A Short Note on Quadratic Programming

When solving the dual form of the SVM, we face a specific type of optimization problem known as a quadratic program. Quadratic programming involves minimizing (or maximizing) a quadratic objective function subject to linear constraints:

$$\min_{x \in \mathbb{R}^n} \quad \frac{1}{2} x^\top P x + q^\top x$$

subject to

$$Gx \leq h, \quad Ax = b,$$

where $P \in \mathbb{R}^{n \times n}$ is a symmetric positive semidefinite matrix, $q \in \mathbb{R}^n$ is a vector, $G \in \mathbb{R}^{m \times n}$ and $h \in \mathbb{R}^m$ define the inequality constraints, and $A \in \mathbb{R}^{p \times n}$, $b \in \mathbb{R}^p$ define the equality constraints.

The term *quadratic programming* comes from the fact that the objective function contains quadratic terms (such as $x^\top P x$, in contrast to linear programming, where both the objective function and the constraints are linear.

A matrix P is called symmetric if it is equal to its transpose, that is, $P = P^\top$. It is called positive semidefinite if for all vectors $x \in \mathbb{R}^n$, the quadratic form satisfies

$$x^\top P x \geq 0.$$

This property ensures that the objective function is convex, meaning that any local minimum is also a global minimum, which is crucial for efficiently solving quadratic programs.

To understand how quadratic programming works in practice, we now introduce a simple real-world example that shows how quadratic programming problems arise and how they can be solved using Python.

Example

Imagine you are investing money in two assets: stocks and bonds. You want to allocate your investment in a way that minimizes the overall risk. In finance, the risk of an investment is often quantified using the variance or covariance of returns. The idea is that if two assets tend to move together, the overall portfolio is riskier. If they move independently or in opposite directions, the risk is reduced. Mathematically, the total risk of the investment can be expressed as a quadratic function of the investment proportions.

Let $x = (x_1, x_2)$ represent the fractions of your money invested in stocks and bonds, respectively. The risk is given by the expression:

$$\text{Risk} = \frac{1}{2} x^\top P x$$

where P is the covariance matrix of returns. Each entry of P has a specific meaning: P_{11} measures how much stock returns fluctuate on their own (the variance of stocks), P_{22} measures the variance of bond returns, and $P_{12} = P_{21}$ measures the covariance between stock and bond returns. A positive covariance means that stocks and bonds tend to move together, while a negative covariance would indicate that they move in opposite directions.

Let us assume that the covariance matrix is:

$$P = \begin{bmatrix} 0.1 & 0.05 \\ 0.05 & 0.2 \end{bmatrix}$$

In this setup, stocks have a variance of 0.1, bonds have a higher variance of 0.2, and their returns are positively correlated with a covariance of 0.05. Our goal is to minimize the risk while investing all of your available money and ensuring that no negative investments are made (no short selling).

The optimization problem can be formally stated as:

$$\min_x \quad \frac{1}{2} x^\top P x$$

subject to:

$$x_1 + x_2 = 1, \quad x_1 \geq 0, \quad x_2 \geq 0$$

This says that the sum of investments must be exactly one (you invest all your money) and each investment must be non-negative.

We can solve this quadratic program using Python and the `cvxopt` library. The `cvxopt` package is designed for convex optimization and can efficiently solve quadratic problems. The following code demonstrates how to set up and solve the problem:

```
import numpy as np
from cvxopt import matrix, solvers

# covariance matrix
P = matrix([[0.1, 0.05],
            [0.05, 0.2]])

# we do not have any linear term in the objective
q = matrix([0.0, 0.0])

# constraints: Gx <= h for x1 >= 0, x2 >= 0
G = matrix([[-1.0, 0.0],
```

```
            [0.0, -1.0]])
h = matrix([0.0, 0.0])

# constraint: Ax = b for x1 + x2 = 1
A = matrix([[1.0], [1.0]])   # Note: A must have size (2, 1)
b = matrix([1.0])

# here we solve the quadratic program
solution = solvers.qp(P, q, G, h, A, b)

# extract and display solution
x = np.array(solution['x']).flatten()
print("Optimal portfolio allocation:", x)
```

Let us walk through various components of this code, where each matrix and vector has a specific role that corresponds to part of the mathematical formulation:

- The matrix P represents the quadratic part of the objective function. The objective we are minimizing is

$$\frac{1}{2}x^\top Px + q^\top x,$$

  where in our case $q = 0$, so the objective reduces to minimizing only the quadratic term. In our example, P is the covariance matrix of asset returns, encoding how much each asset fluctuates on its own (the variances) and how much the two assets fluctuate together (the covariance). The diagonal elements P_{11} and P_{22} represent the variance of stocks and bonds, respectively, while the off-diagonal elements $P_{12} = P_{21}$ represent the covariance between stocks and bonds.

- The vector q represents the linear part of the objective function. Since there is no linear component in the risk function, q is simply the zero vector.

- The matrix G and the vector h encode the inequality constraints. Inequality constraints are written as

$$Gx \leq h,$$

  and in this case, they enforce that the investment fractions x_1 and x_2 must be non-negative:

$$x_1 \geq 0, \quad x_2 \geq 0.$$

  These conditions ensure that no negative investments (no short selling) are allowed.

- The matrix A and the vector b encode the equality constraints. Equality constraints are written as

$$Ax = b,$$

and here they enforce that the entire available amount is invested, meaning:

$$x_1 + x_2 = 1.$$

This guarantees that the full investment is allocated between stocks and bonds without any leftover.

When setting up a quadratic program, arranging the problem into the standard form with matrices P, q, G, h, A, and b is crucial, as solvers like cvxopt expect the input in exactly this structure. This same decomposition will be used in the next section to solve the dual form of the support vector machine.

When we run this code, the solver finds the optimal fractions of money to invest in stocks and bonds to achieve the minimum risk according to the given covariance matrix.

```
     pcost       dcost       gap    pres   dres
 0:  4.8915e-02 -9.7278e-01  1e+00  6e-17  2e+00
 1:  4.8045e-02  1.9093e-02  3e-02  1e-16  6e-02
 2:  4.3933e-02  4.1156e-02  3e-03  2e-16  4e-18
 3:  4.3750e-02  4.3676e-02  7e-05  1e-16  1e-17
 4:  4.3750e-02  4.3749e-02  7e-07  1e-16  4e-18
 5:  4.3750e-02  4.3750e-02  7e-09  1e-16  5e-18
Optimal solution found.
Optimal portfolio allocation: [0.74999986 0.25000014]
```

In convex optimization, the primal cost (`pcost`) refers to the value of the original optimization objective, while the dual cost (`dcost`) refers to the value of the dual optimization problem, which is mathematically derived from the primal by introducing Lagrange multipliers. The corresponding dual problem, in general, is:

$$\max_{\lambda,\nu} \; -\frac{1}{2}(G^\top \lambda + A^\top \nu)^\top P^{-1}(G^\top \lambda + A^\top \nu) - h^\top \lambda - b^\top \nu$$

subject to

$$\lambda \geq 0,$$

where λ and ν are the Lagrange multipliers associated with the inequality and equality constraints, respectively.

For convex problems, strong duality usually holds, meaning that at the optimal solution, the primal and dual costs should be equal. During the optimization process, the solver monitors both the primal and dual costs, and the difference between them, called the duality

gap, measures how close the current solution is to optimality. When the primal and dual costs converge and the duality gap becomes very small, the solver concludes that it has found an optimal solution.

The final result shows that approximately 75% of the money should be invested in stocks and 25% in bonds to minimize the overall portfolio risk according to the given covariance matrix.

If breaking up the optimization problem to bits and pieces as dictated by `cvxopt` is too much hassle, we can use `cvxpy`, where the quadratic programming problem is expressed directly in a high-level, math-like form and automatically transformed into a solver-ready representation:

```
import numpy as np
import cvxpy as cp

P = np.array([[0.1, 0.05],
              [0.05, 0.2]])
q = np.array([0.0, 0.0])
x = cp.Variable(2)

objective = cp.Minimize(0.5 * cp.quad_form(x, P) + q @ x)
constraints = [x >= 0, cp.sum(x) == 1]

# solve the quadratic program
problem = cp.Problem(objective, constraints)
problem.solve(verbose=False)
print("Optimal portfolio allocation:", x.value)
```

Note that in the above code, both the objective, given explicitly as the quadratic form $\frac{1}{2}x^T Px + q^T x$, and the constraints, written directly as mathematical expressions (non-negativity and a linear equality), are expressed, and the code is in fact simpler and hopefully more readable.

Ideas for experiments

Now that we know how to use the `cvxopt` and `cvxpy` libraries, and because in the lecture notes on Kernels there were examples with the former, we would like to encourage students to code some use cases with the later, simpler library. For instance:

1. Implement SVM with linear and polynomial kernels and compare them on some 2D data set, counting the number of support vectors and observing their dependency on regularization. With 2D problems, it is always interesting to plot the results and decision boundaries, just like we did in the lecture notes.

2. For a simple low-dimensional example, explicitly construct the feature mapping corresponding to a polynomial kernel and compare the results with the kernelized implementation. This helps illustrate concretely what the kernel trick is avoiding computationally.

3. Study the scalability of kernel methods by gradually increasing the size of the data set. Measure the time and memory required to compute the Gram matrix and solve the quadratic program, and relate this to the theoretical $\mathcal{O}(n^2)$ complexity.

4. Compare the dual-based implementation of SVM with a library implementation (such as from `scikit-learn`). Verify that the solutions (support vectors, decision boundary) are consistent, and investigate any differences due to numerical precision or solver details.

5. Explore the influence of kernel hyperparameters. For the polynomial kernel, vary the degree d; for the RBF kernel, vary the parameter γ. Observe how these choices affect the flexibility of the decision boundary, the number of support vectors, and the tendency to overfit.

6. Investigate the effect of the regularization parameter C in soft-margin SVMs. For a fixed data set, vary C over several orders of magnitude and observe how the margin width, number of support vectors, and classification accuracy change. Plotting the results can help illustrate the trade-off between margin maximization and error minimization.

7. Collect a set of interesting data sets with binary classifications, and, using cross-validation, compare the accuracy of linear SVM, SVM with some more complex kernels, logistic regression, and kernelized logistic regression. An interesting data set would for instance be that from molecular biology, with few samples (say, up to 100 data instances) and tens of thousands of attributes. Turns out that linear SVMs performed well on this data set, which is strange as the decision boundaries really should not be linear in this domain, and also, since the cross-entropy scoring should favor variants of logistic regression. Comparisons of kernelized logistic regressions in this domain have been rare, so this would be worth exploring and understanding why a particular method works best.

Use critical distance graph described in Demšar's Statistical Comparisons of Classifiers over Multiple Data Sets (JMLR, 2006) to figure out if any of the methods win.