOfLine(const MatrixX2d &data, double beta, const MatrixXd &covariance, const Vector3d& kernelParameters)
297 {
298     MatrixXd inv_covariance = covariance.inverse();
299     double inv_beta = 1 / beta;
300     MatrixX3d f(250, 3);
301     f.col(0) = VectorXd::LinSpaced(250, -60, 60);
302
303     for (auto row : f.rowwise())
304     {
305         double k = calculateRationalQuadraticKernel(row[0], row[0], kernelParameters) + inv_beta;
306         VectorXd kernel = calculateRationalQuadraticKernel(data.col(0), row[0], kernelParameters);
307
308         Eigen::Matrix<double, 1, Eigen::Dynamic> common = kernel.transpose() * inv_covariance;
309         row[1] = (common * data.col(1)).value();
310         row[2] = k - (common * kernel).value();
311     }
312
313     return f;
314 }
```

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

```

314 void modelData(const MatrixXd &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     // external 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 }

```

- `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

```

267     double evaluateOptimFn(const VectorXd &parameters, VectorXd *grad_out, void *args)
268     {
269         auto lossArguments = reinterpret_cast<LossArguments*>(args);
270         autodiff::real u;
271         autodiff::Vector3real paramtersd = parameters.eval();
272
273         if (grad_out != nullptr)
274         {
275             // in order to use external data in autodiff library
276             // create a lambda function to wrap the loss function
277             auto lossFn = [lossArguments](const autodiff::Vector3real &paramtersd)
278             {
279                 return calculateLoss(paramtersd, lossArguments->data, lossArguments->beta);
280             };
281
282             *grad_out = autodiff::gradient(lossFn, autodiff::wrt(paramtersd), autodiff::at(paramtersd), u);
283         }
284         else
285         {
286             u = calculateLoss(paramtersd, lossArguments->data, lossArguments->beta);
287         }
288
289         return u.val();
290     }

```

- `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

$$\ln p(\mathbf{y}|\theta) = \frac{1}{2} \ln \det \mathbf{C}_\theta + \frac{1}{2} \mathbf{y}^\top \mathbf{C}_\theta^{-1} \mathbf{y} + \frac{N}{2} \ln(2\pi) \quad (5)$$

```

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

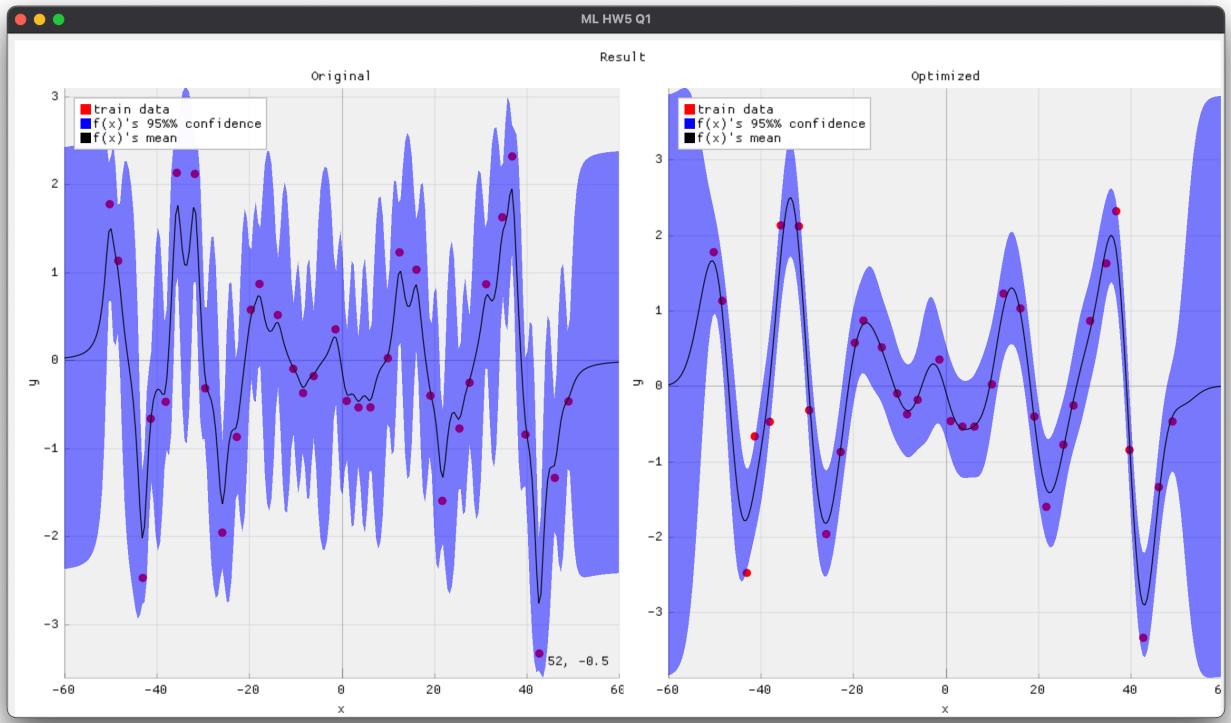
- Optimized parameters

```

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.

The inverse of β is the variance of error noise.

In theory, the bigger the β is, the smaller the variance is.

As you can see, if we increase the β value, the regression line will coverage to different local minimum.

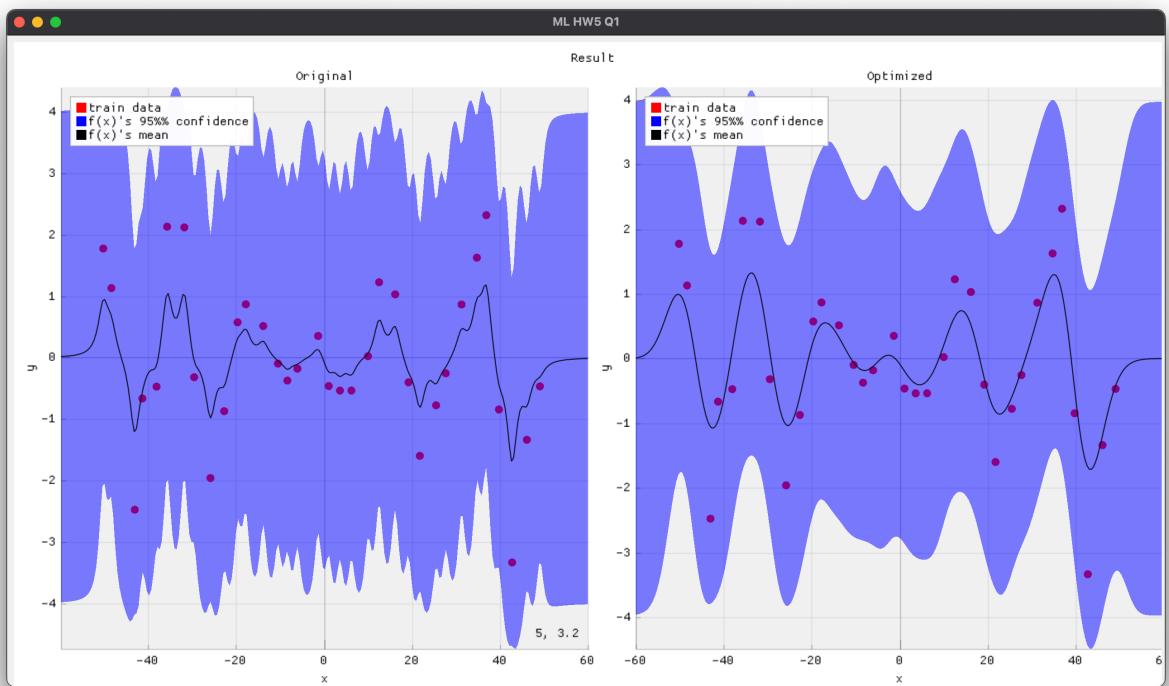
In particular, from $\beta = 1$ to 4, from $\beta = 5$ to 7, from $\beta = 10$ to 100.

In the first part, it is still keep coveraging the training data.

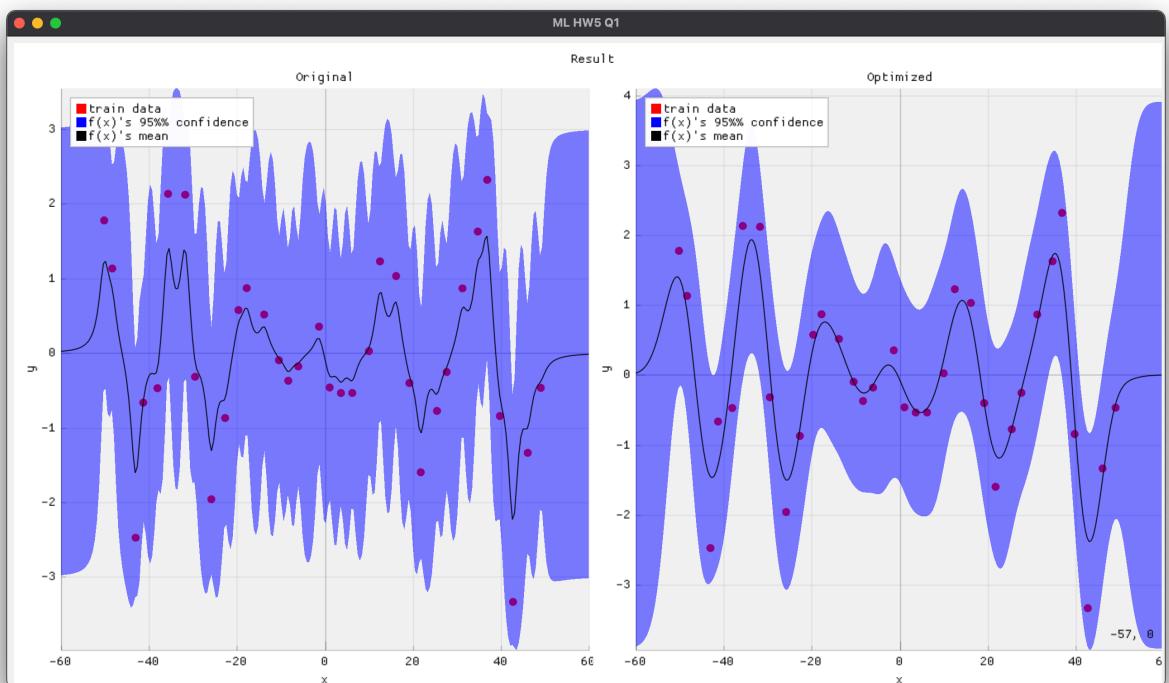
In the second part, almost all points are covered.

In the last part, the regression line will be overfitting to the training data.

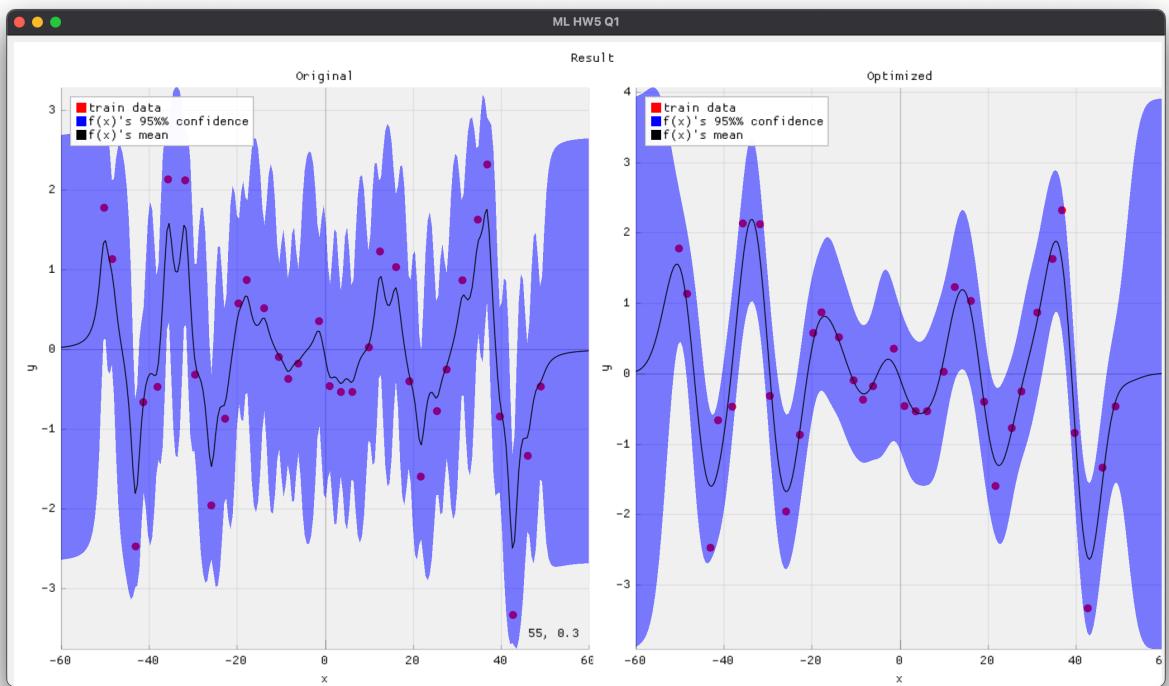
- $\beta = 1$



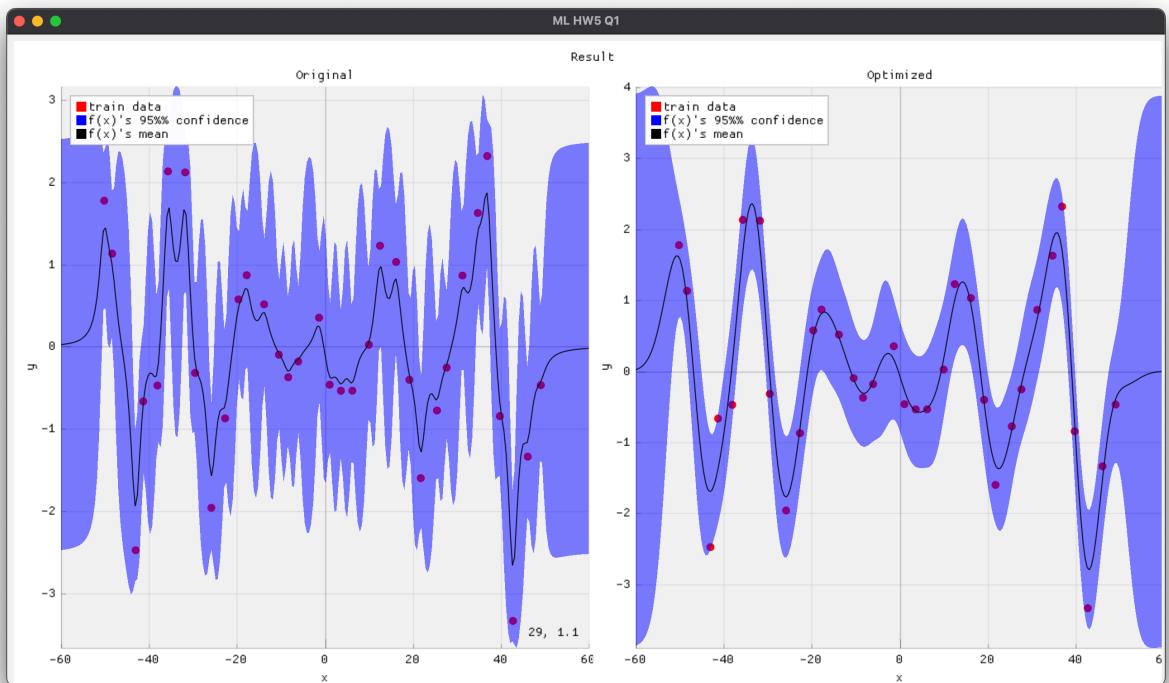
- o $\beta = 2$



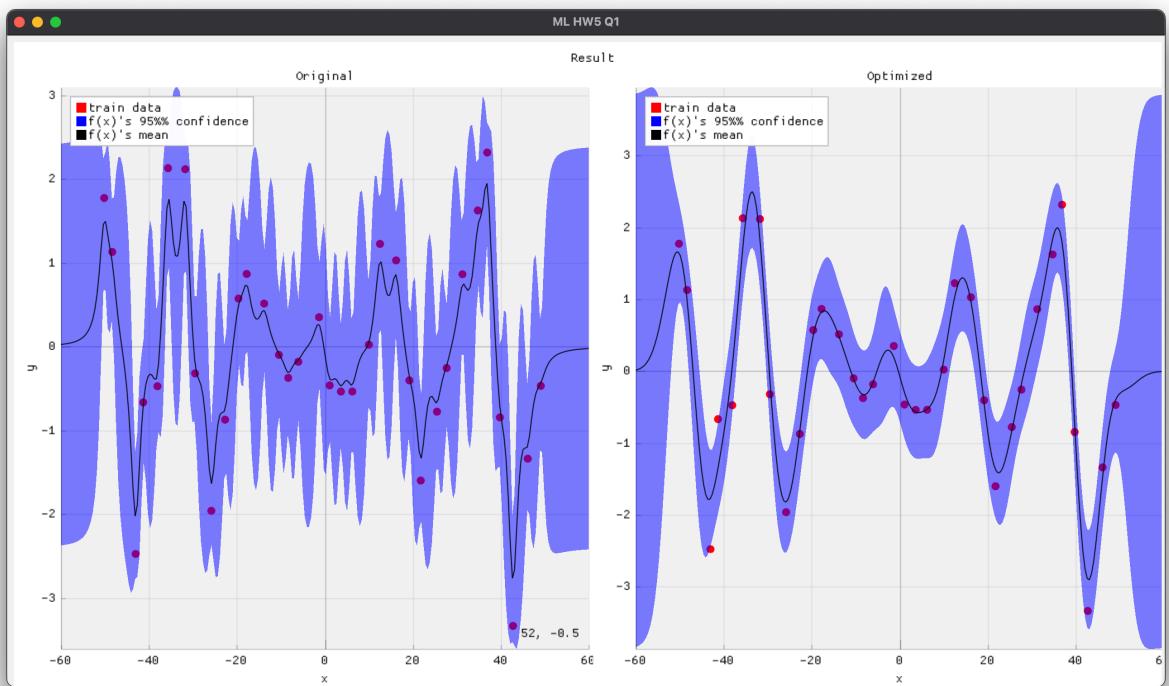
- o $\beta = 3$



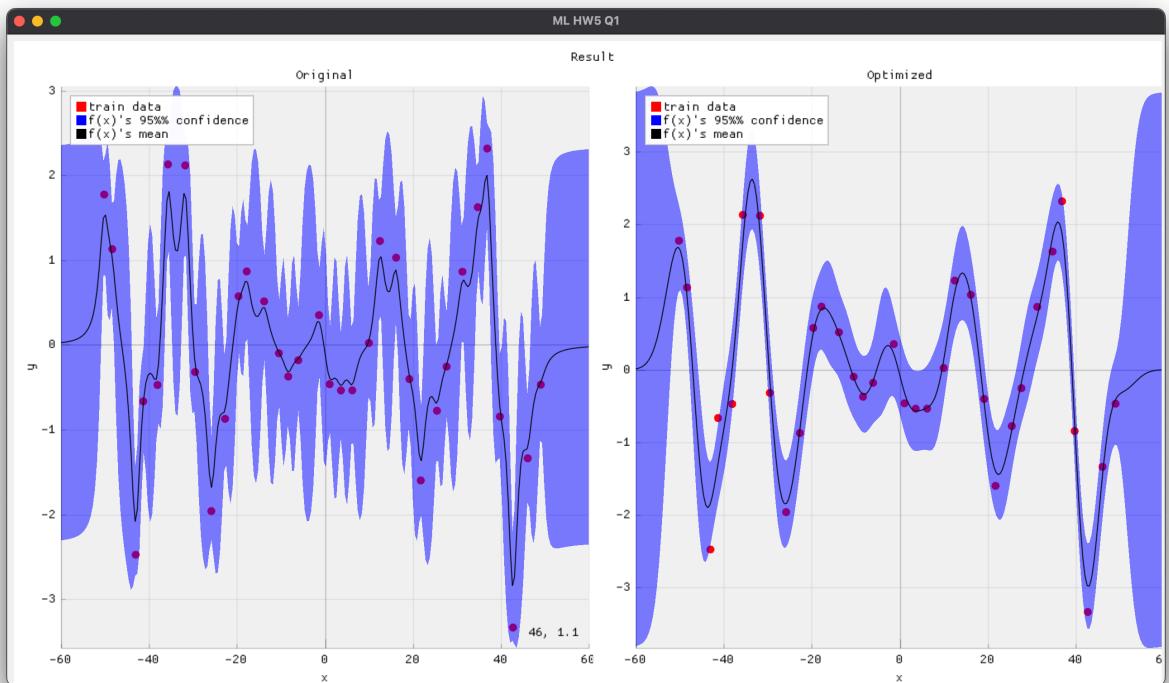
- $\beta = 4$



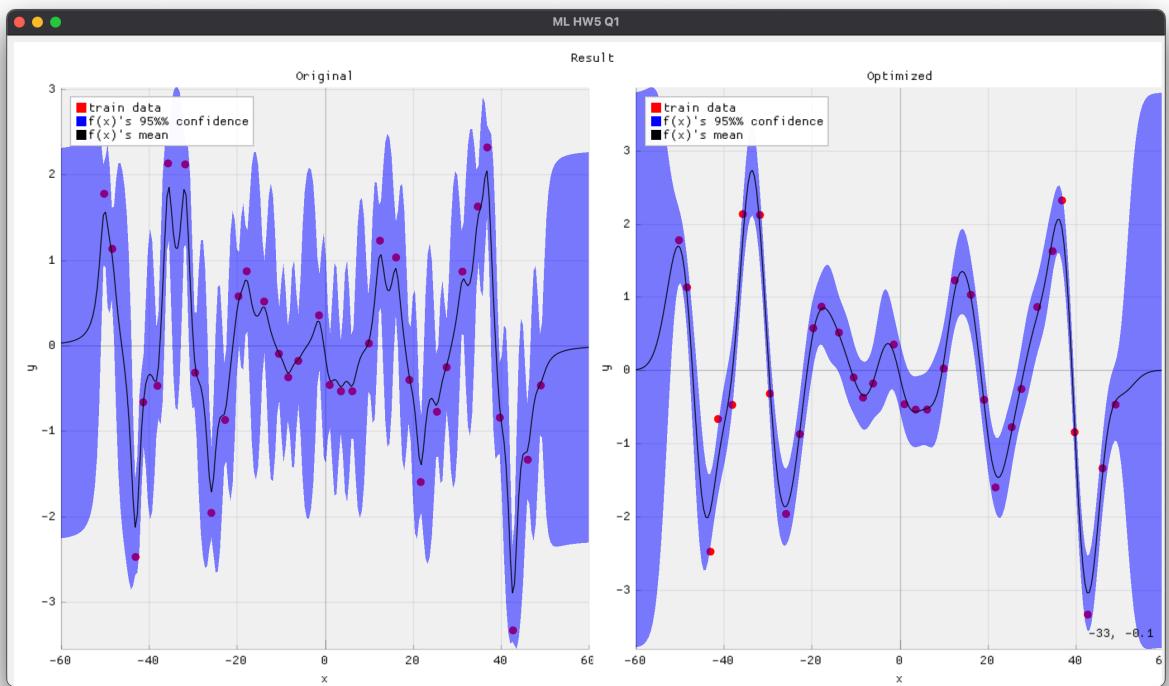
- $\beta = 5$



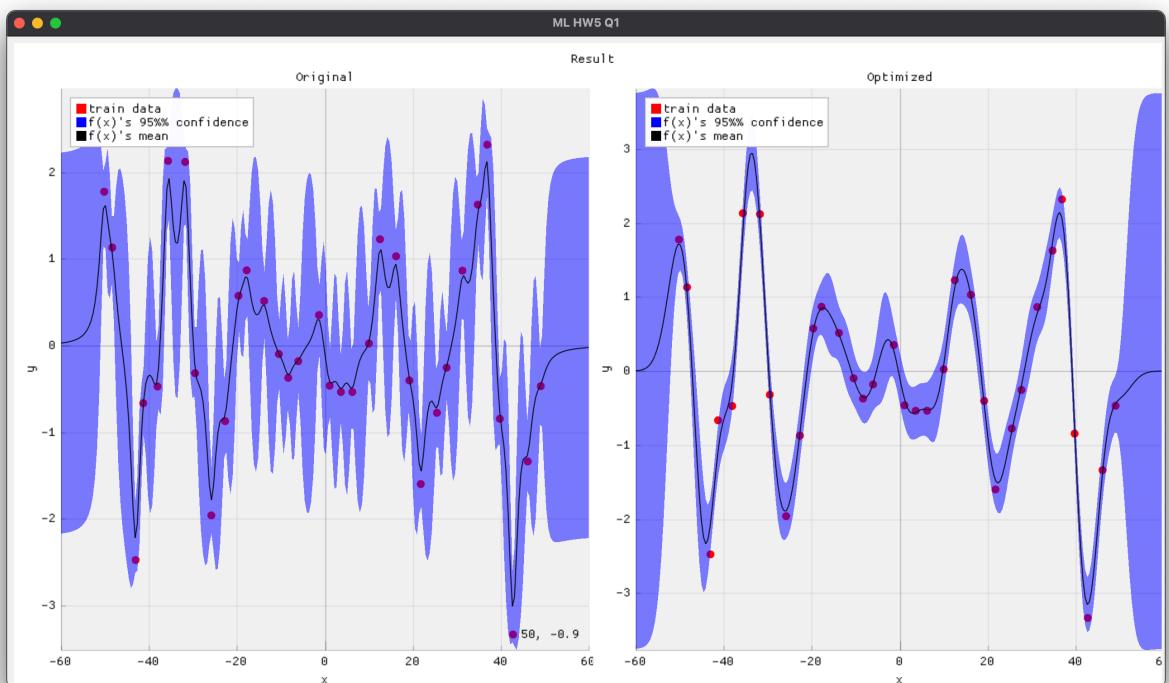
- $\beta = 6$



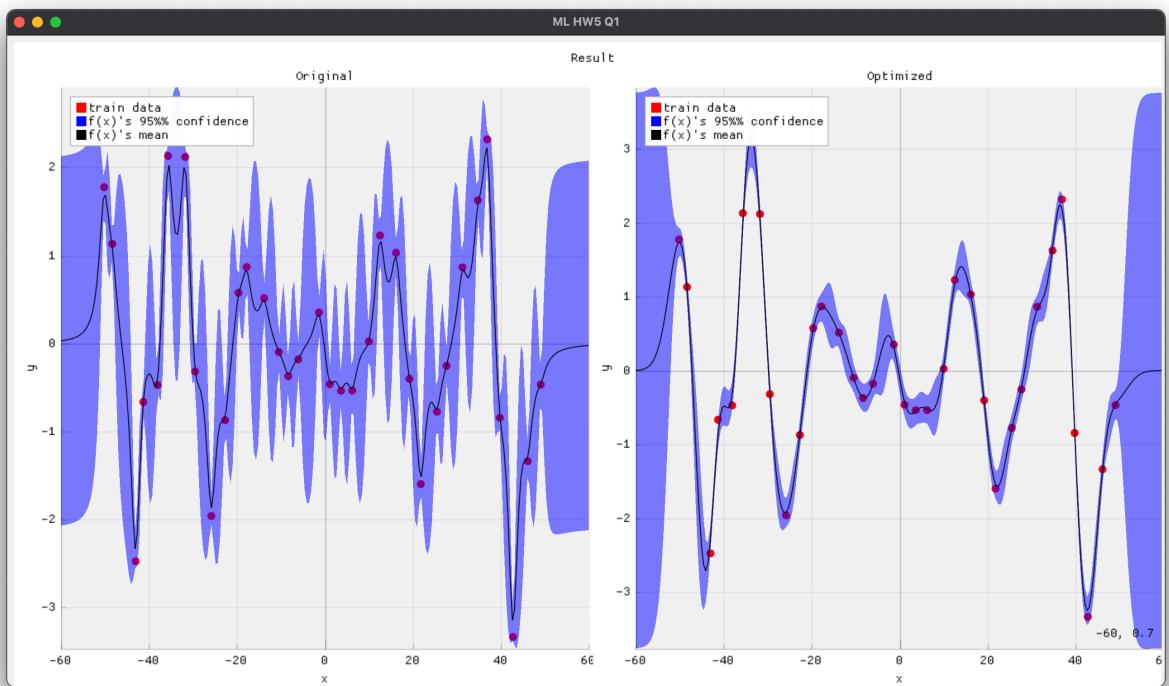
- $\beta = 7$



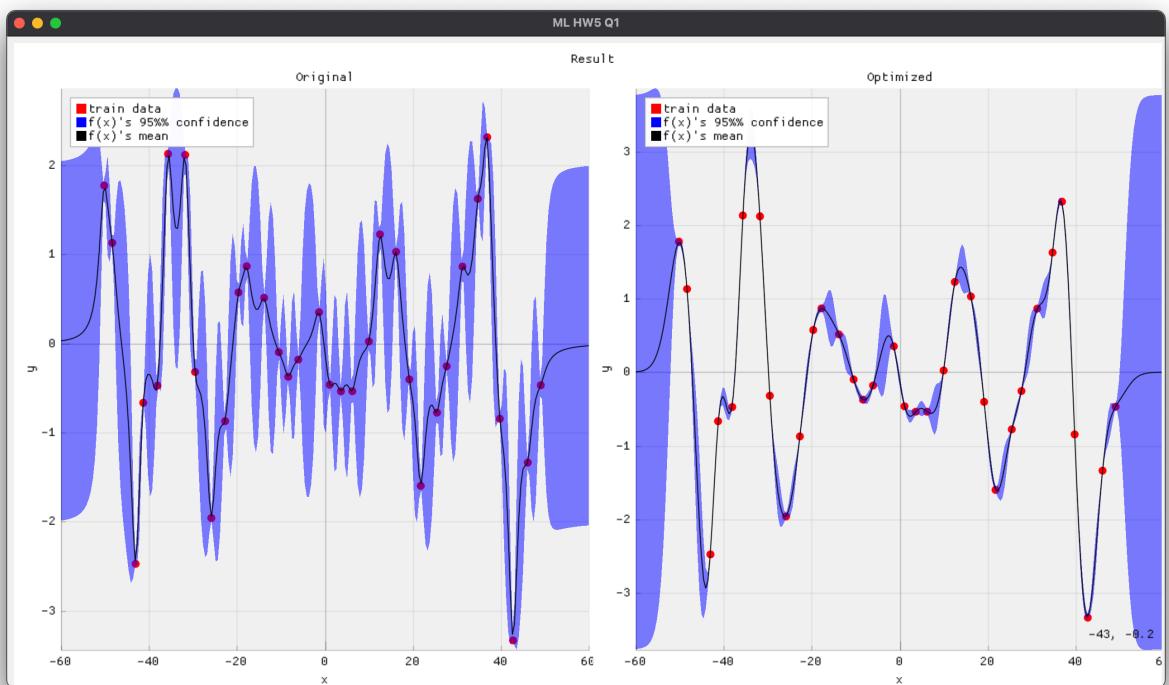
- $\beta = 10$



- $\beta = 20$



- $\beta = 100$



SVM

Code with detailed explanations

Libraries

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

Part1

- Kernel functions

- Linear

$$k(\mathbf{x}, \mathbf{x}') = \mathbf{x}\mathbf{x}'^\top \quad (6)$$

- RBF

$$k(\mathbf{x}, \mathbf{x}') = \exp^{-\gamma \|\mathbf{x} - \mathbf{x}'\|_2^2} \quad (7)$$

- Polynomial

$$k(\mathbf{x}, \mathbf{x}') = (\gamma \mathbf{x}\mathbf{x}'^\top + coef0)^{degree} \quad (8)$$

- 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;
};
```

```

95  svm_problem makeProblem(const std::vector<std::vector<double>> &x, std::vector<double> &y)
96  {
97      auto featureSize = static_cast<int>(x[0].size());
98      svm_problem problem;
99
100     // number of data instances
101     problem.l = static_cast<int>(x.size());
102     // data labels
103     problem.y = y.data();
104
105     // data features: [N, pixels]
106     problem.x = new svm_node *[problem.l];
107     for (int i = 0; i < problem.l; i++)
108     {
109         problem.x[i] = new svm_node[featureSize];
110     }
111
112     // data conversion
113 #pragma omp parallel for
114     for (int i = 0; i < problem.l; i++)
115     {
116         std::vector<svm_node> features;
117
118         for (int j = 0; j < featureSize; j++)
119         {
120             double feature = x[i][j];
121             if (feature == 0)
122             {
123                 continue;
124             }
125
126             features.push_back(svm_node{j + 1, feature});
127         }
128
129         // insert end of features
130         features.push_back(svm_node{-1, 0});
131
132         std::move(features.begin(), features.end(), problem.x[i]);
133     }
134
135     return problem;
136 }

```

- `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)

```

404     ~ void solvePart1(const svm_problem &trainProblem, const svm_problem &testProblem, int number_of_features)
405     {
406         auto linearParameter = createSVMParameter(number_of_features);
407
408         auto polyParameter = createSVMParameter(number_of_features);
409         polyParameter.kernel_type = POLY;
410
411         auto rbfParameter = createSVMParameter(number_of_features);
412         rbfParameter.kernel_type = RBF;
413
414         // Part 1 Defaults
415         std::cout << "Part 1" << std::endl;
416
417         std::cout << "Linear Model" << std::endl;
418         train_evaluate(trainProblem, testProblem, linearParameter);
419
420         std::cout << "-----" << std::endl;
421
422         std::cout << "Polynomial Model" << std::endl;
423         train_evaluate(trainProblem, testProblem, polyParameter);
424
425         std::cout << "-----" << std::endl;
426
427         std::cout << "RBF Model" << std::endl;
428         train_evaluate(trainProblem, testProblem, rbfParameter);
429
430         std::cout << "===== " << std::endl;
431     }

```

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

LIBSVM data structure

```

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).

```
196  ↘ svm_parameter createSVMParameter(int numOfFeatures)
197  {
198      svm_parameter parameter;
199      parameter.svm_type = C_SVC;
200      parameter.kernel_type = LINEAR;
201      parameter.degree = 3;
202      parameter.gamma = 1 / static_cast<double>(numOfFeatures); // 1/num_features
203      parameter.coef0 = 0;
204      parameter.nu = 0.5;
205      parameter.cache_size = 100;
206      parameter.C = 1;
207      parameter.eps = 1e-3;
208      parameter.p = 0.1;
209      parameter.shrinking = 1;
210      parameter.probability = 0;
211      parameter.nr_weight = 0;
212      parameter.weight_label = nullptr;
213      parameter.weight = nullptr;
214      return parameter;
215 }
```

- `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

```
262 void train_evaluate(const svm_problem &trainProblem, const svm_problem &testProblem, const svm_parameter &parameter)
263 {
264     auto model = train(trainProblem, parameter);
265     auto predictions = predict(*model, testProblem);
266     evaluate(testProblem, predictions);
267     svm_free_and_destroy_model(&model);
268 }
```

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

```
217     svm_model *train(const svm_problem &problem, const svm_parameter &parameter)
218     {
219         if (auto error = svm_check_parameter(&problem, &parameter); error != nullptr)
220         {
221             std::cerr << error << std::endl;
222             throw std::runtime_error(error);
223         }
224
225         std::cout << "Start training..." << std::endl;
226
227         return svm_train(&problem, &parameter);
228     }
```

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

```

230 √ std::vector<double> predict(const svm_model &model, const svm_problem &problem)
231 {
232     std::vector<double> predictions(problem.l);
233
234     std::cout << "Start predicting ... " << std::endl;
235 #pragma omp parallel for
236 √   for (int i = 0; i < problem.l; i++)
237     {
238         predictions[i] = svm_predict(&model, problem.x[i]);
239     }
240     return predictions;
241 }
```

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

```

243 √ double evaluate(const svm_problem &problem, const std::vector<double> &predictions)
244 {
245     int correctCount = 0;
246
247     std::cout << "Start evaluating ... " << std::endl;
248 #pragma omp parallel for reduction(+: correctCount)
249 √   for (int i = 0; i < problem.l; i++)
250     {
251         if (problem.y[i] == predictions[i])
252         {
253             correctCount++;
254         }
255     }
256
257     double accuracy = static_cast<double>(correctCount) / problem.l;
258     std::cout << "Accuracy: " << accuracy << std::endl;
259     return accuracy;
260 }
```

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;
};

```

```

433 void solvePart2(const svm_problem &trainProblem, const svm_problem &testProblem, int numberOfFeatures)
434 {
435     auto linearParameter = createSVMParameter(numberOfFeatures);
436
437     auto polyParameter = createSVMParameter(numberOfFeatures);
438     polyParameter.kernel_type = POLY;
439
440     auto rbfParameter = createSVMParameter(numberOfFeatures);
441     rbfParameter.kernel_type = RBF;
442
443     // Part 2 Grid Search
444     std::cout << "Part 2" << std::endl;
445
446     GridSearchSettings settings;
447     settings.degree.start = 1;
448     settings.degree.end = 10;
449     settings.coef0.start = -10;
450     settings.coef0.end = 11;
451     settings.gamma.start = -10;
452     settings.gamma.end = 11;
453     settings.C.start = -10;
454     settings.C.end = 11;
455     settings.kFold = 5;
456
457     std::cout << "Linear Model" << std::endl;
458     linearParameter = findCSVCParametersByGridSearch(trainProblem, linearParameter, settings);
459     train_evaluate(trainProblem, testProblem, linearParameter);
460
461     std::cout << "-----" << std::endl;
462
463     std::cout << "Polynomial Model" << std::endl;
464     polyParameter = findCSVCParametersByGridSearch(trainProblem, polyParameter, settings);
465     train_evaluate(trainProblem, testProblem, polyParameter);
466
467     std::cout << "-----" << std::endl;
468
469     std::cout << "RBF Model" << std::endl;
470     rbfParameter = findCSVCParametersByGridSearch(trainProblem, rbfParameter, settings);
471     train_evaluate(trainProblem, testProblem, rbfParameter);
472
473     std::cout << "-----" << std::endl;
474 }

```

- `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).

```

284     svm_parameter findCSVCPParametersByGridSearch(const svm_problem &problem, const svm_parameter &defaultParameter, const GridSearchSettings &settings)
285     {
286         GridSearchSettings _settings = settings;
287
288         if (defaultParameter.kernel_type != PRECOMPUTED && defaultParameter.kernel_type != POLY)
289         {
290             _settings.degree.start = defaultParameter.degree;
291             _settings.degree.end = defaultParameter.degree + 1;
292             _settings.degree.step = 1;
293
294             _settings.coef0.start = defaultParameter.coef0;
295             _settings.coef0.end = defaultParameter.coef0 + 1;
296             _settings.coef0.step = 1;
297
298             if (defaultParameter.kernel_type != RBF)
299             {
300                 _settings.gamma.start = defaultParameter.gamma;
301                 _settings.gamma.end = defaultParameter.gamma + 1;
302                 _settings.gamma.step = 1;
303             }
304         }
305
306         auto degreeList = generateSequence(_settings.degree);
307         auto coef0List = generateSequence(_settings.coef0);
308         auto gammaList = generateSequence(_settings.gamma);
309         auto CList = generateSequence(_settings.C);
310
311         double bestAccuracy = 0;
312         svm_parameter bestParameter = defaultParameter;
313 #pragma omp parallel for collapse(4)
314         for (auto degree : degreeList)
315         {
316             for (auto coef0 : coef0List)
317             {
318                 for (auto gamma : gammaList)
319                 {
320                     for (auto C : CList)
321                     {
322                         svm_parameter parameter = defaultParameter;
323                         parameter.degree = degree;
324                         parameter.coef0 = std::pow(2, coef0);
325                         parameter.gamma = std::pow(2, gamma);
326                         parameter.C = std::pow(2, C);
327
328                         std::vector<double> targets(problem.l);
329                         svm_cross_validation(&problem, &parameter, _settings.kFold, targets.data());
330
331                         auto accuracy = evaluate(problem, targets);
332 #pragma omp critical
333                         if (accuracy > bestAccuracy)
334                         {
335                             bestParameter = parameter;
336                             bestAccuracy = accuracy;
337                         }
338                     }
339                 }
340             }
341         }
342
343         std::cout << "Best Cross Validation Accuracy: " << bestAccuracy << std::endl;
344         std::cout << "Degree: " << bestParameter.degree << std::endl;
345         std::cout << "Coef0: " << bestParameter.coef0 << std::endl;
346         std::cout << "Gamma: " << bestParameter.gamma << std::endl;
347         std::cout << "C: " << bestParameter.C << std::endl;
348
349         return bestParameter;
350     }

```

Part3

- Custom kernel function (RBF + Linear kernel)

$$k(\mathbf{x}, \mathbf{x}') = \mathbf{x}\mathbf{x}'^\top + \exp^{-\gamma\|\mathbf{x}-\mathbf{x}'\|_2^2} \quad (9)$$

- 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;
}
```

- `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)

```

476 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)
477 {
478     auto parameter = createSVMParameter(numberOffeatures);
479     parameter.kernel_type = PRECOMPUTED;
480
481     // Part 3 RBF + Linear kernel
482     std::cout << "Part 3" << std::endl;
483
484     GridSearchSettings settings;
485     settings.degree.start = parameter.degree;
486     settings.degree.end = parameter.degree + 1;
487     settings.degree.step = 1;
488
489     settings.coef0.start = parameter.coef0;
490     settings.coef0.end = parameter.coef0 + 1;
491     settings.coef0.step = 1;
492
493     settings.gamma.start = -10;
494     settings.gamma.end = 11;
495     settings.C.start = -10;
496     settings.C.end = 11;
497     settings.kFold = 5;
498
499     parameter = findCSVCPParametersByGridSearch(trainX, trainY, parameter, settings);
500
501     auto trainProblem = makeKernel(trainX, trainX, trainY, parameter);
502     auto testProblem = makeKernel(testX, trainX, testY, parameter);
503     train_evaluate(trainProblem, testProblem, parameter);
504
505     std::cout << "-----" << std::endl;
506     releaseProblem(trainProblem);
507     releaseProblem(testProblem);
508 }

```

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

Arguments

`x1`: \mathbf{x} , shape [N, pixels]
`x2`: \mathbf{x}' , shape [N]
`y`: \mathbf{y} , shape [N]
`parameter`: SVM parameter

```

145 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)
146 {
147     svm_problem problem;
148     auto numberOff1Data = static_cast<int>(x1.size());
149     auto numberOff2Data = static_cast<int>(x2.size());
150
151     problem.l = numberOff1Data;
152     problem.y = y.data();
153
154     // data features: [N, pixels]
155     problem.x = new svm_node*[numberOff1Data];
156     for (int i = 0; i < numberOff1Data; i++)
157     {
158         problem.x[i] = new svm_node[numberOff2Data + 2];
159     }
160
161 #pragma omp parallel for
162     for (int i = 0; i < numberOff1Data; i++)
163     {
164         std::vector<svm_node> features(numberOff2Data + 2);
165
166         auto xi = std::valarray<double>(x1[i].data(), x1[i].size());
167
168         features[0] = svm_node{0, static_cast<double>(i + 1)};
169
170 #pragma omp simd
171         for (int j = 0; j < numberOff2Data; j++)
172         {
173             features[j + 1] = svm_node{j + 1, calculateKernel(xi, std::valarray<double>(x2[j].data(), x2[j].size()), parameter)};
174         }
175
176         features[numberOff2Data + 1] = svm_node{-1, 0};
177
178         std::move(features.begin(), features.end(), problem.x[i]);
179     }
180
181     return problem;
182 }

```

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

Arguments

`x1`: \mathbf{x} , shape [N, pixels]
`x2`: \mathbf{x}' , shape [N]
`parameter`: SVM parameter

Formula

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

```

138     double calculateKernel(const std::valarray<double> &x1, const std::valarray<double> &x2, const svm_parameter &parameter)
139     {
140         auto linear = (x1 * x2).sum();
141         auto rbf = std::exp(-parameter.gamma * std::pow((x1 - x2), 2).sum());
142         return linear + rbf;
143     }

```

- `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.

```

353 svm_parameter findCSVCParametersByGridSearch(const std::vector<std::vector<double>> &x, std::vector<double> &y, const svm_parameter &defaultParameter, const GridSearchSettings &settings)
354 {
355     auto degreeList = generateSequence(settings.degree);
356     auto coef0List = generateSequence(settings.coef0);
357     auto gammaList = generateSequence(settings.gamma);
358     auto CList = generateSequence(settings.C);
359
360     double bestAccuracy = 0;
361     svm_parameter bestParameter = defaultParameter;
362 #pragma omp parallel for collapse(4)
363     for (auto degree : degreeList)
364     {
365         for (auto coef0 : coef0List)
366         {
367             for (auto gamma : gammaList)
368             {
369                 for (auto C : CList)
370                 {
371                     svm_parameter parameter = defaultParameter;
372                     parameter.degree = degree;
373                     parameter.coef0 = std::pow(2, coef0);
374                     parameter.gamma = std::pow(2, gamma);
375                     parameter.C = std::pow(2, C);
376
377                     auto problem = makeKernel(x, x, y, parameter);
378
379                     std::vector<double> targets(problem.l);
380                     svm_cross_validation(&problem, &parameter, settings.kFold, targets.data());
381
382                     auto accuracy = evaluate(problem, targets);
383                     releaseProblem(problem);
384
385 #pragma omp critical
386                     if (accuracy > bestAccuracy)
387                     {
388                         bestParameter = parameter;
389                         bestAccuracy = accuracy;
390                     }
391                 }
392             }
393         }
394     }
395 }

```

Experiments settings and results

Default Settings

```
196 ✓ svm_parameter createSVMParameter(int numOfFeatures)
197 {
198     svm_parameter parameter;
199     parameter.svm_type = C_SVC;
200     parameter.kernel_type = LINEAR;
201     parameter.degree = 3;
202     parameter.gamma = 1 / static_cast<double>(numOfFeatures); // 1/num_features
203     parameter.coef0 = 0;
204     parameter.nu = 0.5;
205     parameter.cache_size = 100;
206     parameter.C = 1;
207     parameter.eps = 1e-3;
208     parameter.p = 0.1;
209     parameter.shrinking = 1;
210     parameter.probability = 0;
211     parameter.nr_weight = 0;
212     parameter.weight_label = nullptr;
213     parameter.weight = nullptr;
214     return parameter;
215 }
```

Part1

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

```
2 Part 1
3 Linear Model
4 Start training ...
5 Start predicting ...
6 Start evaluating ...
7 Accuracy: 0.9508
8
9 Polynomial Model
10 Start training ...
11 Start predicting ...
12 Start evaluating ...
13 Accuracy: 0.3452
14
15 RBF Model
16 Start training ...
17 Start predicting ...
18 Start evaluating ...
19 Accuracy: 0.9532
20
```

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.

```
15  Best Cross Validation Accuracy: 0.9714
16  Degree: 3
17  Coef0: 1
18  Gamma: 1.00088
19  C: 0.03125
20  Start training ...
21  Start predicting ...
22  Start evaluating ...
23  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

```
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
```

- RBF * Linear kernel

```
244    Best Cross Validation Accuracy: 0.9856
245    Degree: 3
246    Coef0: 1
247    Gamma: 0.015625
248    C: 256
249    Start training ...
250    Start predicting ...
251    Start evaluating ...
252    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
```