COMP0120 Project

Griffin Bassman (19141838)

May 19, 2020

Motivation for SVMs: Classification

Classification is the problem of identifying which category an observation belongs to based on its features and prior knowledge. An algorithm which solves the classification problem should be able to identify what information is important in a new observation, and be able to use this information to correctly classify it. There are many classes of algorithms geared toward solving the classification problem. One which has been thoroughly investigated, and has proven highly successful is the Support Vector Machine (SVM). SVM is a supervised classification method which creates decision surfaces which maximize the margin between classes. This surface is referred to as the optimal hyperplane.

Basic Structure of SVMs and Terminology

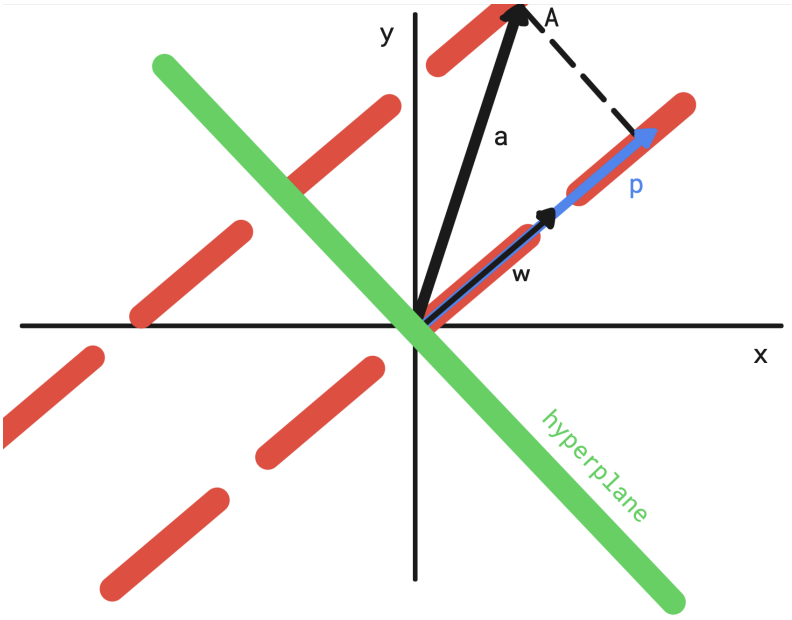
In a basic binary classification problem, we could consider having L training points, where each point x_i has D attributes and is considered to be in one of two classes $y_i = -1$ or $+1$. An example of this could be classifying flowers as either roses or daisies based on numerical features such as height and width. A single flower would have a feature vector x_i which would contain its measurements, as well as a binary value y_i classifying it as either a rose or a daisy. We can assume for now that height and width are highly correlated with flower type and that the data is linearly separable, meaning you can draw a line between the two classes such that all data points from each class are on opposite sides. This idea can be extended for $D > 2$ by drawing a hyperplane which can completely separate both classes. In this case, each data point might have more than just two features (ie data on mass and water concentration could also be included). We can define this hyperplane by the equation $\mathbf{w} \cdot \mathbf{x} = 0$ where $\mathbf{w}$ is normal to the hyperplane. For example if $D = 2$, a hyperplane would satisfy the equation $y = ax + b$, which can be rewritten as $y - ax - b = 0$. We could write this in vector notation as

$$\mathbf{w} = \begin{pmatrix} -b \\ -a \\ 1 \end{pmatrix} \text{ and } \mathbf{x} = \begin{pmatrix} 1 \\ x \\ y \end{pmatrix}$$

We can then define the unit vector of $\mathbf{w}$

$$\mathbf{u} = \frac{\mathbf{w}}{||\mathbf{w}||}$$

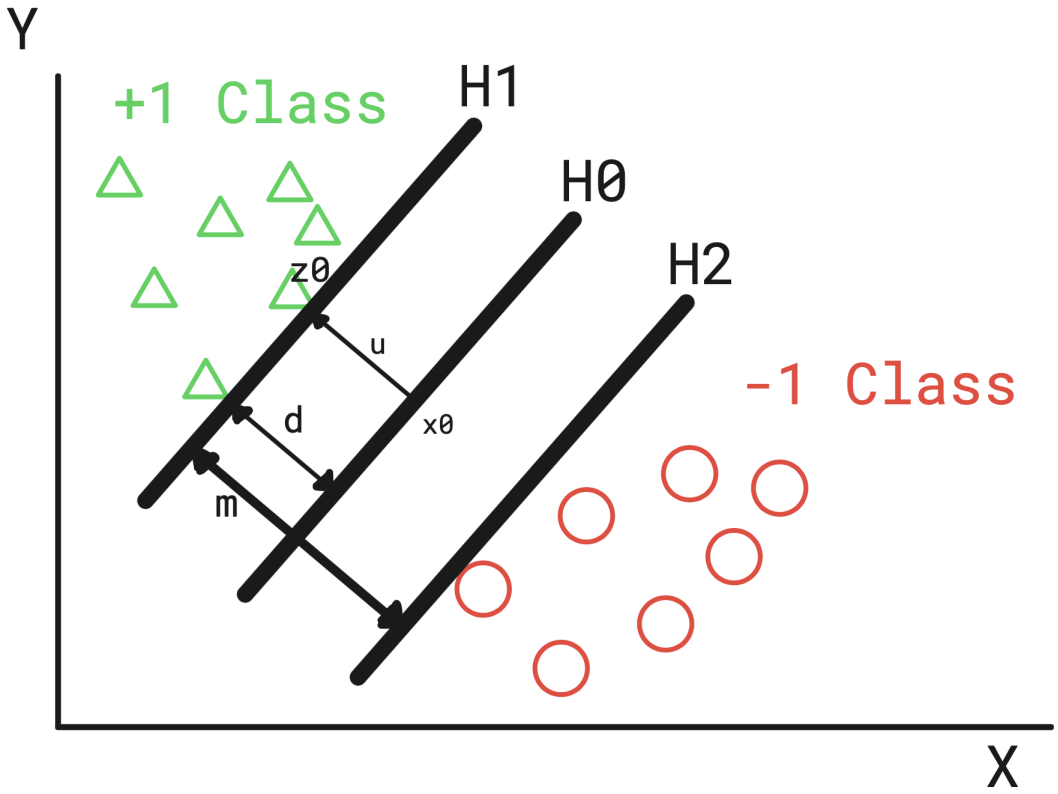
Which is important in defining the distance from this hyperplane to any data point. Consider the image below



If A is a data point and $\mathbf{a}$ describes the vector to this point, then we can define $\mathbf{p} = (\mathbf{u} \cdot \mathbf{a})\mathbf{u}$ as the projection of point A onto the plane of the $\mathbf{w}$ vector. The distance of point A to the hyperplane is equal to the length of $\mathbf{p}$.

Margins and the Optimal Hyperplane

Using the definitions above, we can define the margin of an SVM as twice the length of the nearest data point to this hyperplane. Thus if $\mathbf{p}$ defines the vector to the nearest point, then the margin $2m = ||\mathbf{p}||$. We will then define the optimal hyperplane as that which minimizes the margin m for a given set of data. We will also define *support vectors*, which are vectors parallel to the optimal hyperplane at a distance m on either side. These support vectors will optimally be touching the nearest data points on either side, as to widen the margin as much as possible. The fundamental task of an SVM is the maximize this margin m which separates the data. Consider the image below



Using our example from above, we could consider X and Y here to be measurements of a flower's height and width, and the $+1$ class to be roses and the -1 class to be daisies. This data is clearly linearly separable with an optimal hyperplane $H0$

which minimizes the margin m separating the two classes. Also included are the support vectors $H1$ and $H2$ which form a border to which all data from each class lie. More rigorously, we can define the optimal hyperplane using a weight vector $\mathbf{w}$ where $\mathbf{w} \cdot \mathbf{x} = 0$ and similarly the left support vector $H1$ with $\mathbf{w} \cdot \mathbf{x} = d$ and the right support vector $H2$ with $\mathbf{w} \cdot \mathbf{x} = -d$. Given a weight vector $\mathbf{x_i}$, we could thus classify $y_i = 1$ if $\mathbf{w} \cdot \mathbf{x} \geq d$, and conversely $y_i = -1$ if $\mathbf{w} \cdot \mathbf{x} \leq -d$.

Primal Problem for SVMs

Now that we have defined the basic elements of an SVM, we will pose an initial optimization problem for finding the optimal hyperplane. Consider a vector $d\mathbf{u}$ given the above definitions, with a starting point $\mathbf{x}_0$ on the optimal hyperplane H_0 and an ending point $\mathbf{z}_0$ which must lie on H_1 by definition. We will then use the definition $\mathbf{z}_0 = \mathbf{x}_0 + d\mathbf{u}$. Since $\mathbf{z}_0$ lies on H_1, we also know that $\mathbf{z}_0 \cdot \mathbf{w} = 1$. We can then expand this formula to:

$$\mathbf{w} \cdot (\mathbf{x}_0 + d\mathbf{u}) = 1$$

$$\mathbf{w} \cdot (\mathbf{x}_0 + d\frac{\mathbf{w}}{||\mathbf{w}||}) = 1$$

$$\mathbf{w} \cdot \mathbf{x}_0 + d||\mathbf{w}|| = 1$$

Then since we we know that $\mathbf{x}_0$ lies on H_0, we have $\mathbf{w} \cdot \mathbf{x}_0 = 0$. Thus we can reduce the above formula to:

$$d||\mathbf{w}|| = 1$$

$$d = \frac{1}{||\mathbf{w}||}$$

And thus for the total margin we have the equation:

$$m = \frac{2}{||\mathbf{w}||}$$

Thus maximizing the margin m is equivalent to minimizing $||\mathbf{w}||$, and this brings us our first equation to optimize an SVM:

$$minf(\mathbf{w}) = ||\mathbf{w}||$$

$$s.t.\ \ y_i(\mathbf{w} \cdot \mathbf{x}_i + w_0) \geq 1, 1 \leq i \leq n$$

A more practical form of this equation which can be optimized using quadratic programming is the **quadratic primal form**:

$$minf(\mathbf{w}) = \frac{1}{2}||\mathbf{w}||^2$$

$$s.t.\ \ y_i(\mathbf{w} \cdot \mathbf{x}_i + w_0) - 1 = 0, 1 \leq i \leq n$$

This form is beneficial because squaring length of $\mathbf{w}$ produces a parabola with a single minimum.

Dual Problem for SVMs

The dual problem for SVMs constructs the Lagrangian based on the objective function and constraint of the primal form. Given the objective function $f := \frac{1}{2}||\mathbf{w}||^2$ and the constraint $g := y_i(\mathbf{w} \cdot \mathbf{x}_i + w_0) - 1 = 0$ we can define the Lagrangian formulation as:

$$L(\alpha, \mathbf{w}, w_0) = \frac{1}{2}||\mathbf{w}||^2 - \sum_i \alpha_i[y_i(\mathbf{w} \cdot \mathbf{x}_i + w_0) - 1], \alpha_i \geq 0, \forall i$$

Where α_i are the Lagrange multipliers. We will then expand the summation to:

$$L(\alpha, \mathbf{w}, w_0) = \frac{1}{2}||\mathbf{w}||^2 - \sum_i \alpha_i y_i(\mathbf{w} \cdot \mathbf{x}_i + w_0) + \sum_i \alpha_i, \alpha_i \geq 0, \forall i$$

We can find the solution of this Lagrangian by setting the gradient along $\mathbf{w}$ to 0. This yields the equation:

$$\frac{\partial}{\partial w} L(\mathbf{w}, w_0) = \mathbf{w} - \sum_i \alpha_i y_i \mathbf{x}_i = 0$$

From which we get the solution:

$$\mathbf{w} = \sum_i \alpha_i y_i \mathbf{x}_i, \alpha_i \geq 0, \forall i$$

And by taking the partial derivative with respect to the bias w_0 we get the constraints:

$$\frac{\partial}{\partial w_0} L(\mathbf{w}, w_0) = \sum_i \alpha_i y_i = 0$$

$$\sum_i \alpha_i y_i = 0, \alpha_i \geq 0$$

Finally, using Lagrangian duality, we can reformulate this optimization problem as:

$$maxL(\alpha) = \sum_i \alpha_i - \frac{1}{2} \sum_i \sum_j \alpha_i \alpha_j y_i y_j \mathbf{x}_i \cdot \mathbf{x}_j$$

$$s.t.\ \ \alpha_i \geq 0$$

$$\sum_i \alpha_i y_i = 0$$

Hard vs Soft Margin Problem

Thus far in our discussion, we have only considered data sets which are completely linearly separable. In other words, we have assumed that there is always at least one hyperplane which can separate the data entirely into two categories. In most real-world applications, data will not be this consistent, and we will need to ease the restrictions on our optimization. We define a *soft margin* SVM as one where some data points will be misclassified, and the points which are further from the margin will incur a steeper penalty. A *hard margin* SVM alternatively does not allow any points to be misclassified, and assumes the data is linearly separable. Creating a soft margin SVM can be achieved by introducing a positive *slack variable* ξ_i. We will reformulate our initial constraints to include ξ_i with the following definitions:

$$\mathbf{x}_i \cdot \mathbf{w} + w_0 \geq 1 - \xi_i;\ y_i = 1$$

$$\mathbf{x}_i \cdot \mathbf{w} + w_0 \leq -1 + \xi_i;\ y_i = -1$$

$$\xi_i \geq 0, \forall_i$$

We can then combine these constraints into the more compact form:

$$y_i(\mathbf{x}_i \cdot \mathbf{w} + w_0) - 1 + \xi_i \geq 0$$

$$\xi_i \geq 0, \forall_i$$

With these initial constraints, we can adapt the primal and dual forms to a soft margin SVM. First we will define a control parameter C which will determine the trade off between the size of the margin and the penalty of the slack variable. We will then reformulate the primal form for a soft margin SVM as follows:

$$minf(\mathbf{w}) = \frac{1}{2}||\mathbf{w}||^2 + C \sum_i \xi_i$$

$$s.t.\ \ y_i(\mathbf{x}_i \cdot \mathbf{w} + w_0) - 1 + \xi_i \geq 0 \forall_i$$

Similarly, you can construct the soft margin SVM for the dual form by updating the Lagrangian with an additional feature $\mu_i \geq 0$ as follows:

$$L = \frac{1}{2}||\mathbf{w}||^2 + C\sum_i \xi_i - \sum_i \alpha_i y_i(\mathbf{w}\cdot\mathbf{x}_i + w_0) + \sum_i \alpha_i - \sum_i \mu_i\xi_i$$

Then by differentiating this with respect to $\mathbf{w}$, w_0 and x_i and setting this to 0 we arrive at the following:

$$\mathbf{w} = \sum_i \alpha_i y_i \mathbf{x}_i$$

$$\sum_i \alpha_i y_i = 0$$

$$C = \alpha_i + \mu_i$$

From here we arrive at the optimization problem for soft margin SMVs in the dual form:

$$max_\alpha[\sum_i \alpha_i - \frac{1}{2}\boldsymbol{\alpha}^T\mathbf{H}\boldsymbol{\alpha}]$$

$$s.t.\ \ 0 \leq \alpha_i \leq C$$
$$\sum_i \alpha_i y_i = 0$$

Where $H_{ij} = y_i y_j \mathbf{x}_i\mathbf{x}_j$. Once we have solve for α_i, we can easily compute $\mathbf{w}$ by using the definition above. To determine the values of w_0 we will first note that for any point satisfying the above constraints is a support vector $\mathbf{x}_s$ where:

$$y_s(\mathbf{x}_s + w_0) = 1$$

$$y_s\sum_{m\in S}\alpha_m y_m \mathbf{x}_m\cdot\mathbf{x}_s + w_0 = 1$$

Where S is the set of support vectors which satisfy $\alpha_i > 0$. We can then multiply both sides by y_s and note that $y_s^2 = 1$ to get a clean definition of w_0:

$$y_s^2\sum_{m\in S}\alpha_m y_m \mathbf{x}_m\cdot\mathbf{x}_s + w_0 = y_s$$

$$w_0 = y_s - \sum_{m\in S}\alpha_m y_m \mathbf{x}_m\cdot\mathbf{x}_s$$

Then instead of using only one support vector x_s, we will take an average over all support vectors, resulting in the final formula:

$$w_0 = \frac{1}{N_s}\sum_{s\in S}(y_s - \sum_{m\in S}\alpha_m y_m \mathbf{x}_m\cdot\mathbf{x}_s)$$

Where N_s is the number of support vectors which satisfy $\alpha_i \geq 0$.

Kernels

So far in our discussion of SVMs, we have only considered using linear combinations of our input attributes. Often data will have an optimal hyperplane which is not linear, and in these cases it is important to adjust our model. In these cases we can use feature mapping. We define a function ϕ which will map an attribute x to a feature $\phi(x)$. For instance, if we want an optimal hyperplane which will consider the squares of the input attributes, we could define:

$$\phi(x) = \begin{bmatrix} x \\ x^2 \end{bmatrix}$$

Now we can update our model by replacing all of our original attribute x with $\phi(x)$. As our algorithm only uses the dot product of attributes $< x, z >$, we will only need to redefine this function. This brings us to our definition of a kernel. For a given feature mapping $\phi(x)$, we will define the corresponding kernel as $K(x, z) = \phi(x)^T \phi(z)$. The use of higher-order kernels will allow the optimal hyperplane to have a higher degree of flexibility and will often allow you to better separate data. It is important however to not use too high of a degree, as this may result in overfitting. This is because the model will be too flexible, and in trying to correctly classify all the data it may extend the optimal hyperplane to include outliers which do not represent the data accurately.

Multi-Class Classification

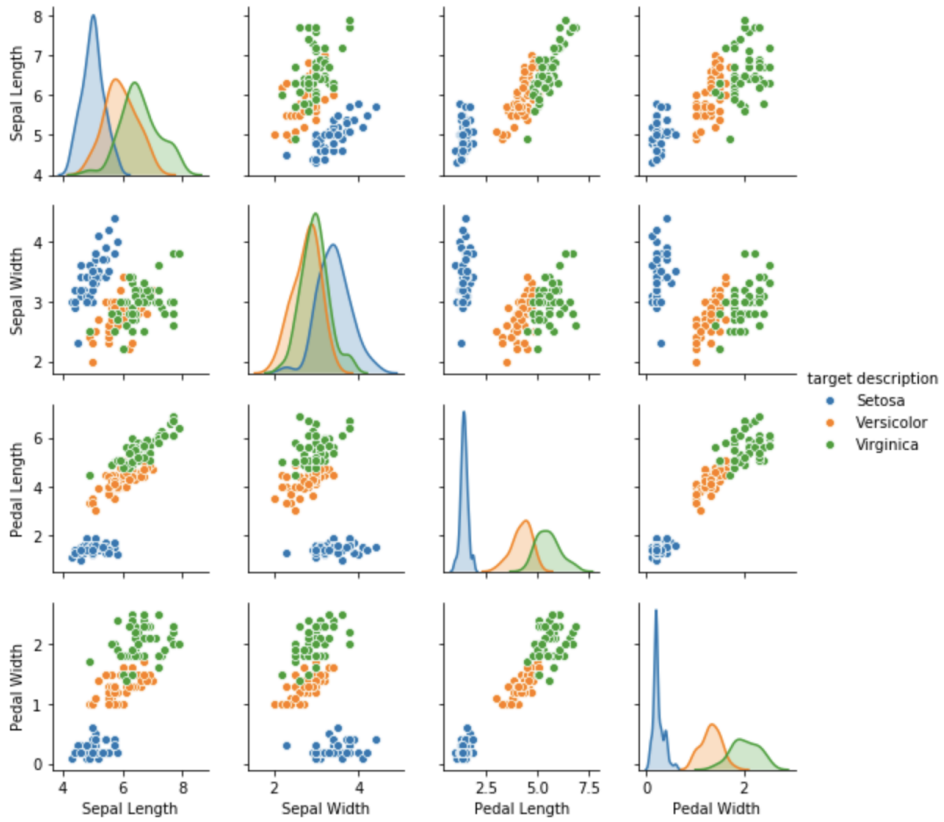
Using the basic definitions and tools we have discussed so far, we can extend our model to cover more than two classes. The most straightforward method of multi-class classification is the "One vs Rest" method. Suppose we have a dataset with k classes. In this method, we would create k SVMs, one for each class. For a given class' SVM, we would set $y = 1$ for the class we are considering, and $y = -1$ for all other classes. We will then omit the final step of our binary classifier where we classify based on the sign of $\mathbf{w} \cdot \mathbf{x} + b$, and instead use that value across all the SVMs for classification. More specifically, we will set an instance's class based on the formula:

$$\hat{y} = argmax_y f(y; \mathbf{w}, \mathbf{x})$$

Where y will be one of the output classes. Thus classes will not simply be determined by which side they are in one of the hyperplanes, but also how far they are from they are from the hyperplane. This is an accurate method because datapoints which are further within the region of a class are more likely to be part of that class than those which are close to the separating hyperplane.

Dataset

I have decided to use Fisher's iris dataset as it allows me to use all of the SVM concepts I have discussed so far. This dataset contain 150 data points on 3 different classes of iris flowers: Virginica, Setosa, and Versicolor. For each flower it includes 4 attributes: sepal length, sepal width, pedal length, pedal width. Below is an image of this dataset plotted in 2-dimensions based on all of their attributes:



I have decided to use the sepal width and pedal length for my SVM as these attributes do a good job of separating the classes. This dataset is ideal for my study for many reasons. Firstly, the datapoints are not perfectly linearly separable, so they will require the use of soft margins when determining an optimal hyperplane. Next there are three classes which will allow for the use of multi-class classification via the one vs rest method. The dataset also calls for the use of kernels, as a line would not be sufficient in separating the data. Namely, when trying to separate the Versicolor class in the middle from the other two, there will need to be a curved hyperplane in order to isolate just this class. This dataset also contains a small enough number of points to allow for quick experimentation, and large enough to produce an interesting model.

Optimization study

For my optimization study I will compare the Quadratic Penalty method and the Sequential Minimal Optimization (SMO) algorithm to solve the dual problem for SVMs. I have reformulated it as a minimization problem:

$$min_\alpha[\frac{1}{2}\boldsymbol{\alpha}^T\mathbf{H}\boldsymbol{\alpha}] - \sum_i \alpha_i$$

$$s.t.\ \ 0 \leq \alpha_i \leq C$$
$$\sum_i \alpha_i y_i = 0$$

Where $H_{ij} = y_i y_j K(\mathbf{x}_i, \mathbf{x}_j)$ and K is a kernel function. I will be investigating two different kernel functions for my study, a cubic kernel function and the radial basis kernel function. The cubic kernel function will be defined as:

$$\phi_{cubic}(x) = \begin{bmatrix} x \\ x^2 \\ x^3 \end{bmatrix}$$

$$K_{cubic}(x_i, x_j) = \phi_{cubic(}x_i)^T \phi_{cubic}(x_j)$$

And the radial basis kernel will be defined as:

$$K_{rbf}(x_i, x_j) = exp(-\frac{||x_i - x_j||^2}{2\sigma^2})$$

It is important to note that ϕ_{cubic} and ϕ_{rbf} do not need to be explicitly defined as we can simply use the kernel function to replace every dot product between x_i, x_j. An extension of this is that we cannot use our generic formula to compute $\mathbf{w} = \sum_i \alpha_i y_i \phi(x_i)$, and we will instead classify data points directly from α using:

$$\mathbf{w} \cdot \mathbf{x} + b = \sum_i \alpha_i y_i K(\mathbf{x}_i, \mathbf{x}) + b$$

This minimization problem is convex with a quadratic objective function and both equality and inequality constraints. This type of problem is referred to as a "quadratic program," and has the general form:

$$min_x \; q(x) = \frac{1}{2} x^t G x + x^T c$$

$$s.t. \; c_i(x) = 0, \; i \in \mathcal{E}$$

$$c_i(x) \geq 0, \; i \in \mathcal{I}$$

Where c_i is a constraint, and $\mathcal{E}$ contains the indices of all equality constraints, and $\mathcal{I}$ contains the indices of all inequality constraints. Quadratic programs are a very common set of optimization problems which can be solved in finite computation, however the time is highly dependent on the complexity of the objective function and number of inequality constraints. One important aspect of quadratic programs is whether the matrix H is positive semidefinite. This would make the objective function convex, and thus the problem would have similar difficulty to a linear program, as there is only one global minima and no other local minima. H is an example of a Gramian matrix, which we can easily prove to be positive definite. In order for a matrix to be positive definite, we must have $v^t H v \geq 0 \; \forall v$. We can then derive:

$$v^t H v =$$

$$\sum_{i,j} < y_i \phi(x_i), y_j \phi(x_j) > v_i v_j =$$

$$\sum_{i,j} < y_i \phi(x_i) v_i, y_j \phi(x_j) v_j >=$$

$$< \sum_i y_i \phi(x_i) v_i \sum_j y_j \phi(x_j) v_j >=$$

$$|| \sum_i y_i \phi(x_i) v_i ||^2 \geq 0$$

A common strategy to tackle quadratic programs is to replace the original problem with a series of subproblems in which the constraints are added to the objective function. One such strategy is the quadratic penalty method, which adds a multiple of the square of each constraint violation to the objective function.

Quadratic Penalty Method

The fundamental idea behind the quadratic penalty method is the replace the constrained optimization problem with a single function consisting of the objective function, and a penalty term for each constrained, which grows as the constraint is further violated. In general terms, we will define this function as:

$$Q(x;\mu) = f(x) + \frac{\mu}{2}\sum_{i\in\mathcal{E}} c_i^2(x) + \frac{\mu}{2}\sum_{i\in\mathcal{I}}([c_i(x)]^-)^2$$

where $[y]^-$ is defined as $max(-y, 0)$. To put our optimization problem in this standard form, we must rearrange our two inequality constraints to the following standard form:

$$\alpha_i \geq 0 \; \forall_i$$

$$C - \alpha_i \geq 0 \; \forall_i$$

From here we can define the quadratic penalty function as:

$$Q(\boldsymbol{\alpha};\mu) = \frac{1}{2}\boldsymbol{\alpha}^T\mathbf{H}\boldsymbol{\alpha} - \sum_i \alpha_i + \frac{\mu}{2}(\sum_i \alpha_i y_i)^2 + \frac{\mu}{2}\sum_i([\alpha_i]^-)^2 + \frac{\mu}{2}\sum_i([C-\alpha_i]^-)^2$$

It is important to note that while the original objective function is smooth and continuously differentiable, this penalty function is not. Namely, the $[\alpha]^-$ function has a discontinuous second derivative, meaning that Q is not twice continuously differentiable. In order to compute the derivatives of the $[\alpha]^-$ function, we will redefine it as a piece-wise function:

$$[\alpha]^- = \begin{cases} -\alpha & \alpha < 0 \\ 0 & \alpha \geq 0 \end{cases}$$

We will then take piece-wise derivatives based on this function. The quadratic penalty method will then find a minimizer for the penalty function x_k, increase the value of μ_k by some constant factor, then compute a new minimizer starting from x_k, and repeat this process until a final convergence is satisfied. We will also define a sequence $\{\tau_k\}$ which will approach 0 and will be used as a tolerance for each minimization, where each iteration will terminate when $||\nabla_\alpha Q(\alpha;\mu_k)|| \leq \tau_k$. The constant factor to increase μ_k by on each iteration is flexible, but should be chosen sufficiently large so that the constraints are satisfied. Similarly, $\{\tau_k\}$ is rather flexible and only requires that $\tau_k \to 0$, making each estimate more accurate throughout the algorithm. One key issue with the quadratic penalty method is that the Hessian $\nabla^2_{\alpha\alpha}Q(\alpha;\mu_k)$ becomes arbitrarily ill-conditioned near the minimum. For this reason, Newton's method and the quasi-Newton algorithm perform poorly. I have thus decided to use gradient descent for each line search, as this does not require the Hessian. While global convergence of the quadratic penalty method is not strictly ensured, it is made easier with adequate parameters and a convex objective function. Supposing that each $\boldsymbol{\alpha}_k$ is an exact minimizer of the function $Q(\boldsymbol{\alpha},\mu_k)$ for a positive and monotonically increasing sequence $\{\mu_k\}$, then every limit point $\boldsymbol{\alpha}^*$ of the sequence $\{\boldsymbol{\alpha}_k\}$ is a global solution for the constrained optimization problem. As our objective function is convex and $\{\mu_k\}$ is monotonically increasing, we should expect global convergence to the constraint problem. This algorithm will use gradient descent with backtracking line search to solve a convex problem. We should expect to see $O(\frac{1}{k})$ sublinear convergence for problems of this class. Thus if $\{\mu_k\}$ increases at a sufficient rate, we should have a moderate convergence to the constraint problem.

SMO Algorithm

Instead of updating the entire vector $\boldsymbol{\alpha}$ at once, the SMO algorithm aims to update each α_i individually. Due to the constraint $\sum_i \alpha_i y_i = 0$ however, it would not be possible to update only one α_i without breaking this constraint. This is the inspiration for this SMO algorithm, which will instead pick a pair (α_i, α_j) at each iteration, and optimize this pair while keeping all other α_k constant. In my implementation of the SMO algorithm, I will sequentially iterate through all α_i and pick a random $\alpha_j \neq \alpha_i$ at each step. Once a pair (α_i, α_j) has been picked, we will optimize α_j with:

$$\alpha_j = \alpha_j - \frac{y_j(E_i - E_j)}{\eta}$$

$$E_k = f(x_k) - y_k$$

$$\eta = 2K(x_i, x_j) - K(x_i, x_i) - K(x_j, x_j)$$

$$f(x) = \sum_i \alpha_i y_i K(x_i, x) + b$$

Where K is some kernel function. We will then determine a lower and upper bound (L, H) which will satisfy $0 \leq \alpha_j \leq C$, and clip α_j accordingly. Next we will update α_i according to:

$$\alpha_i = \alpha_i + y_i y_j (\alpha_j^{(old)} - \alpha_j)$$

Next we must update b to satisfy the KKT conditions at the i and j indices. If α_i is not in the bounds $0 < \alpha_i < C$ then the threshold b_1 will ensure that the SVM will output y_i given x_i:

$$b_1 = b - E_i - y_i(\alpha_i - \alpha_i^{(old)})K(x_i, x_i) - y_j(\alpha_j - \alpha_j^{(old)})K(x_i, x_j)$$

We will similarly set a threshold b_2 for α_j:

$$b_2 = b - E_j - y_i(\alpha_i - \alpha_i^{(old)})K(x_i, x_j) - y_j(\alpha_j - \alpha_j^{(old)})K(x_j, x_j)$$

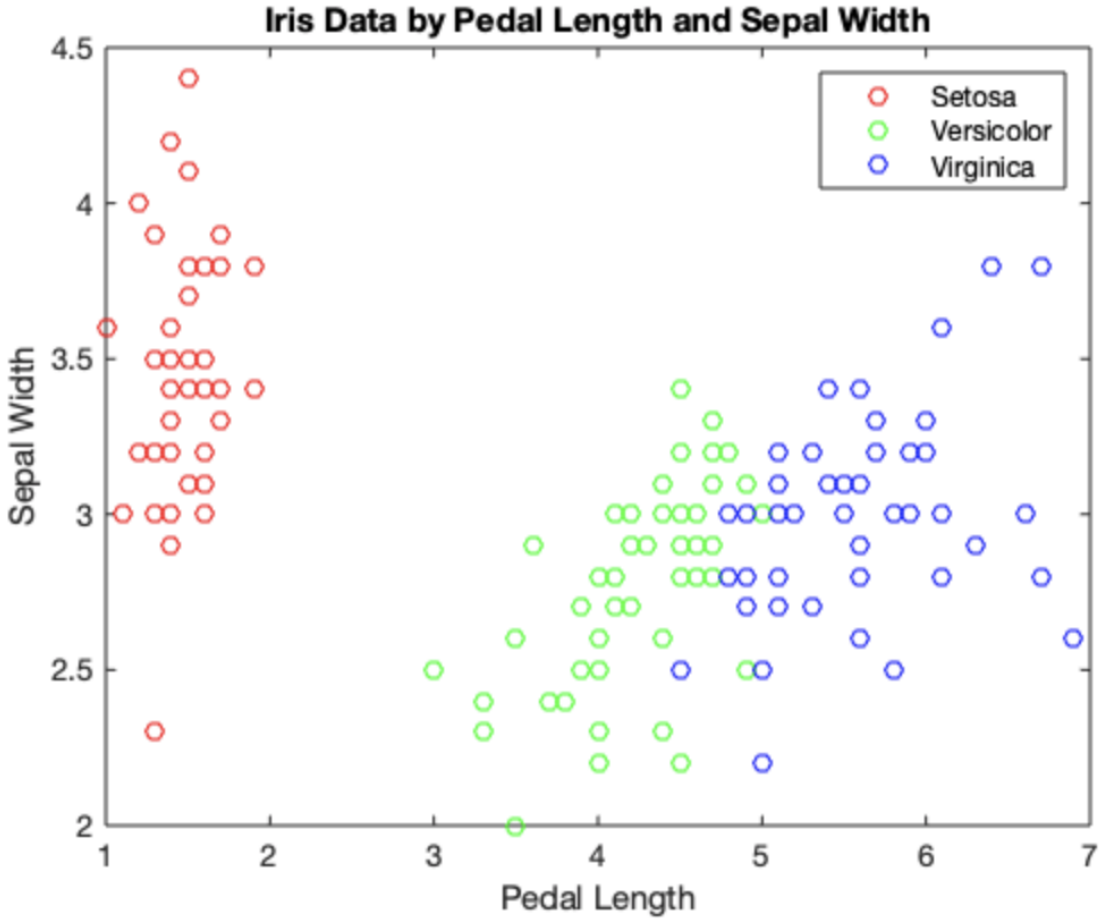
If both $0 < \alpha_i < C$ and $0 < \alpha_j < C$, then b_1 and b_2 will both be valid and equal thresholds. Otherwise if both α_i, α_j are on the bound 0 or C, then all thresholds between b_1 and b_2 will be valid. We will thus set b accordingly:

$$b = \begin{cases} b_1 & 0 < \alpha_i < C \\ b_2 & 0 < \alpha_j < C \\ \dfrac{(b_1 + b_2)}{2} & otherwise \end{cases}$$

We will then continue this algorithm until no α_i are updated for a certain number of passes through all α_i, where we only update some α_i if its error is above a certain tolerance. The convergence rate of the generalized SMO algorithm for a convex function is linear. While heuristics for selecting (α_i, α_j) may speed up the algorithm, convergence is still guaranteed without these. The convergence rate of the SMO algorithm is highly dependent on the dataset, and performs slower when the data is dense and requires many support vectors. On relatively small and sparse datasets however, the SMO algorithm has proven to be quite efficient. I therefore expect the SMO algorithm to perform better than the Quadratic Penalty method in terms of convergence.

Cubic and RBF Kernels

As discussed above, I will be investigating both the cubic and RBF kernels for my study. Kernels are necessary when a linear model is not sufficient to separate data. The correct kernel to use for an SVM problem is highly dependent on the data, as this kernel will determine the shape of the separator between classes. Below is an image of the data and attributes I am using for my study.



I have decided to investigate a cubic kernel because a cubic curve seems sufficient in separating all three classes by looking at the data. A quadratic curve would likely not have enough flexibility to properly separate the Versicolor class, as there must be a sharp curve in order to avoid both other classes. The cubic kernel will likely perform quickly when classifying the Setosa and Virginica classes, however it may not perform as quickly with the Versicolor class. This is because the Versicolor class is between two other classes, and thus will have support vectors on both sides. Since the quadratic penalty method updates the entire vector $\boldsymbol{\alpha}$ at once, it may prove difficult to consider the many additional support vectors. The SMO algorithm may also suffer since it updates a pair (α_i, α_j) at each iteration. Thus each update will prove more complicated as the pairs could be any combination of (Versicolor,Setosa), (Versicolor, Virginica), etc. This will likely result in each α_i needing more updates to reach an equilibrium, and thus slowing down convergence.

The RBF kernel is a combination of all polynomial kernels with degree $n > 0$. We thus say that the RBF kernel projects to an infinite-dimensional feature space. The RBF kernel has very high flexibility, and is often considered a standard when choosing a kernel. For our dataset, I believe the RBF kernel will outperform the cubic kernel since it will be superior at separating the Versicolor class. Since the RBF kernel uses a radial basis, it uses the distance between two points rather than a dot product between them at each iteration. This allows for a radial separator between classes. Thus when separating the Versicolor class, it will only matter how far away datapoints are from each other, not the direction between them. This will make it simpler to separate classes on either side. I expect that the RBF kernel will have quicker convergence than the cubic kernel as its flexibility will make it far easier to optimize compared to a less accurate cubic model.

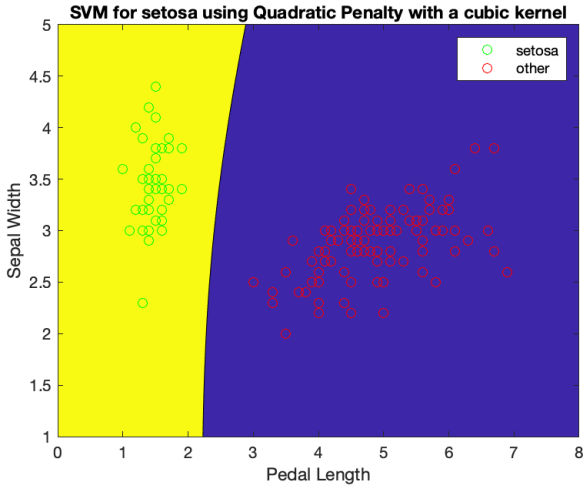
Parameter Choices

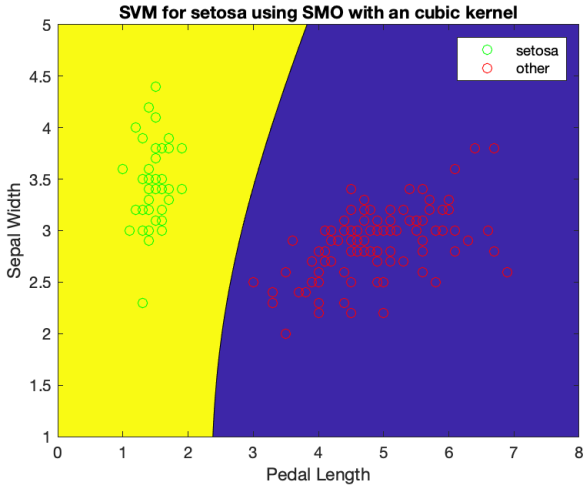
$maxIter = 400$
This is the maximum number of iterations for both algorithms with both kernels. The algorithms all seem to converge before this maximum, however it allows the algorithms to cut off reasonably quickly.
$C_{cubic} = 0.01$
This is the C value for both algorithms using the cubic kernel. A low C value allows for more slack and can increase the rate of convergence. Through inspection, it became clear that the cubic kernel required a fairly high degree of flexibility, specifically when classifying Versicolor.
$C_{RBF} = 0.2$
This value was also computed through inspection, and the RBF kernel proved to need less slack as the model was more

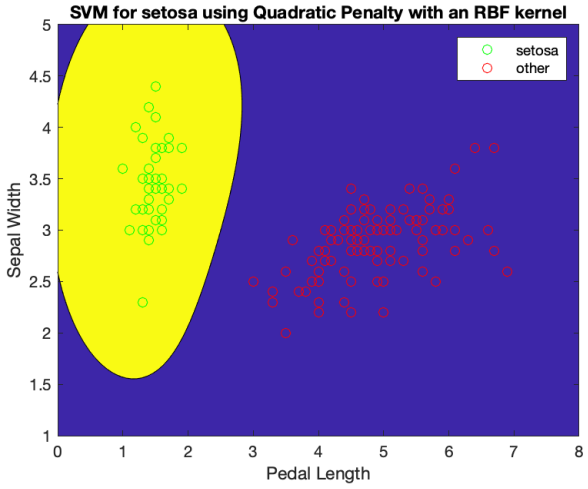
flexible and suited to the data.
$\sigma_{RBF} = 1.0$
σ will determine the reach of the RBF kernel. I high σ value will make for a high reach, meaning that points far away from the boundary will have a greater affect. $\sigma = 1.0$ is a standard parameter value and proved adequate for both algorithms.
$tol_{QP} = 10^{-8}$
This is the global tolerance used for the Quadratic Penalty method. This value was also computed through inspection, and it proved low enough to produce an accurate model, and high enough for reasonably quick convergence.
$\tau_{QP} = 10^{-12}$
This is the tolerance used for each gradient descent in the Quadratic penalty algorithm. This is a standard value which I arrived at through inspection.
$\tau_{fac} = 0.5$
This is the factor to decrease τ_{QP} at each iteration. The Quadratic Penalty only requires that $\{\tau_k\} \to 0$, thus 0.5 proved sufficient.
$\mu_{QP} = 1.0$
This is the μ used for Quadratic Penalty. I have set it very low at first so that the method could arrive at a reasonable minimizer of the objective function, and then further enforce the constraints at each iteration.
$\mu_{fac} = 1.1$
This is the factor the increase μ by in each iteration of Quadratic Penalty. I have set it relatively low so that the algorithm would not only optimize for the constraints, but also the objective function. The Quadratic Penalty method only requires that $\{\mu_k\} \to \infty$ and this factor proved sufficient.
$tol_{SMO} = 10^{-3}$
This is the tolerance used to determine if some α_i must be updated at each iteration of the SMO algorithm. This value was recommended in the paper from which I based my algorithm and proved sufficient.
$maxPasses_{SMO} = 5$
This is the maximum number of iterations the SMO algorithm can be unchanged before declaring convergence. This was determined through inspection and proved high enough to ensure a model was accurate.

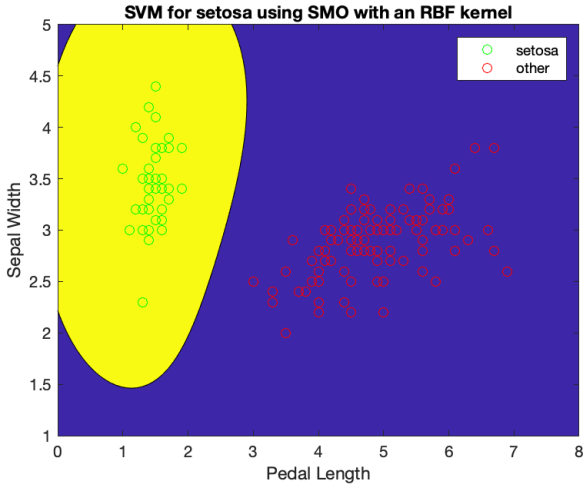
Visual Results of All SVMs

SVMs for Setosa class:

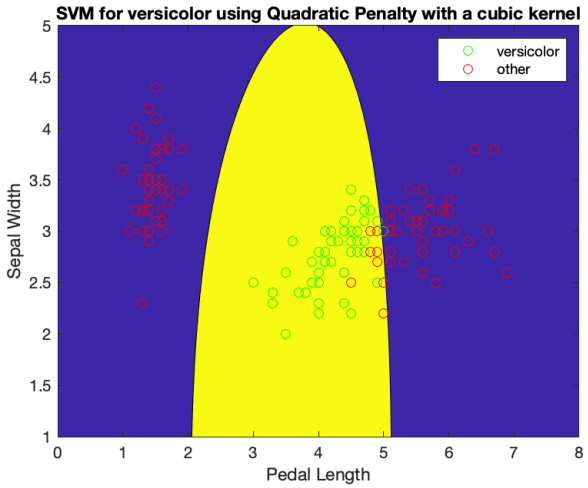


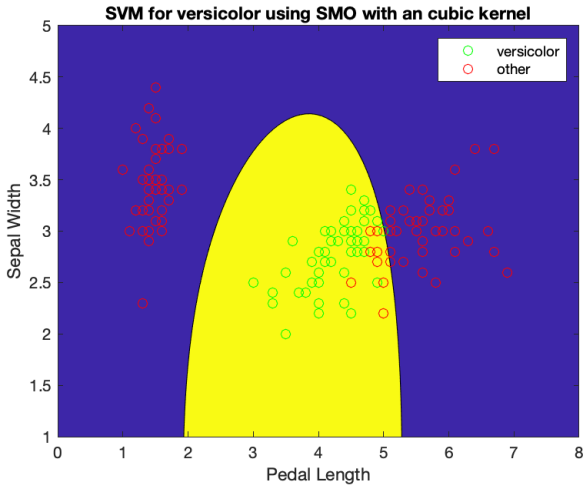


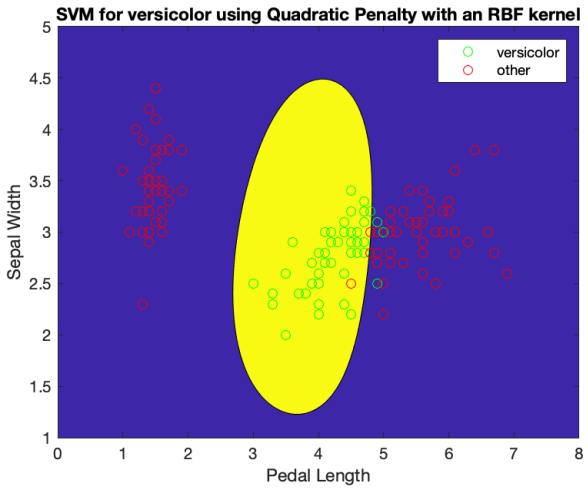


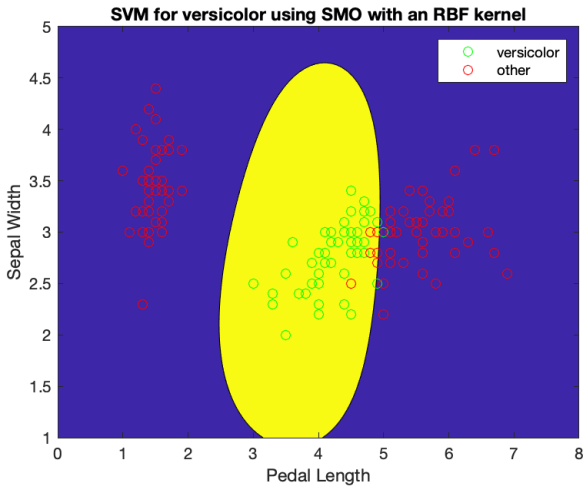


SVMs for Versicolor class:

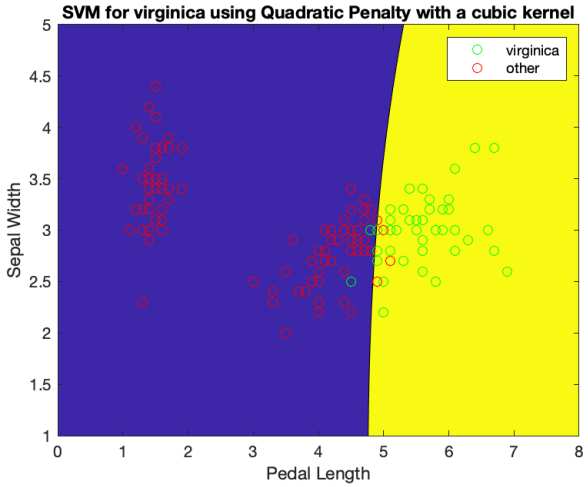


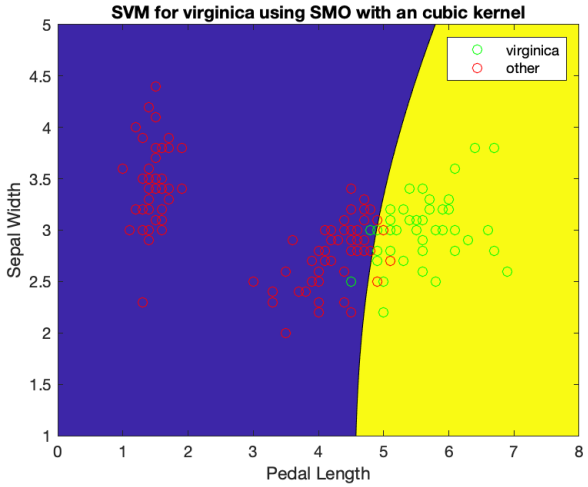


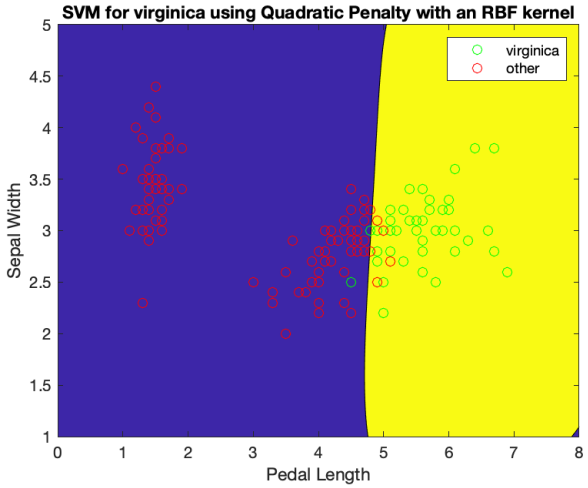


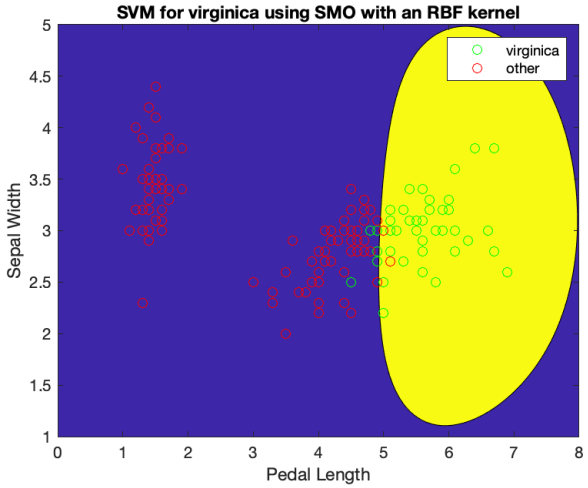


SVMs for Virginica class:

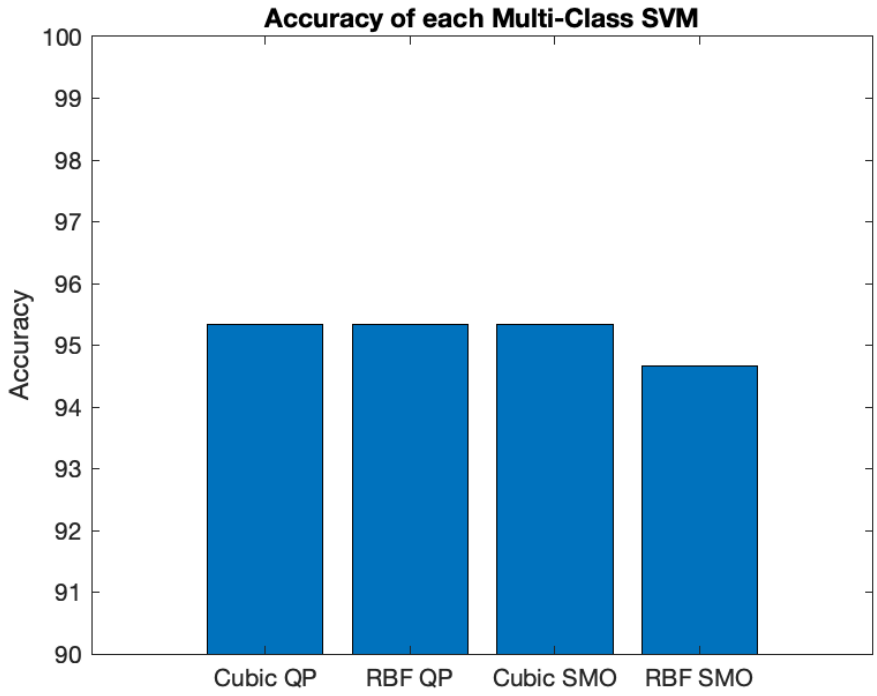








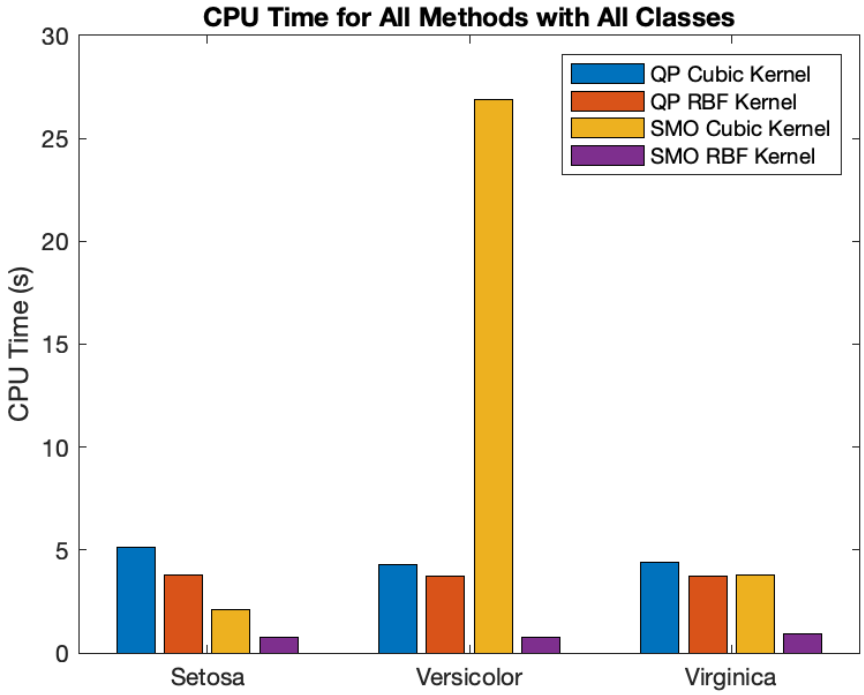
Multiclass SVM Accuracy for each Method:



Recall that my "one vs rest" implementation of a multi-class SVM called for a separate SVM for each class. Above I have shown each of these SVMs for both of the algorithms and both of the kernels I was investigating. The multi-class SVM

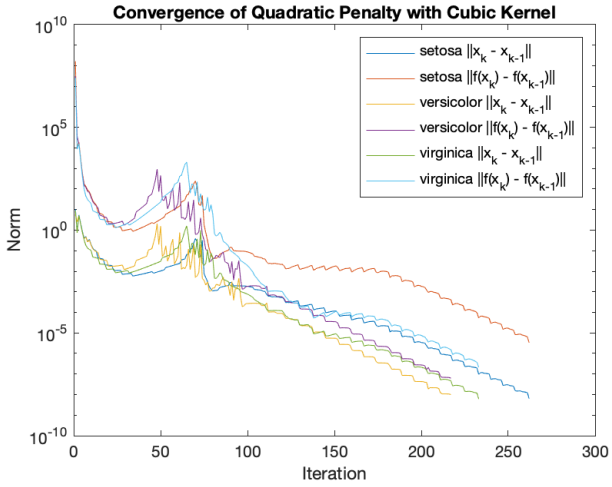
accuracy above applies the formula $\hat{y} = argmax_y f(y; \mathbf{w}, \mathbf{x})$ to classify the data, and shows what percent of the data has been classified correctly for each algorithm. It is clear that all algorithms worked very well as they achieved $\sim 95\%$ for data which was not entirely separable. This is an ideal accuracy as any higher could signify over fitting, and lower could signify an inaccurate model. From the plots above, it is clear to see that using a cubic kernel vs RBF produced different models. It is clear that the cubic kernel created a cubic boundary which proved sufficient for this data. The RBF kernel on the other hand proved to be extremely flexible, and could form any shape around data is was classifying. If there were more classes, or the data showed more radial tendencies, the cubic kernel would likely fail. This is because this kernel is restricted to only a cubic shape, whereas the RBF kernel has no such restrictions, being an infinite degree polynomial kernel. There also is not a clear difference in the models produced with Quadratic Penalty and SMO, which shows that both algorithms produced similar and accurate results.

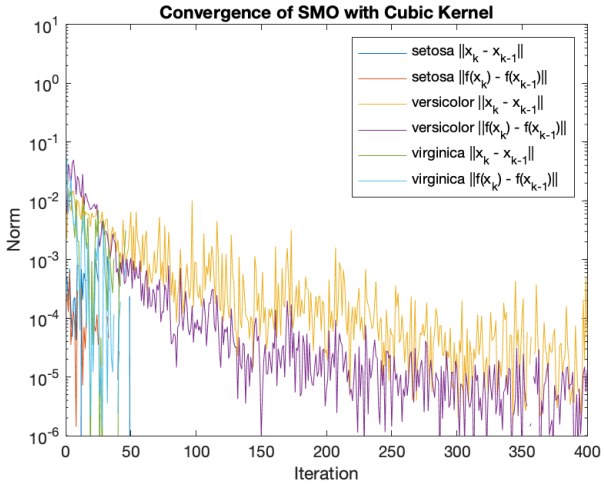
CPU Timing Results of All SVMs

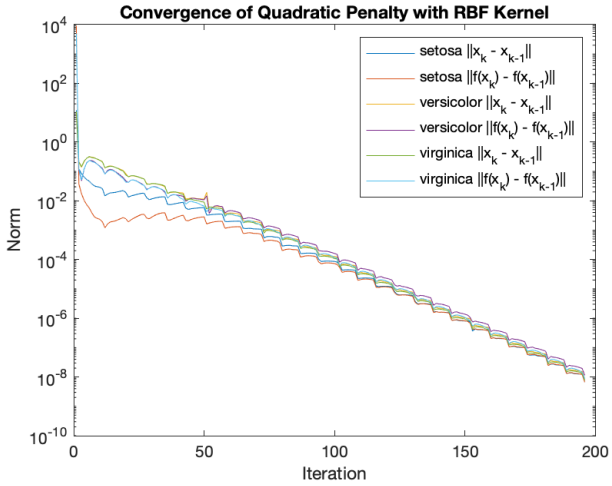


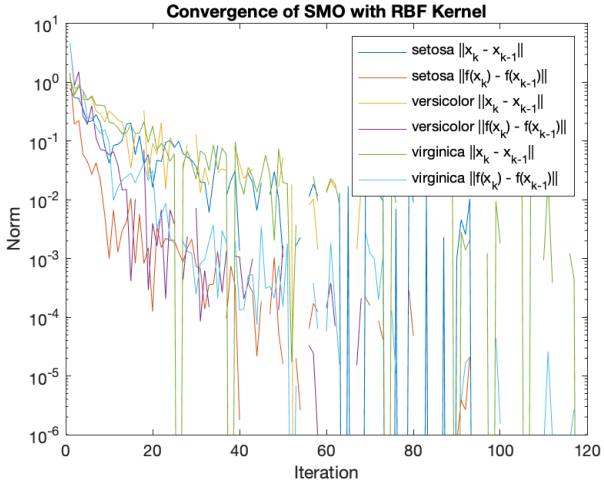
There are many interesting takeaways from the plot above. Firstly it has affirmed my assertion that the SMO algorithm would converge faster than the Quadratic Penalty, and the RBF kernel would converge faster than the cubic kernel, in general. The best combination is using the SMO algorithm with an RBF kernel. This is likely because SMO has the faster theoretical convergence rate, and the RBF kernel is more flexible, making it easier to converge to an accurate model. It is also interesting to note the large outlier when classifying Versicolor with the SMO algorithm and a cubic kernel. This slow compute time was due to the fact that the SMO algorithm optimizes between two random datapoints during each iteration, and this optimization could be drastically different if the two points were (Versicolor, Setosa) or (Versicolor, Virginica), as the two other classes lie on opposite sides of Versicolor. This is less of an issue with the Quadratic Penalty method since it optimizes all points at once, and thus is less affected by the large variability of selecting data on either side. This is also not an issue when using the RBF kernel, since this kernel looks at the distance between two points while disregarding the direction. This is a key example of how the RBF kernel is superior for multi-class SVMs, as it can isolate a specific class regardless of where the other classes lie.

Convergence Results for All SVMs



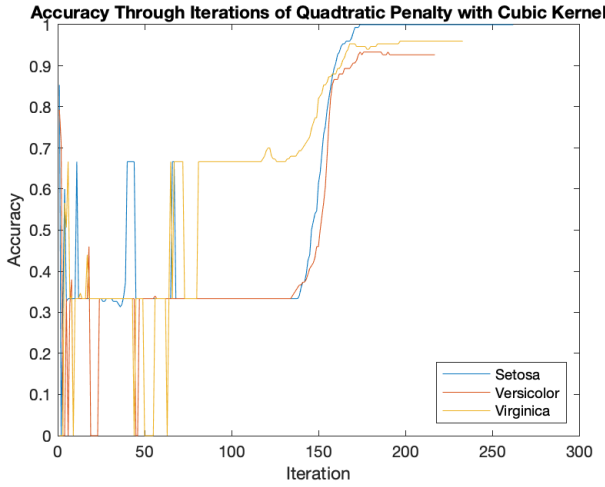


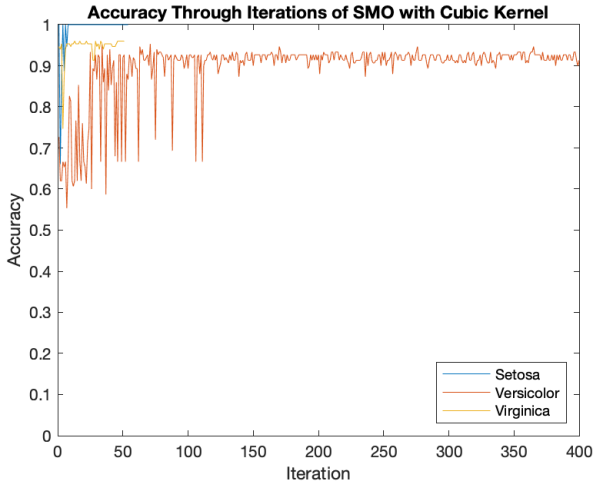


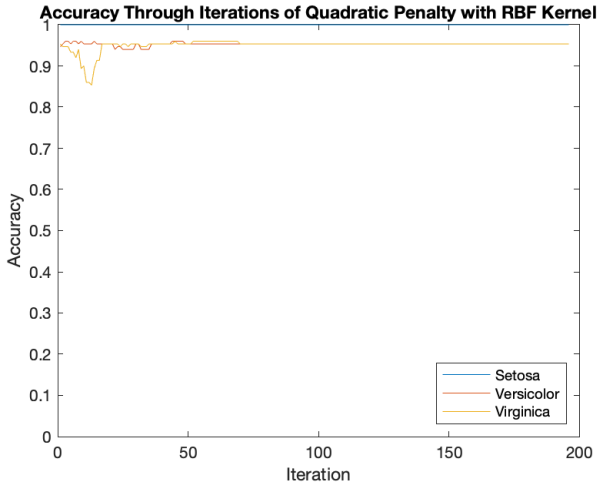


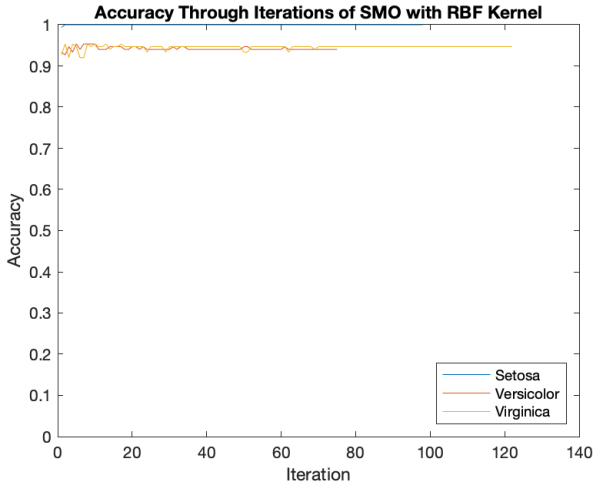
Above I have shown how the $||\alpha_k - \alpha_{k-1}||$ and $||f(\alpha_k) - f(\alpha_{k-1})||$ norms differ at each iteration for all SVMs. For the Quadratic penalty method, I used the penalty function with the constraints, and for the SMO algorithm I simply used the objective function. Note that the SMO algorithm has very jagged convergence plots because the updates are stochastic, and convergence is only reached once an adequate number of "non-update" iterations have passed. Thus many iterations have empty data because no α_i were updated. One key takeaway is that the SMO algorithm converges in less steps than the Quadratic Penalty method. The SMO algorithm has a linear convergence rate, whereas the Quadratic Penalty method is sub-linear, so this result is expected. This excludes the outlier of classifying Versicolor with SMO and a cubic kernel as discussed above. It is also clear from these plots that the RBF kernel converges quicker and more consistently than the cubic kernel. Using the quadratic penalty method, the RBF kernel has consistent and fast convergence, whereas the cubic kernel is far less consistent, as fitting a slightly inaccurate cubic model is more difficult. Additionally, the SMO algorithm with the RBF kernel requires the least iterations to converge, and does not suffer from the problem of classifying Versicolor, as the cubic kernel does.

Convergence of Accuracy for All SVMs









One key issue with comparing convergences between algorithms and kernels is that they all have different hyper parameters, which can be a trade off between quick convergence and high accuracy. While I tried to keep all hyper parameters as similar as possible for all models, it is not entirely clear if some model converged faster than others because their specific hyper parameters allowed for less accuracy. Above I have plotted the accuracy of each model throughout each iteration. This shows how quickly they became accurate regardless of stop conditions. The plots affirm my assertion that the SMO algorithm outperformed the Quadratic Penalty method, and the RBF kernel outperformed the cubic kernel. The Quadratic Penalty method with a cubic kernel took about 150 iterations to start producing accurate results, whereas the SMO algorithm with an RBF kernel was accurate after 1 or 2 iterations. This shows that the convergence conditions could be considered stricter for the SMO algorithm, and further proves that this algorithm was superior to the Quadratic Penalty method.

Description of Code Files

Project.ipynb
This is a jupyter notebook file which uses a Matlab kernel. It is the main component of my project which produces all SVMs and visualisations.

Project_python.ipynb
This is a regular Python notebook which I used to create a pairplot of Iris data.

backtracking.m
This implements the backtracking line search algorithm for gradient descent in Quadratic Penalty.

bFunc.m
This computes b in the Quadratic Penalty method.

binAcc.m
This computes the accuracy of a binary SVM.

descentLineSearch.m
This implements gradient descent for the Quadratic Penalty method.

getY.m
This creates a binary vector y based on the class you are using.

ineqfQP.m
This computes the inequality restraints for the Quadratic Penalty method.

ineqdfQP.m
This computes the gradient of the inequality restraints for the Quadratic Penalty method.

ineqd2fQP.m
This computes the hessian of the inequality restraints for the Quadratic Penalty method.

plotClass.m
This plots the SVM for a given class.

quadraticPenalty.m
This implements the Quadratic Penalty method.

SMO.m
This implements the SMO algorithm.

smoE.m
This computes the error E in the SMO algorithm.

SMOObj.m
This computes the objective function for the SMO algorithm.

svmQP.m
This implements the SVM algorithm using Quadratic Penalty.

value.m
This computes the value $\mathbf{w} \cdot \mathbf{x} + b$ indirectly with a kernel function and $\boldsymbol{\alpha}$.

References

[1] *Andrew Ng, CS229 Lecture notes , Support vector machines, Part V.*

[2] *Andrew Ng, CS229 Lecture notes , The Simplified SMO Algorithm*

[3] *Ben Aisen, A Comparison of Multiclass SVM Methods (2006)*

[4] *C. Campbell, An Introduction to Kernel Methods*

[5] *I. Necoara Y. Nesterov, & F. Glineur, Linear convergence of first order methods for non-strongly convex optimization (2015)*

[6] *J. Platt, Sequential minimal optimization: A fast algorithm for training support vector machines (1998)*

[7] *J. Shawe-Taylor & N. Cristianini, Kernel Methods for Pattern Analysis (2004)*

[8] *Jennifer She & Mark Schmidt, Linear Convergence and Support Vector Identification of Sequential Minimal Optimization (2017)*

[9] *Jorge Nocedal & Stephen Wright, Numerical Optimization, Second Edition (2006)*

[10] *Raj Bridgellal, Introduction to Support Vector Machines (2017)*

[11] *Tristan Fletcher, Support Vector Machines Explained (2008)*