Stat 6910, Section 003
Statistical Learning and Data Mining II
Fall 2018

Homework 4

Jared Hansen

A-number: A01439768

e-mail: jrdhansen@gmail.com

Due: 5:00 PM, Tuesday 11/20/18

1. **Convex Losses (10 pts).** We say that a loss is convex if for each fixed y, $L(y,t)$ is a convex function of t.

   (a) (5 pts) Show that the logistic loss is convex.

   - In the "Unconstrained Optimization" notes, we deduced the following property:
     f is convex $\iff \nabla^2 f(\boldsymbol{x})$ is positive semidefinite $\forall \boldsymbol{x} \in \mathbb{R}^d$.

   - Let's make use of this property by taking the gradient of $L(y,t)$ with respect to t and showing that the gradient is ≥ 0. If we can do this, then we will have shown that the logistic loss function is convex.

   - In our notes we're given that the logistic loss is $L(y,t) = \log(1+e^{-yt})$. Per the prompt, we'll regard y as a constant for calculations.

   - $$\begin{aligned}\nabla_t\Big(L(y,t)\Big) &= \Big(\frac{1}{1+e^{-yt}}\Big)\Big(\frac{-ye^{-yt}}{1}\Big)\\ &= \Big(\frac{-ye^{-yt}}{1+e^{-yt}}\Big)\end{aligned}$$

   - $$\begin{aligned}\nabla^2{}_t\Big(L(y,t)\Big) &= \nabla_t\Big(\nabla_t L(y,t)\Big)\\ &= \nabla_t\left(\frac{-ye^{-yt}}{1+e^{-yt}}\right)\\ &= -\frac{(1+e^{-yt})(-y^2e^{-yt}) - (ye^{-yt})(-ye^{-yt})}{(1+e^{-yt})^2}\\ &= -\frac{-y^2e^{-yt} + \big(-y^2(e^{-yt})(e^{-yt})\big) + \big(y^2(e^{-yt})(e^{-yt})\big)}{\big(1+e^{-yt}\big)^2}\\ \nabla^2{}_t\Big(L(y,t)\Big) &= \frac{y^2e^{-yt}}{\big(1+e^{-yt}\big)^2}\end{aligned}$$

   - Since $y^2 \geq 0$ and $e^{-yt} \geq 0 \implies \big(y^2\big)\big(e^{-yt}\big) \geq 0$, so the numerator of $\nabla^2{}_t \geq 0$.

   - Also, since $e^{-yt} \geq 0 \implies \big(1+e^{-yt}\big) \geq 1 \implies \big(1+e^{-yt}\big)^2 \geq 1^2$.
     So the denominator of $\nabla^2{}_t \geq 1$.

   - Therefore, we can colloquially express $\nabla^2{}_t$ as $\frac{\text{(“}\geq 0\text{”)}}{\text{(“}\geq 1\text{”)}} \geq 0 \implies \nabla^2{}_t \geq 0$
     $\nabla^2{}_t \geq 0 \implies \nabla^2{}_t$ is positive semidefinite $\implies L(y,t)$ is convex for fixed y, proving the desired result of showing the logistic loss to be convex.

(b) (5 pts) Show that if L is a convex loss, then

$$\hat{R}(\boldsymbol{w}, b) = \frac{1}{n} \sum_{i=1}^{n} L(y_i, \boldsymbol{w}^T \boldsymbol{x}_i + b)$$

is a convex function of $\boldsymbol{\theta} = \begin{bmatrix} \boldsymbol{w} \\ b \end{bmatrix}$.

- The first thing we need to do is show that $L(y_i, \boldsymbol{w}^T \boldsymbol{x}_i + b)$ is convex for fixed $\boldsymbol{x}_i$ and fixed y_i. Since $\boldsymbol{x}_i$ and y_i are fixed, the only thing that can vary is $\boldsymbol{w}$ and b, which is defined as $\boldsymbol{\theta}$. Rewriting in terms of $\boldsymbol{\theta}$, we can say $g(\boldsymbol{\theta}) = L(y_i, \boldsymbol{\theta}^T \boldsymbol{x}_i)$. So now our aim is to show that $g(\boldsymbol{\theta})$ is convex.
- By definition of convexity we're trying to show that, for some $\boldsymbol{\theta}_1$ and $\boldsymbol{\theta}_2$ and $\lambda \in [0, 1]$, $g\Big(\lambda\boldsymbol{\theta}_1 + (1-\lambda)\boldsymbol{\theta}_2\Big) \leq \lambda g(\boldsymbol{\theta}_1) + (1-\lambda) g(\boldsymbol{\theta}_2)$ holds.
- $g\Big(\lambda\boldsymbol{\theta}_1 + (1-\lambda)\boldsymbol{\theta}_2\Big) = L\Big(y_i, \Big[\lambda\boldsymbol{\theta}_1 + (1-\lambda)\boldsymbol{\theta}_2\Big]^T \boldsymbol{x}_i\Big) = L\Big(y_i, \big(\lambda {\boldsymbol{\theta}_1}^T \boldsymbol{x}_i + (1-\lambda) {\boldsymbol{\theta}_2}^T \boldsymbol{x}_i\big)\Big)$
- Now, for ease of notation, let $t_1 = ({\boldsymbol{\theta}_1}^T \boldsymbol{x}_i)$ and $t_2 = ({\boldsymbol{\theta}_2}^T \boldsymbol{x}_i)$ so that we have $L\Big(y_i, \lambda t_1 + (1-\lambda) t_2\Big)$. Since the prompt says "...if L is a convex loss," we can say that $L\Big(y_i, \lambda t_1 + (1-\lambda) t_2\Big) \leq \lambda\Big[L(y_i, t_1)\Big] + (1-\lambda)\Big[L(y_i, t_2)\Big]$ by the definition of convexity.
- Substituting things back in for t_1 and t_2 we have that $L\Big(y_i, \lambda\big({\boldsymbol{\theta}_1}^T \boldsymbol{x_i}\big) + (1-\lambda)\big({\boldsymbol{\theta}_2}^T \boldsymbol{x_i}\big)\Big) \leq \lambda\Big[L\big(y_i, {\boldsymbol{\theta}_1}^T \boldsymbol{x_i}\big)\Big] + (1-\lambda)\Big[L\big(y_i, {\boldsymbol{\theta}_2}^T \boldsymbol{x_i}\big)\Big]$. Based on our defintion of $g(\boldsymbol{\theta})$, this is: $g\Big(\lambda\boldsymbol{\theta}_1 + (1-\lambda)\boldsymbol{\theta}_2\Big) \leq \lambda g(\boldsymbol{\theta}_1) + (1-\lambda) g(\boldsymbol{\theta}_2)$, so by definition we have shown $g(\boldsymbol{\theta})$ to be convex.
- Since $\Big[\hat{R}(\boldsymbol{w}, b) = \frac{1}{n}\sum_{i=1}^{n} L(y_i, \boldsymbol{w}^T \boldsymbol{x}_i + b)\Big] = \Big[\hat{R}(\boldsymbol{\theta}) = \frac{1}{n}\sum_{i=1}^{n} g(\boldsymbol{\theta})\Big]$ and we've shown that $g(\boldsymbol{\theta})$ is convex, we know that $\hat{R}(\boldsymbol{w}, b)$ is convex since it is the sum of convex functions $g(\boldsymbol{\theta})$. Therefore we have proven the desired result.

2. **Conceptual questions (8 pts).**

(a) (4 pts) **Linear classifiers** Discuss the differences between LDA, Logistic Regression, separating hyperplanes, and the Optimal Soft-Margin Hyperplane. In what situations would you use each of the classifiers? *Hint:* You may find the discussion in sections 4.5.2 and 12.2 in the ESL book or sections 9.1, 9.2, and 9.5 in the ISL book helpful.

All of the classifiers in the prompt are potential choices when classes are linearly separable since all of these classifiers create some kind of linear decision boundary. If the data can be perfectly linearly separated, there are an infinite number of hyperplanes that can separate the classes. Therefore, some kind of cross-validation must be performed to determine which hyperplane is the best separator. In contrast, there is precisely one Optimal Soft-Margin Hyperplane (the "Optimal" thing gives it away).
One of the main differences between LDA and logistic regression is parameter estimation. LDA estimates parameters c_0 and c_1 using estimated mean and variance from a normal distribution, while logistic regression estimates β_0 and β_1 using maximum likelihood. As such, LDA is the better choice when (we can assume) our data are modeled well by a normal distribution with a common covariance matrix for each class. Logsitic regression is a better alternative when this assumption cannot be made (which is often the case). Finally, we ought to use an SVM (a variant of the OSM Hyperplane) when classes are well-separated, and logistic regression when classes have some overlap.

(b) (4 pts) Describe how you would apply the SVM to the multiclass case.

There are two mains ways I came across for using the SVM on a multi-class classification problem. The One-VS-One method creates an SVM for each pair of classes. For N classes, this means that there will be $\binom{N}{2}$ SVMs created. For example, if there are 3 classes (call them A, B, and C) the algorithm will create an SVM for A-vs-B, B-vs-C, and A-vs-C. After creating these $\binom{N}{2}$ SVMs, they will vote to generate the final predicted class of each instance.
Another way of applying the SVM on a multi-class problem is the One-VS-All method. Rather than fitting $\binom{N}{2}$ SVMs, we only have to fit N of them. Each SVM classifies an observation as being a member of the single class, or as being outside of that class. These N SVMs are similarly combined into a single prediction for each instance via voting. Since $N \leq \binom{N}{2}$ for $N \in \{3, 4, \ldots\}$ and fitting and tuning SVMs can be a lot of work, the One-VS-All method is often a more computationally realistic approach.

3. **Kernels (22 pts).**

(a) (4 pts) To what feature map Φ does the kernel

$$k(\boldsymbol{u}, \boldsymbol{v}) = (\langle u, v \rangle + 1)^3$$

correspond? Assume the inputs have an arbitrary dimension d and the inner product is the dot product.

- Here is my approach: solve this for $d = 2$ such that $u, v \in \mathbb{R}^2$, where $\langle u, v \rangle = \left(u^{(1)}v^{(1)} + u^{(2)}v^{(2)}\right)$, then extrapolate from inputs $\in \mathbb{R}^2$ to inputs $\in \mathbb{R}^d$.
- $\left(\langle u, v \rangle + 1\right)^3 = \left(u^{(1)}v^{(1)} + u^{(2)}v^{(2)} + 1\right)^3$. This will be messy so let $a = u^{(1)}v^{(1)}$, $b = u^{(2)}v^{(2)}$, and $c = 1$.
- $(a+b+c)^3 = (a+b+c)^2(a+b+c) = a^3+b^3+c^3+3a^2b+3a^2c+3ab^2+3b^2c+3ac^2+3bc^2+6abc$
- Here, $\Phi(u) = \Big[(u^{(1)})^3, (u^{(2)})^3, 1, \sqrt{3}(u^{(1)})^2(u^{(1)}), \sqrt{3}(u^{(1)})^2, \sqrt{3}(u^{(1)})(u^{(1)})^2, \sqrt{3}(u^{(2)})^2, \sqrt{3}(u^{(1)}), \sqrt{3}(u^{(2)}), \sqrt{6}(u^{(1)})(u^{(2)})\Big]$ and $\Phi(v)$ is the same, except with "v" in place of "u" $\forall u$.
- NOTE: When attempting to extrapolate this to higher dimensions, I found additional terms not present for $\mathbb{R}^2$ when experimenting with $d = 3$ and $d = 4$. I successfully detected and characterized this pattern for $d = d$ through a lot of scratch work. For my sake (I can't spend 2+ hours LaTex-ing scratch work) and your sake (a lot of extra reading), I have omitted this scratch work. The feature map can be found on the next page (it takes up enough space I put it on its own page).

- NOTE: I have intentionally formatted this a bit oddly, but it's done with the intent of grouping like elements of the feature map. See explanation below feature map definition for details. Extrapolating our $d = 2$ feature map to higher, arbitrary dimension d we'll have $\Phi(u) =$

$$\Big[(u^{(1)})^3, (u^{(2)})^3, ..., (u^{(d)})^3,$$

$$1,$$

$$\sqrt{3}(u^{(1)})^2(u^{(2)}), \sqrt{3}(u^{(1)})^2(u^{(3)}), ..., \sqrt{3}(u^{(1)})^2(u^{(d)}), ..., \sqrt{3}(u^{(d)})^2(u^{(1)}), ..., \sqrt{3}(u^{(d)})^2(u^{(d-1)}),$$

$$\sqrt{3}(u^{(1)})^2, \sqrt{3}(u^{(2)})^2, ..., \sqrt{3}(u^{(d)})^2,$$

$$\sqrt{3}(u^{(1)}), \sqrt{3}(u^{(2)}), ..., \sqrt{3}(u^{(d)}),$$

$$\sqrt{6}\big(u^{(1)}u^{(2)}\big), \sqrt{6}\big(u^{(1)}u^{(3)}\big), ..., \sqrt{6}\big(u^{(1)}u^{(d)}\big), \sqrt{6}\big(u^{(2)}u^{(3)}\big), \sqrt{6}\big(u^{(2)}u^{(4)}\big)..., \sqrt{6}\big(u^{(d-1)}u^{(d)}\big),$$

$$\sqrt{6}\big(u^{(1)}u^{(2)} \cdots u^{(d-1)}\big), \sqrt{6}\big(u^{(1)}u^{(2)} \cdots u^{(d-2)}u^{(d)}\big), ..., \sqrt{6}\big(u^{(1)}u^{(3)} \cdots u^{(d-1)}u^{(d)}\big)\Big]$$

  i. The first line of $\Phi(u)$ has d elements
  ii. The second line of $\Phi(u)$ has 1 element
  iii. The third line of $\Phi(u)$ has $d(d-1)$ elements
  iv. The fourth line of $\Phi(u)$ has d elements
  v. The fifth line of $\Phi(u)$ has d elements
  vi. The sixth line of $\Phi(u)$ has $\frac{d(d-1)}{2}$ elements
  vii. The seventh line of $\Phi(u)$ has $\frac{(d-2)(d-1)(d)}{6}$ elements

  There end up being a total of $\left[\frac{(d+1)(d+2)(d+3)}{6}\right]$ elements in the feature map $\Phi(u)$.
  For example, for $d = 2$ there are 10 elements, which matches what we got for the work at the beginning of the problem. I checked this for higher-dimensional d and it holds. (Again, this comes from the ugly scratch work that isn't worth LaTex-ing).

  The feature map $\Phi(v)$ looks the same as $\Phi(u)$ except that all "u's" are replaced with "v's".

  Thus the desired feature map has been defined for $k(u, v) = \langle \Phi(u), \Phi(v) \rangle$ and we're done.

(b) (18 pts) Let k_1, k_2 be symmetric, positive-definite kernels over $\mathbb{R}^D \times \mathbb{R}^D$, let $a \in \mathbb{R}^+$ be a positive real number, let $f : \mathbb{R}^D \to \mathbb{R}$ be a real-valued function, and let $p : \mathbb{R} \to \mathbb{R}$ be a polynomial with positive coefficients. For each of the functions k below, state whether it is necessarily a positive-definite kernel. If you think it is, prove it. If you think it is not, give a counterexample.

NOTE: for the remainder of this problem, I'll be using abbreviations to cut down on writing:

- PD = positive definite
- PSD = positive semi-definite
- SPD = symmetric positive definite
- IP = inner product

i. $k(x, z) = k_1(x, z) + k_2(x, z)$

- In this case the function k **IS** necessarily a PD kernel.
- Let's prove that k's corresponding kernel matrix is PSD $\forall \boldsymbol{y_i} \in \mathbb{R}^d$.
- From our notes, we say that a kernel is PD if its kernel matrix is PSD $\forall \boldsymbol{y}$.
- Since we already have $\boldsymbol{x}$'s in the problem, I'm going to use y for the next part.
  Let $K_1 = \begin{bmatrix} k_1(y_1, y_1) & k_1(y_1, y_2) & \cdots & k_1(y_1, y_n) \\ k_1(y_2, y_1) & k_1(y_2, y_2) & \cdots & k_1(y_2, y_n) \\ \vdots & \vdots & \ddots & \vdots \\ k_1(y_n, y_1) & k_1(y_n, y_2) & \cdots & k_1(y_n, y_n) \end{bmatrix}$
  Also, define the matrix K_2 similarly, replacing all k_1's with k_2's. These matrices – K_1 and K_2 – are the respective kernel matrices for $k_1(x, z)$ and $k_2(x, z)$ respectively.
- We know that $\Big(\boldsymbol{y}^T\big[K_1 + K_2\big]\boldsymbol{y}\Big) = \Big(\boldsymbol{y}^T\big[K_1\big]\boldsymbol{y}\Big) + \Big(\boldsymbol{y}^T\big[K_2\big]\boldsymbol{y}\Big)$. Since it is given that k_1 and k_2 are SPD kernels, we know that both kernels are PD kernels, and have kernel matrices (K_1 and K_2) that are PSD by definition.
- Therefore, we know that $\Big(\boldsymbol{y}^T\big[K_1\big]\boldsymbol{y} \geq 0\Big)$ and $\Big(\boldsymbol{y}^T\big[K_2\big]\boldsymbol{y} \geq 0\Big)$
  $\implies \boldsymbol{y}^T\big[K_1 + K_2\big]\boldsymbol{y} \geq 0 \implies [K_1 + K_2]$ is a PSD kernel matrix $\implies k_1(x, z) + k_2(x, z)$ is a PD kernel $\implies$ the function $k(x, z)$ is necessarily a PD kernel.

ii. $k(x,z) = k_1(x,z) - k_2(x,z)$

- In this case the function k **IS NOT** necessarily a PD kernel.
- Counterexample
  Define K_1 and K_2 (kernel matrices for $k_1(x,z)$ and $k_2(x,z)$) as we did in part (i.) of this problem. However, let's specify that $K_2 = 2 \cdot K_1$. Also, let's specify that K_1 and K_2 are both PD matrices.
  By definition, K_1 and K_2 must be PSD matrices, but we are looking at a specific case where K_1 and K_2 are both PD. Since {PD matrices} $\subset$ {PSD matrices} this means K_1 and K_2 are still (also) PSD matrices.
- Now, using the kernel matrix for $k(x,z)$, $[K_1 - K_2] = [K_1 - 2 \cdot K_1]$ examining $\Big(\boldsymbol{y}^T\big[K_1 - 2 \cdot K_1\big]\boldsymbol{y}\Big) = \Big(\boldsymbol{y}^T\big[K_1\big]\boldsymbol{y}\Big) - 2\Big(\boldsymbol{y}^T\big[K_1\big]\boldsymbol{y}\Big)$.
  Since we chose K_1 to be PD, we know that $\Big(\boldsymbol{y}^T\big[K_1\big]\boldsymbol{y}\Big) > 0 \; \forall \boldsymbol{y} \in \mathbb{R}^d$ and $\boldsymbol{y} \neq \mathbf{0}$.
  Let $c = \Big(\boldsymbol{y}^T\big[K_1\big]\boldsymbol{y}\Big) \implies \big(c > 0\big)$. Knowing that $c > 0$ we can reduce the value of the kernel matrix for $k(x,z)$ from $\Big(\boldsymbol{y}^T\big[K_1\big]\boldsymbol{y}\Big) - 2\Big(\boldsymbol{y}^T\big[K_1\big]\boldsymbol{y}\Big)$ down to $\Big(c - 2c\Big) = -c$.
  Since $(c > 0) \implies (-c < 0) \implies$ the kernel matrix $[K_1 - K_2] = [K_1 - 2 \cdot K_1]$ is not PSD $\implies$ k is not a PD kernel in this case.

iii. $k(x,z) = ak_1(x,z)$

- In this case the function k **IS** necessarily a PD kernel.
- For this function let's show that its corresponding kernel matrix is PSD.
- Let $K_1 = \begin{bmatrix} k_1(y_1,y_1) & k_1(y_1,y_2) & \cdots & k_1(y_1,y_n) \\ k_1(y_2,y_1) & k_1(y_2,y_2) & \cdots & k_1(y_2,y_n) \\ \vdots & \vdots & \ddots & \vdots \\ k_1(y_n,y_1) & k_1(y_n,y_2) & \cdots & k_1(y_n,y_n) \end{bmatrix}$
  By definition, since we know that $k_1(x,z)$ is an SPD kernel, we know that its kernel matrix K_1 is PSD for all $\boldsymbol{y}_i \in \mathbb{R}^d$, mathematically notated as $\Big(\boldsymbol{y}^T\big[K_1\big]\boldsymbol{y}\Big) \geq 0$.
- We know that multiplying $a \cdot K_1$ will give a PSD matrix since $\Big(\boldsymbol{y}^T\big[a \cdot K_1\big]\boldsymbol{y}\Big) = a \cdot \Big(\boldsymbol{y}^T\big[K_1\big]\boldsymbol{y}\Big)$ and we know that $\Big(\boldsymbol{y}^T\big[K_1\big]\boldsymbol{y}\Big) \geq 0$. Since a is defined to be a positive real number (in prompt), we know that $(a) \cdot (\text{“} \geq 0\text{”}) \geq 0$, and therefore $a \cdot \Big(\boldsymbol{y}^T\big[K_1\big]\boldsymbol{y}\Big) \geq 0 \implies \Big(\boldsymbol{y}^T\big[a \cdot K_1\big]\boldsymbol{y}\Big) \geq 0 \implies [a \cdot K_1]$ is a PSD matrix.
- Since $[a \cdot K_1]$ is the kernel matrix for $ak_1(x,z)$ and it is a PSD matrix, we know that $k(x,z) = ak_1(x,z)$ is necessarily a PD kernel.

iv. $k(x,z) = k_1(x,z)k_2(x,z)$

- In this case the function k **IS** necessarily a PD kernel.
- We will show this by proving $k(x,z)$ is an IP kernel.
- From the prompt, it's given that k_1 and k_2 are SPD kernels. Using the theorem (given in notes) that [k is an SPD kernel] $\iff$ [k is an IP kernel] we know that k_1 and k_2 are IP kernels. As such, k_1 and k_2 can each be expressed as an IP of feature maps.
- Using these facts, define $k_1(x,z) = [\phi^{(1)}(x)]^T[\phi^{(1)}(z)] = \sum_{j=1}^{d}\left(\phi_j^{(1)}(x)\right)\left(\phi_j^{(1)}(z)\right)$
  $= \left[\left(\phi_1^{(1)}(x)\right)\left(\phi_1^{(1)}(z)\right) + \left(\phi_2^{(1)}(x)\right)\left(\phi_2^{(1)}(z)\right) + \ldots + \left(\phi_d^{(1)}(x)\right)\left(\phi_d^{(1)}(z)\right)\right]$
- Similarly, $k_2(x,z) = \left[\left(\phi_1^{(2)}(x)\right)\left(\phi_1^{(2)}(z)\right)+\left(\phi_2^{(2)}(x)\right)\left(\phi_2^{(2)}(z)\right)+\ldots+\left(\phi_d^{(2)}(x)\right)\left(\phi_d^{(2)}(z)\right)\right]$
- Now, using these definitions, let's see what $k_1(x,z)k_2(x,z)$ is:
  $k_1(x,z)k_2(x,z) =$
  $\left[\left(\left(\phi_1^{(1)}(x)\right)\left(\phi_1^{(2)}(x)\right)\right)\left(\left(\phi_1^{(1)}(z)\right)\left(\phi_1^{(2)}(z)\right)\right)\right]+\left[\left(\left(\phi_1^{(1)}(x)\right)\left(\phi_2^{(2)}(x)\right)\right)\left(\left(\phi_1^{(1)}(z)\right)\left(\phi_2^{(2)}(z)\right)\right)\right]$
  $+\ldots+\left[\left(\left(\phi_1^{(1)}(x)\right)\left(\phi_d^{(2)}(x)\right)\right)\left(\left(\phi_d^{(1)}(z)\right)\left(\phi_d^{(2)}(z)\right)\right)\right]+$
  $\left[\left(\left(\phi_2^{(1)}(x)\right)\left(\phi_1^{(2)}(x)\right)\right)\left(\left(\phi_2^{(1)}(z)\right)\left(\phi_1^{(2)}(z)\right)\right)\right]+\left[\left(\left(\phi_2^{(1)}(x)\right)\left(\phi_2^{(2)}(x)\right)\right)\left(\left(\phi_2^{(1)}(z)\right)\left(\phi_2^{(2)}(z)\right)\right)\right]$
  $+\ldots+\left[\left(\left(\phi_2^{(1)}(x)\right)\left(\phi_d^{(2)}(x)\right)\right)\left(\left(\phi_2^{(1)}(z)\right)\left(\phi_d^{(2)}(z)\right)\right)\right]$
  $+\ldots+\left[\left(\left(\phi_d^{(1)}(x)\right)\left(\phi_1^{(2)}(x)\right)\right)\left(\left(\phi_d^{(1)}(z)\right)\left(\phi_1^{(2)}(z)\right)\right)\right]+\left[\left(\left(\phi_d^{(1)}(x)\right)\left(\phi_2^{(2)}(x)\right)\right)\left(\left(\phi_d^{(1)}(z)\right)\left(\phi_2^{(2)}(z)\right)\right)\right]$
  $+\ldots+\left[\left(\left(\phi_d^{(1)}(x)\right)\left(\phi_d^{(2)}(x)\right)\right)\left(\left(\phi_d^{(1)}(z)\right)\left(\phi_d^{(2)}(z)\right)\right)\right]$
- Thus, to show that $k_1(x,z)k_2(x,z)$ can be expressed as the inner product of two feature maps (and is hence an IP kernel), we can use the above expression of k_1k_2 to show $k_1(x,z)k_2(x,z) = [\psi(x)]^T[\psi(z)]$.
- From the above expression, $\psi(x) =$
  $\Big[\left(\left(\phi_1^{(1)}(x)\right)\left(\phi_1^{(2)}(x)\right)\right),\left(\left(\phi_1^{(1)}(x)\right)\left(\phi_2^{(2)}(x)\right)\right),\ldots,\left(\left(\phi_1^{(1)}(x)\right)\left(\phi_d^{(2)}(x)\right)\right),$
  $\left(\left(\phi_2^{(1)}(x)\right)\left(\phi_1^{(2)}(x)\right)\right),\left(\left(\phi_2^{(1)}(x)\right)\left(\phi_2^{(2)}(x)\right)\right),\ldots,\left(\left(\phi_2^{(1)}(x)\right)\left(\phi_d^{(2)}(x)\right)\right),$
  $\ldots\ldots,\left(\left(\phi_d^{(1)}(x)\right)\left(\phi_1^{(2)}(x)\right)\right),\left(\left(\phi_d^{(1)}(x)\right)\left(\phi_2^{(2)}(x)\right)\right),\ldots,\left(\left(\phi_d^{(1)}(x)\right)\left(\phi_d^{(2)}(x)\right)\right)\Big]$
  NOTE: the feature map is purposely broken up onto different lines to more clearly illustrate the patterns associated with the indices of the ϕ functions.
- Similarly, $\psi(z)$ will look identical to $\psi(x)$, except all x's will be replaced with z's.
- Since we are able to express $k(x,z) = k_1(x,z)k_2(x,z) = \langle\psi(x),\psi(z)\rangle\ \forall x,z \in \mathbb{R}^d$ we know that $k(x,z)$ is an IP kernel. Using the theorem
  (k is an IP kernel) $\iff$ (k is an SPD kernel), k must be an SPD kernel. Since all SPD kernels are also PD kernels, k must necessarily be a PD kernel.

v. $k(x,z) = f(x)f(z)$

- In this case the function k **IS** necessarily a PD kernel.
- To begin, here is a definition from course notes:
  We say $k : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$ is an IP kernel is $\exists$ an IP space V and a feature map $\Phi : \mathbb{R}^d \to V$ such that $k(x,z) = \langle \Phi(x), \Phi(z) \rangle \ \forall x, z \in \mathbb{R}^d$.
- Let's define feature map $\Phi(x) = f(x)$. Since $f(x) : \mathbb{R}^d \to \mathbb{R}$, from the definition of the feature map above ($\Phi : \mathbb{R}^d \to V$) we can deduce that $\mathbb{R}$ is our IP space V. Similarly, define $\Phi(z) = f(z)$. This $f(z)$ also maps $\mathbb{R}^d \to V$, where V is $\mathbb{R}$.
- Now that we've defined $\Phi(x)$ and $\Phi(z)$, we can use the definition to go from $k(x,z) = \langle \Phi(x), \Phi(z) \rangle$ to $k(x,z) = f(x)f(z)$.
- Since $f(x) : \mathbb{R}^d \to \mathbb{R}$ and $f(z) : \mathbb{R}^d \to \mathbb{R}$, we know that the result of $f(x)f(z)$ is $a \cdot b = c$, where $[f(x) = a] \in \mathbb{R}$, $[f(z) = b] \in \mathbb{R}$, and $c \in \mathbb{R}$ since the space $\mathbb{R}$ is closed under multiplication. Therefore, we've found a feature map $\Phi : \mathbb{R}^d \to V$ (where V is $\mathbb{R}$) such that $k(x,z) = f(x)f(z) \ \forall x, z \in \mathbb{R}^d$.
- Hence, we've shown that $k(x,z)$ is an IP kernel. Invoking the theorem
  (k is an IP kernel) $\iff$ (k is an SPD kernel), we conclude that k is also an SPD kernel. Since all SPD kernels are also PD kernels, we conclude that k must necessarily be a PD kernel.

vi. $k(x,z) = p\left(k_1(x,z)\right)$

- In this case the function k **IS** necessarily a PD kernel.
- Definition of a polynomial with degree p is $p(t) = \sum_{i=0}^{p} a_i t^i$. For this problem we're given that $a_i \in \mathbb{R}^+ \ \forall i$.
- We can express $k(x,z)$ as $p(k_1(x,z))$, where $p(k_1(x,z)) = \sum_{i=0}^{p} a_i \big(k_1(x,z)\big)^i$
  where $\sum_{i=0}^{p} a_i \big(k_1(x,z)\big)^i = a_0 \big(k_1(x,z)\big)^0 + a_1 \big(k_1(x,z)\big)^1 + a_2 \big(k_1(x,z)\big)^2 + \ldots + a_p \big(k_1(x,z)\big)^p$.
- We know, by definition in the prompt, that $k_1(x,z)$ is an SPD kernel. From **part (iv)** of this problem, we've shown the property that the product of two SPD kernels is a PD kernel. We can extend this property to each of the terms $\big(k_1(x,z)\big)^i$ in our sum. So we know that $\big(k_1(x,z)\big)^i$ is a PD kernel for $i \in \{1, 2, \ldots, p\}$.
- Now that we know this, we can apply the property shown in **part (iii)** of this problem to know that $a_i \big(k_1(x,z)\big)^i$ is a PD kernel, since $a_i \in \mathbb{R}^+$ and we've just shown that $\big(k_1(x,z)\big)^i$ is a PD kernel for $i \in \{1, 2, \ldots, p\}$.
- Now that we know each term $a_i \big(k_1(x,z)\big)^i$ is a PD kernel $\forall i$, we know that each term in the sum $p(k_1(x,z)) = \sum_{i=0}^{p} a_i \big(k_1(x,z)\big)^i = a_0 \big(k_1(x,z)\big)^0 + a_1 \big(k_1(x,z)\big)^1 + a_2 \big(k_1(x,z)\big)^2 + \ldots + a_p \big(k_1(x,z)\big)^p$ is a PD kernel. From **part (i)** of this problem, we've shown that the sum of PD kernels is a PD kernel. Applying this to our sum $\sum_{i=0}^{p} a_i \big(k_1(x,z)\big)^i$, we know that this is necessarily a PD kernel.

4. **Alternative OSM hyperplane (10 pts).** An alternative way to extend the max-margin hyperplane to nonseparable data is to solve the following quadratic program (another name for an optimization problem with a quadratic objective function):

$$\begin{array}{ll} \min_{\boldsymbol{w},b,\boldsymbol{\xi}} & \frac{1}{2}\left\|\boldsymbol{w}\right\|^2 + \frac{C}{n}\sum_{i=1}^{n}\xi_i^2 \\ s.t. & y_i(\boldsymbol{w}^T\mathbf{x}_i + b) \geq 1 - \xi_i \ \forall i \\ & \xi_i \geq 0 \ \forall i \end{array}$$

The only difference with respect to the OSM hyperplane is that we are now squaring the slack variables. This assigns a stronger penalty to data points that violate the margin.

(a) (4 pts) Which loss is associated with the above quadratic program? In order words, show that learning a hyperplane by the above optimization problem is equivalent to ERM with a certain loss.

- The first step is to scale by a constant. We have seen other loss functions that are scaled by some constant, so this is a "legal move." Leaving the constraints as they are originally, scale this quadratic program by λ where $\lambda = \frac{1}{C}$. In the first term, use λ as is, and multiply by the definition $\frac{1}{C}$ in the second term to get:
  $\min_{\boldsymbol{w},b,\boldsymbol{\xi}} \frac{\lambda}{2}\left\|\boldsymbol{w}\right\|^2 + \frac{1}{n}\sum_{i=1}^{n}\xi_i^2$

- Now, consolidate constraints by putting things in terms of ξ.
  $y_i(\boldsymbol{w}^T\mathbf{x}_i + b) \geq 1 - \xi_i \implies \xi_i \geq 1 - y_i(\boldsymbol{w}^T\mathbf{x}_i + b)$
  Next, combine the two constraints: $\xi_i \geq \max(0, 1 - y_i(\boldsymbol{w}^T\mathbf{x}_i + b))$. Since we know $\xi_i \geq 0$ we know that the statement ${\xi_i}^2 \geq [max(0, 1 - y_i(\boldsymbol{w}^T\mathbf{x}_i + b))]^2$ also holds.

- It is clear that the solution to the original optimization problem must satisfy
  ${\xi_i}^2 = [max(0, 1 - y_i(\boldsymbol{w}^T\mathbf{x}_i + b))]^2$. Therefore, substituting in our value of ${\xi_i}^2$, the optimal solution will also solve $\min_{\boldsymbol{w},b,\boldsymbol{\xi}} \frac{\lambda}{2}\left\|\boldsymbol{w}\right\|^2 + \frac{1}{n}\sum_{i=1}^{n}[max(0, 1 - y_i(\boldsymbol{w}^T\mathbf{x}_i + b))]^2$.
  Referring back to past notes, we can see that this is optimization problem is now equivalent to regularized ERM with a squared hinge loss.

(b) (4 pts) Argue that the second set of constraints can be dropped without changing the solution.

Our combined constraint from part (a) was ${\xi_i}^2 \geq [max(0, 1 - y_i(\boldsymbol{w}^T\mathbf{x}_i + b))]^2$.
Since (any real-valued term)$^2 \geq 0$, we know that $\left(1 - y_i(\boldsymbol{w}^T\mathbf{x}_i + b)\right)^2 \geq 0$ since $1 - y_i(\boldsymbol{w}^T\mathbf{x}_i + b)$ is a real-valued term. Since this will always hold, taking the maximum of $(0, \left(1 - y_i(\boldsymbol{w}^T\mathbf{x}_i + b)\right)^2)$ is redundant, and we can drop the "max" and "0" parts, to just be left with the consolidated, equivalent constraint of ${\xi_i}^2 \geq \left(1 - y_i(\boldsymbol{w}^T\mathbf{x}_i + b)\right)^2$

(c) (2 pts) Identify an advantage and a disadvantage of this loss compared to the hinge loss.

The advantage and disadvantage of using this squared hinge loss are one and the same. Whether or not it's an advantage depends on the context of the problem for which it's being employed. In essence, this loss will make the classifier more sensitive to extreme training values: highly misclassified values will have a larger impact on the decision boundary (by shifting it). If the aim of the classifier is to make sure to avoid misclassification, using this loss is an advantage over the regular hinge loss. If not, this loss will likely over-penalize relative to the hinge loss, and won't give as good of results.

5. **Subgradient methods for the optimal soft margin hyperplane (25 pts).**
In this problem you will implement the subgradient and stochastic subgradient methods for minimizing the convex but nondifferentiable function

$$J(\boldsymbol{w}, b) = \frac{1}{n}\sum_{i=1}^{n} L\left(y_i, \boldsymbol{w}^T\mathbf{x}_i + b\right) + \frac{\lambda}{2}\|\boldsymbol{w}\|^2$$

where $L(y,t) = \max 0, 1 - yt$ is the hinge loss. As we saw in class, this corresponds to the optimal soft margin hyperplane.

(a) (5 pts) Determine $J_i(, b)$ such that

$$J(, b) = \sum_{i=1}^{n} J_i(, b).$$

Determine a subgradient $\mathbf{u}_i$ of each J_i with respect to the variable $\boldsymbol{\theta} = [b\,^T]^T$. A subgradient of J is then $\sum_i \mathbf{u}_i$.
Note: Recall that if $f() = g(h())$ where $g : \mathbb{R} \to \mathbb{R}$ and $h : \mathbb{R}^d \to \mathbb{R}$, and both g and h are differentiable, then the chain rule gives

$$\nabla f() = \nabla h() \cdot g'(h()).$$

If g is convex and h is differentiable, then the same formula gives a subgradient of f at where $g'(h())$ is replaced by a subgradient of g at $h()$.

- It is given that $L(y_i, \boldsymbol{w}^T\boldsymbol{x}_i + b) = max\Big(0, (1 - y_i(\boldsymbol{w}^T\boldsymbol{x}_i + b))\Big)$. Substituting this into $J(\boldsymbol{w}, b)$ we get that $J(\boldsymbol{w}, b) = \frac{1}{n}\sum_{i=1}^{n}\left[max\Big(0, (1 - y_i(\boldsymbol{w}^T\boldsymbol{x}_i + b))\Big)\right] + \frac{\lambda}{2}\|\boldsymbol{w}\|^2$.
- Next, let's bring the $\frac{1}{n}$ into the summed terms (both) to give
  $J(\boldsymbol{w}, b) = \sum_{i=1}^{n}\left[\frac{1}{n}max\Big(0, (1 - y_i(\boldsymbol{w}^T\boldsymbol{x}_i + b))\Big) + \frac{\lambda}{2n}\|\boldsymbol{w}\|^2\right]$
- If $J(\boldsymbol{w}, b) = \sum_{i=1}^{n} J_i(\boldsymbol{w}, b)$ as desired in the prompt, then we have shown that
  $J_i(\boldsymbol{w}, b) = \left[\frac{1}{n}max\Big(0, (1 - y_i(\boldsymbol{w}^T\boldsymbol{x}_i + b))\Big) + \frac{\lambda}{2n}\|\boldsymbol{w}\|^2\right]$
- Now, to find the formulation of the subgradient $\boldsymbol{u}_i\ \forall J_i$ we'll consider 3 cases:
  i. **CASE 1:** $1 - y_i(\boldsymbol{w}^T\boldsymbol{x}_i + b) > 0$
  $J_i = \frac{1}{n}\big(1 - y_i(\boldsymbol{w}^T\boldsymbol{x}_i + b\big) + \frac{\lambda}{2n}\|\boldsymbol{w}\|^2 \implies \nabla J_i = \frac{1}{n}\big(-y_i[1, \boldsymbol{x}_i]^T\big) + \frac{\lambda}{n}[0, \boldsymbol{w}]^T$

  ii. **CASE 2:** $1 - y_i(\boldsymbol{w}^T\boldsymbol{x}_i + b) < 0$
  $J_i = \frac{1}{n}\big(0\big) + \frac{\lambda}{2n}\|\boldsymbol{w}\|^2 = \frac{\lambda}{2n}\|\boldsymbol{w}\|^2 \implies \nabla J_i = \frac{1}{n}\big(0\big) + \frac{\lambda}{n}[0, \boldsymbol{w}]^T \implies \nabla J_i = \frac{\lambda}{n}[0, \boldsymbol{w}]^T$

  iii. **CASE 3:** $1 - y_i(\boldsymbol{w}^T\boldsymbol{x}_i + b) = 0$
  $J_i = \frac{1}{n}\big(max(0, 0)\big) + \frac{\lambda}{2n}\|\boldsymbol{w}\|^2$. In this case, the gradient ∇J_i DNE, so we'll select a subgradient instead (The gradients in cases 1 and 2 are subgradients for this case.)
- Putting these cases together, we get subgradient (sum $\forall i$ to get subgradient of J):
  $\boldsymbol{u}_i =$

$$\begin{cases} \frac{1}{n}\big(-y_i[1, \boldsymbol{x}_i]^T\big) + \frac{\lambda}{n}[0, \boldsymbol{w}]^T & for\big[1 - y_i(\boldsymbol{w}^T\boldsymbol{x}_i + b\big] \geq 0 \\ \frac{\lambda}{n}[0, \boldsymbol{w}]^T & for\big[1 - y_i(\boldsymbol{w}^T\boldsymbol{x}_i + b\big] < 0 \end{cases}$$

(b) (6 pts) Download the file `nuclear.mat` from Canvas. The variables x and y contain training data for a binary classification problem. The variables correspond to the total energy and tail energy of waveforms produced by a nuclear particle detector. The classes correspond to neutrons and gamma rays. Neutrons have a slightly larger tail energy for the same total energy relative to gamma rays, which allows the two particle types to be distinguished. This is a somewhat large data set ($n = 20,000$), and subgradient methods are appropriate given their scalability.
Implement the subgradient method for minimizing J and apply it to the nuclear data. Submit two figures: One showing the data and the learned line, the other showing J as a function of iteration number. Also report the estimated hyperplane parameters and the minimum achieved value of the objective function.
Comments:

- Use $\lambda = 0.001$. Since this is a linear problem in a low dimension, we don't need much regularization.
- Use a step-size of $\alpha_j = 100/j$, where j is the iteration number.
- To compute the subgradient of J, write a subroutine to find the subgradient of J_i and then sum those results.
- Since the objective will not be monotone decreasing, determining a good stopping rule can be tricky. For this problem, just look at the graph of the objective function and "eyeball it" to decide when the algorithm has converged.
- Debugging goes faster if you start out with just a subsample of the data.

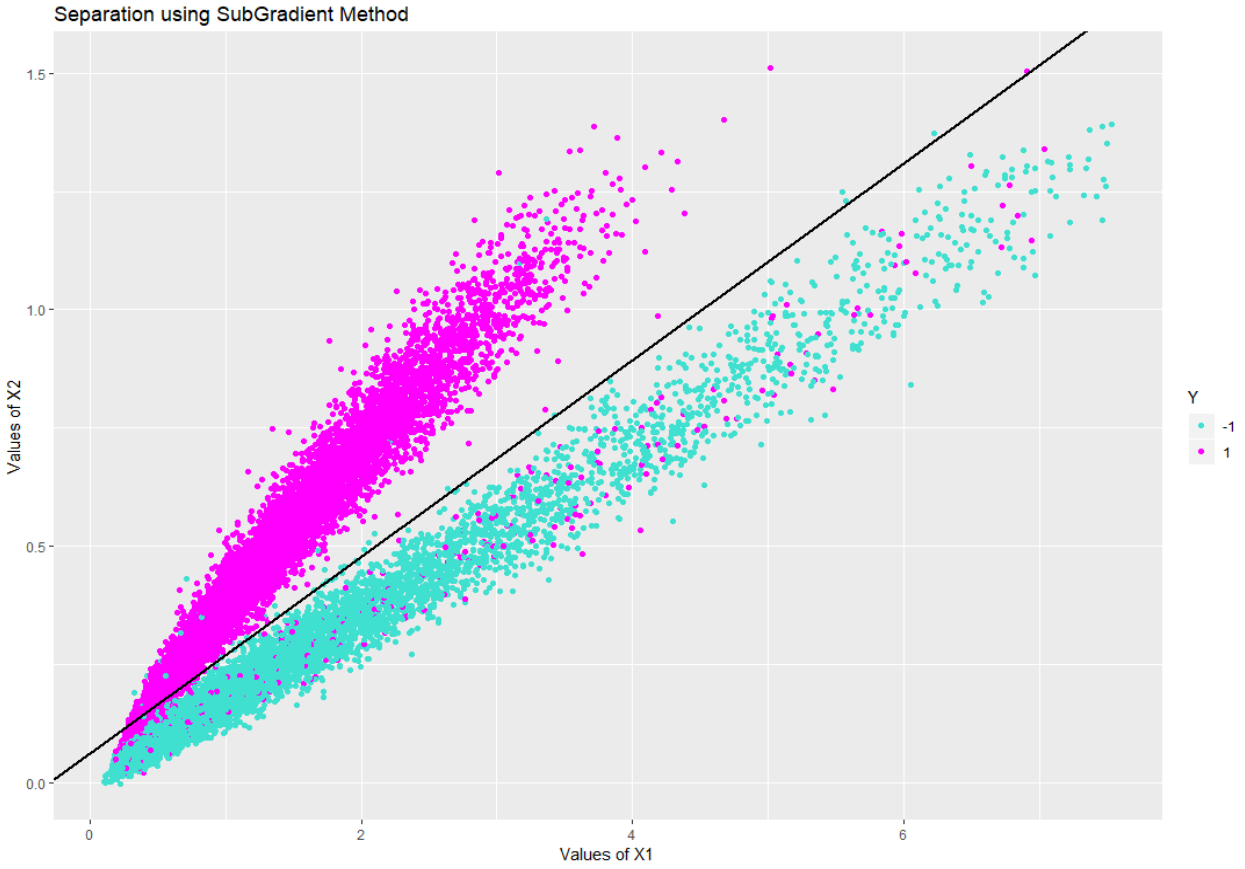

Figure 1: Learned Line using Subgradient Method

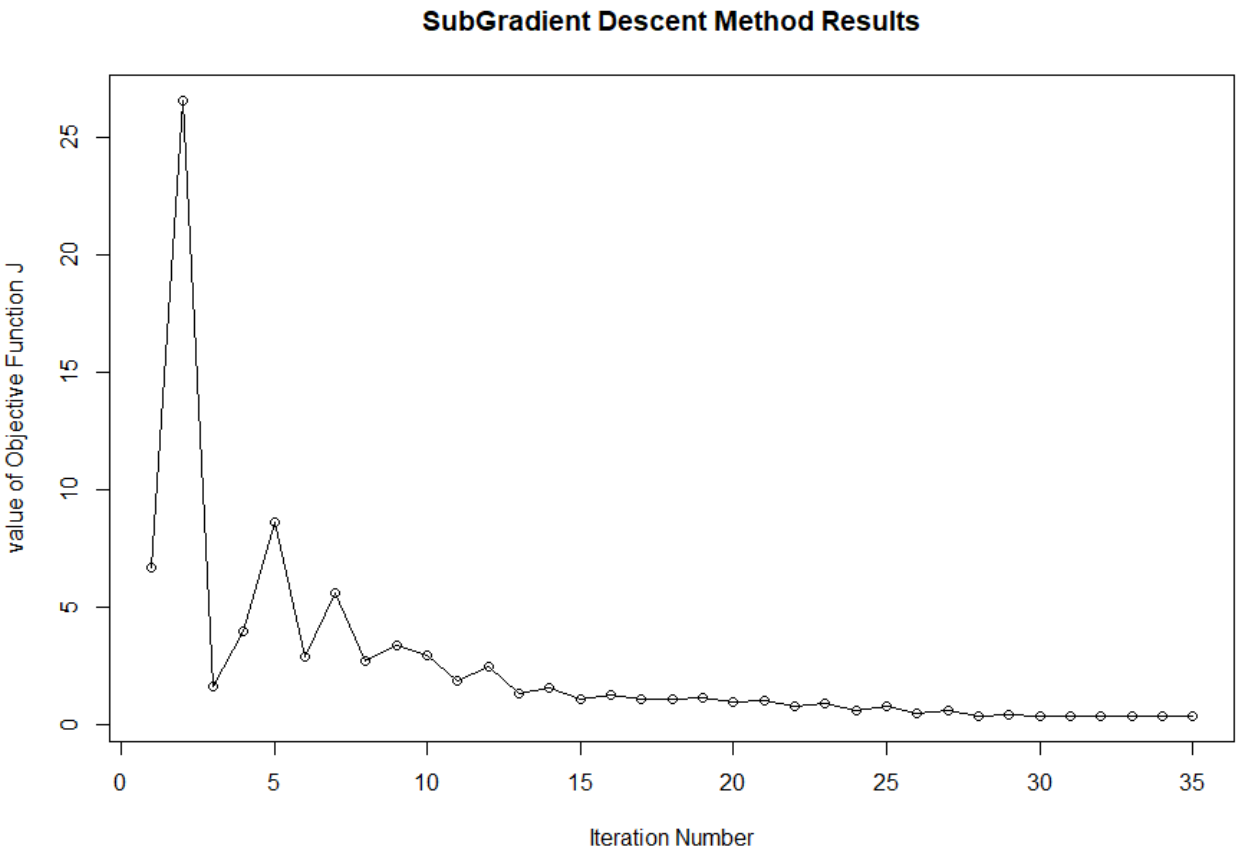

Figure 2: J (obj fctn) as a function of the iteration number

- Estimated hyperplane parameters: $\boldsymbol{\theta} = \begin{bmatrix} b = -1.15413 \\ w_1 = -3.891742 \\ w_2 = 18.68866 \end{bmatrix}$
- Minimum achieved value of the objective function: 0.3595984

(c) (6 pts) Now implement the stochastic subgradient method with a minibatch size of $m = 1$. Be sure to cycle through all data points before starting a new loop through the data. Report/hand in the same items as for part (b).
More comments:

- Use the same λ, stopping strategy, and α_j as in part (b). Here j indexes the number of times you have cycled (randomly) through the data; i.e., it is the number of epochs you have trained for.
- Your plot of J versus iteration number will have roughly n times as many points as in part (b) since you have n updates for every one update of the full subgradient method.

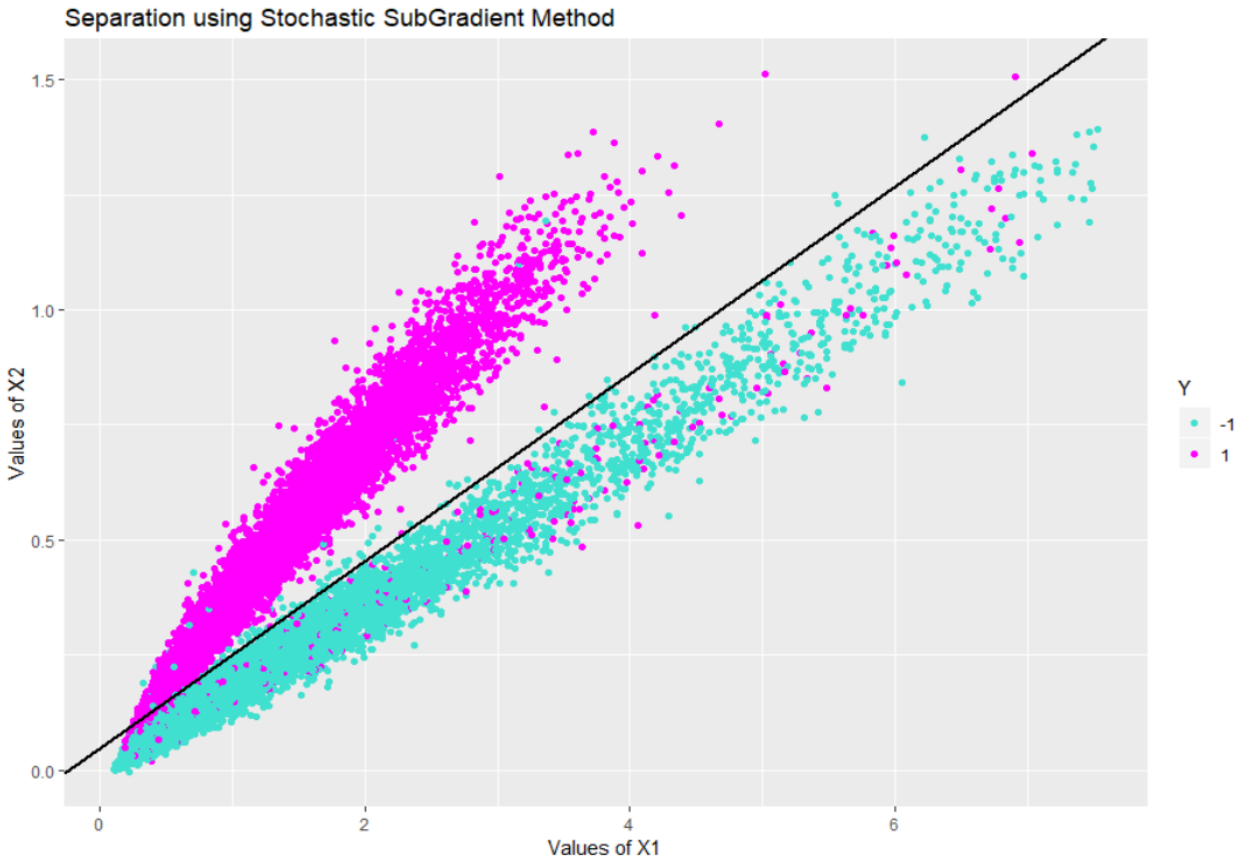

Figure 3: Learned Line using Stochastic Subgradient Method

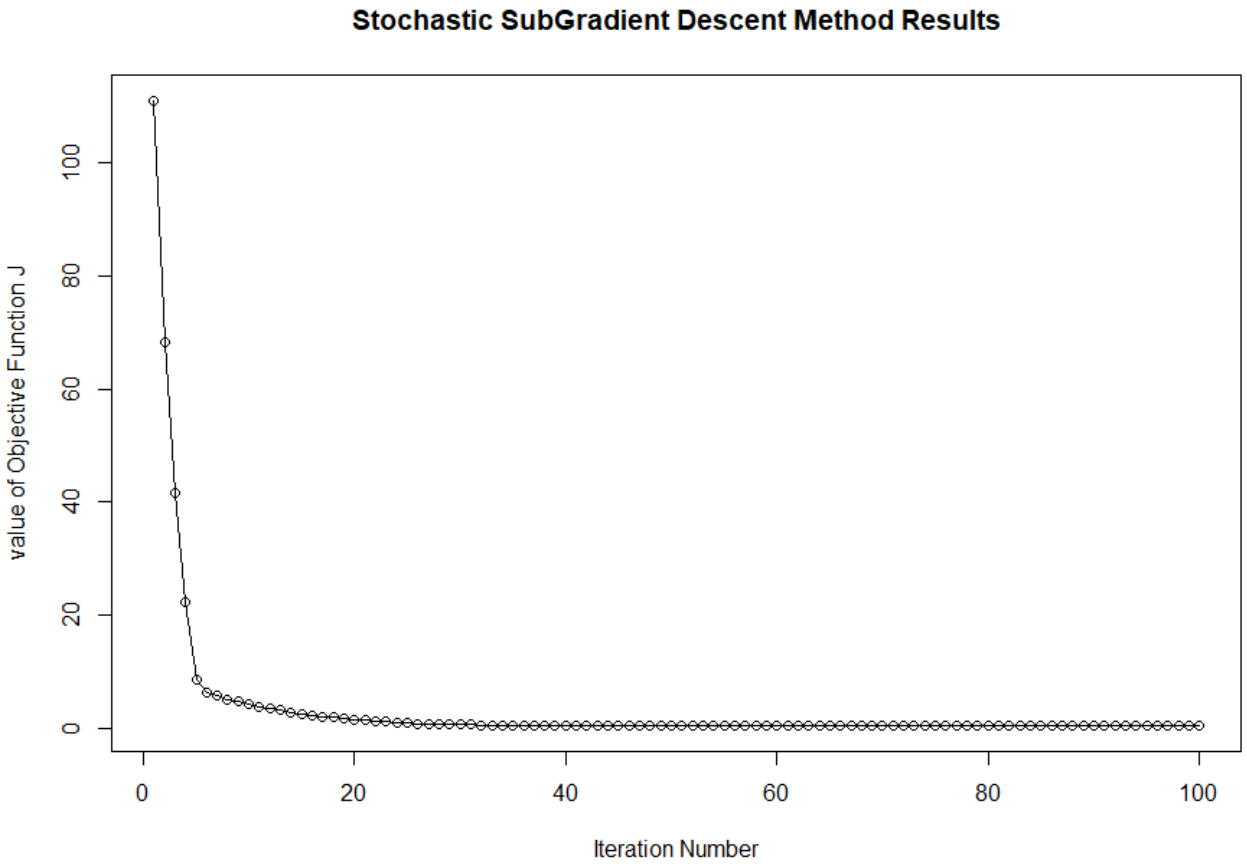

Figure 4: J (obj fctn) as a function of the iteration number

- Estimated hyperplane parameters: $\boldsymbol{\theta} = \begin{bmatrix} b = -1.186326 \\ w_1 = -5.106034 \\ w_2 = 25.07871 \end{bmatrix}$
- Minimum achieved value of the objective function: 0.4904483

(d) (3 pts) Comment on the empirical rate of convergence of the stochastic subgradient method relative to the subgradient method. Explain your finding.

We can see that the rate of convergence is faster for the subgradient method (in terms of number of iterations necessary to converge). This makes sense, since the subgradient method is utilizing the entire dataset when performing computations, and will achieve an optimum more quickly in a smaller number of iterations.
However, I noticed when running the code that the stochastic subgradient method iterations ran much more quickly than the subgradient iterations. While this method may have taken more iterations to converge, it seemed to be similar from a "time-to-compute" standpoint.

(e) (5 pts) Submit your code.

See below

```r
## STAT LEARNING 2
## HOMEWORK 4, Problem 5

# Read in necessary libraries
library(ggplot2)
library(magrittr)
library(R.matlab)
library(geometry)
library(dplyr)


# Set the working directory
setwd("C:/Users/jrdha/OneDrive/Desktop/USU_Fa2018/Moon__SLDM2/hw4/Problem5")


# This function kicks off the recursive subGradient function.
# It takes as arguments: max_Iter (max number of iterations), predictors
# (dataframe of input features), response (dataframe containing values of
# responses), lambda (used for calculation of regularization term).
# The function returns optimized theta vector (weights, b).
initSubGradient <- function(max_Iter, predictors, response, lambda){
  b = 1
  ws = rep(0, ncol(predictors) - 1)
  inittheta <- c(b, ws)

  # Dataframe to store info from iterations
  iteration_Info <- data.frame(numItr = double(0), objFun = double(0),
                        w1 = double(0), w2 = double(0), b = double(0))

  # call subGradient function
  theta <- subGradient(theta = inittheta, num_Iter = 1, max_Iter = max_Iter,
                   xs = predictors, y = response,
                   lambda = lambda, iteration_Info = iteration_Info)
  return(theta)
}


# This function (recursively) performs the calculations to determine optimal value
# of theta or calls itself again.
# the initSubGradient function.
# Takes as arguments: theta (vector of weights and b), num_Iter and max_Iter (are
# self-explanatory), xs (dataframe containing input data), y (dataframe containing
# response data), lambda (regularization parameter), iteration_Info (dataframe
# that stores parameter estimates and value of objective function).
subGradient <- function(theta, num_Iter, max_Iter, xs, y, lambda, iteration_Info){

  # Alpha is the step size, n is the number of instances in preds
  alpha <- 100/num_Iter
  n <- nrow(preds)
```

```r
  # Either retunr optimized theta and iteration info, or keep going
  if(num_Iter > max_Iter){
    return(list("theta" = theta, "iteration_Info" = iteration_Info))
  } else {
    print(paste("Iteration: ", num_Iter))

    u <- double(length(theta))
    for(i in 1:nrow(xs)){
      # calculate gradient
      grad <- calcGrad(b = theta[1], w = theta[2:length(theta)],
                       yi = as.numeric(y$Y[i]), xi = as.numeric(xs[i,]),
                       n = n, lambda = lambda)
      # Update u
      u <- u + grad
    }

    # Having calculated gradient, take a step in the opposite direction
    theta.1 <- theta - (alpha * u)

    # calculate value of objective function
    objFctn_val <- calc_objFctn(preds = xs, resp = y, theta = theta.1, lambda = lam
bda)

    # This row will be added to the bottom of the iteration_Info dataframe
    hnew <- data.frame(numItr = num_Iter, objFun = objFctn_val, b = theta.1[1],
                       w1 = theta.1[2], w2 = theta.1[3])

    iteration_Info <- rbind(iteration_Info, hnew)
    num_Iter <- num_Iter + 1

    return(subGradient(theta = theta.1, num_Iter = num_Iter, max_Iter = max_Iter,
                   xs = xs, y = y, lambda = lambda, iteration_Info = iteration_Info
))
  }
}


# This function kicks of the recursive stochSubGradient function.
# It takes as arguments: max_Iter, predictors, response, and lambda (we've seen
# all of these before, so I won't re-explain what they are).
# It returns the optimal value of the theta vector.
initStochSubGradient <- function(max_Iter, predictors, response, lambda){

  # Initial guesses for theta (b and the weights)
  b = 1
  ws = rep(0, ncol(predictors) - 1)
  inittheta <- c(b, ws)

  # Initialize the iteration_Info data frame
  iteration_Info <- data.frame(numItr = double(0), objFun = double(0),
                         w1 = double(0), w2 = double(0), b = double(0))
```

```r
  # Recursion of stochSubGradient() begins
  theta <- stochSubGradient(theta = inittheta, num_Iter = 1, max_Iter = max_Iter,
                        xs = predictors, y = response,
                        lambda = lambda, iteration_Info = iteration_Info)
  return(theta)
}


# This recursive function calls the subGradient function.
# Takes as arguments all the things we know and love: theta vector, num_Iter,
# max_Iter, xs, y, lambda, and iteration_Info dataframe.
# This function either returns the optimal value of theta, or calls itself again
# depending on how many iterations have taken place.
stochSubGradient <- function(theta, num_Iter, max_Iter, xs, y, lambda, iteration_In
fo){

  # Set the step size, how many predictor instances there are, and recommended
  # minibatch size
  alpha <- 100/num_Iter
  n <- nrow(preds)
  m <- 1

  # The use of the sample() function below will be how we randomly sample
  if(num_Iter > max_Iter){
    return(list("theta" = theta, "iteration_Info" = iteration_Info))
  } else {
    print(paste("Iteration Number: ", num_Iter))
    index <- 1:n
    rand.index <- sample(index, n)
    for(i in rand.index){
        grad <- calcGrad(b = theta[1], w = theta[2:length(theta)],
                         yi = as.numeric(y$Y[i]), xi = as.numeric(xs[i,]),
                         n = 1, lambda = lambda)
        theta <- theta - (alpha * grad) # Move in opposite direction of subGrad
                                        # to update theta.
    }
    theta.1 <- theta

    # Calculate the (current) value of the objective function.
    objFctn_val <- calc_objFctn(preds = xs, resp = y, theta = theta.1, lambda = lam
bda)

    # This row will be added to the bottom of the iteration_Info dataframe
    hnew <- data.frame(numItr = num_Iter, objFun = objFctn_val, b = theta.1[1],
                       w1 = theta.1[2], w2 = theta.1[3])

    iteration_Info <- rbind(iteration_Info, hnew)
    num_Iter <- num_Iter + 1
    return(subGradient(theta = theta.1, num_Iter = num_Iter, max_Iter = max_Iter, x
s = xs, y = y, lambda = lambda, iteration_Info = iteration_Info))
  }
}
```

```r
# This function calculates the gradient.
# Takes as arguments: w (weights vector), b (part of theta, intercept term),
# yi (response), xi (vector containing predictor instance), n (num of instances),
# lambda (same parameter for calculating regularization penalty term).
# Returns the subGradient.
calcGrad <- function(w, b, yi, xi, n, lambda){

  wvec <- c(0, w)
  term <- 1 - (yi * (dot(wvec, xi) + b))
  if(term >= 0){
    gradJi <- (1/n)*(-yi*xi) + (lambda/n)*wvec
  } else {
    gradJi <- (lambda/n)*wvec
  }
  sg <- (gradJi)
  return(sg)
}


# Calculates the value of the objective function for given inputs.
# Takes as arguments: preds (predictor data), resp (response value), theta (same
# vector of weights and b), lambad (same regularization penalty parameter).
calc_objFctn <- function(preds, resp, theta, lambda){

  n <- nrow(preds)
  b <- theta[1]
  w <- theta[2:length(theta)]
  wvec <- c(0, w)
  val_Summation <- 0
  for(i in 1:n){
    yi = as.numeric(resp$Y[i])
    xi = as.numeric(preds[i,])
    m1 <- 0
    m2 <- 1 - yi*(dot(wvec, xi) + b)
    term <- max(m1, m2)
    val_Summation <- val_Summation + term
  }
  val_objFctn <- 1/n * (val_Summation) + (lambda/2)*dot(w,w)
  return(val_objFctn)
}

# This function plots the values of the objective functions against the iteration
# number for that given value of J.
# Takes as arguments the iteration_Info dataframe for plotting, and a title.
# Output is the plot described above.
plot_objFctn_vals <- function(iteration_Info, title){
  plot(iteration_Info$numItr, iteration_Info$objFun, xlab = "Iteration Number",
       ylab = "value of Objective Function J", type = 'o',
       main = title)
}
```

```r
#============================================================================
#== Implementing the defined functions to solve the given problem ==========
#============================================================================

givenData <- R.matlab::readMat("nuclear.mat") %>% lapply(t) %>% lapply(as_tibble)
colnames(givenData[[1]]) <- sprintf("X_%s",seq(1:ncol(givenData[[1]])))
colnames(givenData[[2]]) <- c("Y")
givenData <- bind_cols(givenData) %>% select(Y, everything())

# givenData <- slice(givenData, 1:300)
preds <- select(givenData, X_1, X_2)
X0 <- rep(1, nrow(givenData))
preds <- cbind(X0, preds)
resp <- select(givenData, Y)
resp$Y <- as.numeric(resp$Y)


#=== Results of implementing subGradient method ================================

# Seems to flatten out after about 35 iterations, chose a fairly small value of lam
bda.
subGrad_results <- initSubGradient(max_Iter = 35, predictors = preds, response = re
sp, lambda = 0.001)



plot_objFctn_vals(subGrad_results$iteration_Info, title = "SubGradient Descent Meth
od Results")

b <- subGrad_results$theta[1]
w1 <- subGrad_results$theta[2]
w2 <- subGrad_results$theta[3]

slopeVal <- w1/-w2
interceptVal <- b/-w2

# Gets mad if not converted to the factor data type
givenData$Y <- as.factor(givenData$Y)

ggplot(data = givenData, aes(x = X_1, y = X_2, color = Y)) +
  geom_point() +
  scale_color_manual(values=c("-1" = "turquoise", "1" = "magenta")) +
  geom_abline(slope =  slopeVal, intercept = interceptVal, lwd = 1) +
  xlab("Values of X1") +
  ylab("Values of X2") +
  ggtitle("Separation using SubGradient Method")
```

```r
#=== Results of implementing stochSubGradient method ==========================
# This one took a lot more iterations to converge as compared to subGradient method
.
stoch_subGrad_results <- initStochSubGradient(max_Iter = 100, predictors = preds,
                          response = resp, lambda = 0.001)

plot_objFctn_vals(stoch_subGrad_results$iteration_Info, title = "Stochastic SubGrad
ient Descent Method Results")

b <- stoch_subGrad_results$theta[1]
w1 <- stoch_subGrad_results$theta[2]
w2 <- stoch_subGrad_results$theta[3]

slopeVal <- w1/-w2
interceptVal <- b/-w2

plot(preds$X_1, preds$X_2)
abline(a = interceptVal, b = slopeVal)

# Again, have to change this data type to factor or it'll get mad
givenData$Y <- as.factor(givenData$Y)

ggplot(data = givenData, aes(x = X_1, y = X_2, color = Y)) +
  geom_point() +
  scale_color_manual(values=c("-1" = "turquoise", "1" = "magenta")) +
  geom_abline(slope = slopeVal, intercept = interceptVal, lwd = 1) +
  xlab("Values of X1") +
  ylab("Values of X2") +
  ggtitle("Separation using Stochastic SubGradient Method")
```

6. **Classification (15 pts).**

(a) (2 pts) Download a dataset for classification from the UCI Machine Learning Repository (you can Google it). Report the number of classes and the number of features in your chosen dataset.

Based on the recommendations at the bottom of the prompt, I ended up going with the Parkinsons data set, which can be found at $http://archive.ics.uci.edu/ml/datasets/Parkinsons$.
The data contains voice measurements (continuous, numeric) as well as the Parksinsons status of an individual. Required some minimal cleaning before application of classifiers.

- There are two classes for the response: 1 (has Parkinsons), 0 (is healthy).
- There are 22 features. I anticipated that including one of these features would result in significant data leakage, so I removed it. Additionally, it was a categorical variable; per your hint that SVMs work best on features that are (continuous) numeric, it made even more sense to remove this variable.

(b) (3 pts) Apply logistic regression to this dataset. Calculate the test error based on k-fold cross-validation (you may select k) and report the training and test error. You may use any built-in methods in your language of choice.

To begin, I randomly split the data into a training and test set (80% of the original data and 20% of the original data respectively). Additionally, I converted the response variable to be a factor (rather than 0, 1 numeric) so that the methods would recognize this as a classification problem and not a regression problem.
For logistic regression, I used the caret package in R, performing 10-fold crossvalidation to fit the model. It gave the following results:

- Training error: 0.1533929 (10-fold CV)
- Test error: 0.1794872 (predicted using the 10-fold crossvalidated model)

(c) (8 pts) Apply the SVM to this dataset using 2 different kernels: linear and Gaussian. Describe the tuning parameters in each case and set these parameters by cross-validation. Report your final test error and training error after cross-validation.

Linear Kernel SVM

Here was my methodology for selecting hyperparameters:

- For this kernel, the only parameter that needs tuning is C, which, for the *tune.svm()* function in R is the *cost* parameter.
- I started by "casting the net wide" and looping through SVMs built using 10-fold cross-validation with a linear kernel with cost values of $(1.d)(10^i)$ where $d \in \{0, 1, 2, \ldots, 9\}$ and $i \in \{-3, -2, \ldots, 3\}$. I ended up finding that cost values of 1.6 and 1.9 gave equally low CV error of 0.14125.
- Having found 1.6 and 1.9 to be the best C values so far, I decided to try searching more closely around them to "tighten up the net" a bit. For this, I looped through the cost values in the set $\{1.00, 1.01, 1.02, \ldots, 2.08, 2.09, 2.10\}$. The best value here, 1.52, also returned the same 10-fold crossvalidated error rate of 0.14125 that $C = 1.6$ and 1.9 gave. My guess is that this is returned since 1.52, 1.6, 1.9 (and potentially others) all tie for error = 0.14125, but 1.52 "shows up first" in the sense that it's the smallest number.
- As such, I concluded that most C values in the range of about $1.5 - 2.0$ will give the same results. Thus, I trained several different models, with cost values of 1.52, 1.6, and 1.9. Then, I used these models to predict onto my test dataset. Each of them gave the same test error rate: 0.05128205.
- SUMMARY OF LINEAR KERNEL SVM RESULTS
  - Tuning parameter C: 1.52, 1.6, or 1.9 (give identical training and test errors)
  - 10-fold crossvalidated training error rate: 0.14125
  - Final test error rate: 0.05128205

Gaussian Kernel SVM

Here was my methodology for selecting hyperparameters:

- In addition to the parameter C, for the Gaussian kernel we have to select a parameter σ^2 that determines how much misclassification is penalized. For the pre-built functions I used in R, this parameter is called γ, where $\gamma = \frac{1}{\sigma^2}$. Therefore, by tuning γ we are effectively tuning σ^2.
- Initially, I started by grid-searching through values of cost $\in \{1 \times 10^{-3}, 1 \times 10^{-2}, \ldots, 1 \times 10^{7}\}$ and values of $\gamma \in \{1 \times 10^{-5}, 1 \times 10^{-4}, \ldots, 1 \times 10^{2}\}$. The best result (in terms of 10-fold CV error) was cost = 10 and $\gamma = 0.1$, giving 10-fold CV error of 0.03208333.
- Next, I experimented with leaving cost = 1.6 since that gave the best results for the linear kernel. This didn't pan out. The best result was 10-fold CV error of 0.06375 with $\gamma = 0.19$.
- Moving on, I grid-searched through values of cost $\in \{1.d * 10^n\}$ where $d \in \{0, 1, 2, \ldots, 9\}$ and $n \in \{-2, -1, 0, 1, 2\}$. The γ values I searched over were $\gamma \in \{0.10, 0.11, 0.12, \ldots, 0.19\}$.
  The best result was a CV error rate of 0.02583333 with cost = 10 and $\gamma = 0.11$.
- For the final "net tightening", I grid-searched through cost $\in \{5.0, 5.5, 6.0, \ldots, 14.0, 14.5, 15.0\}$ and $\gamma \in \{0.100, 0.105, 0.110, 0.115, \ldots, 0.185, 0.190\}$. It returned a 10-fold CV error rate of 0.02583333 for cost = 8 and $\gamma = 0.105$. (Again, I think that this is returned because these numbers are lower than 10 and 0.11, which give the same error rate).
- SUMMARY OF GAUSSIAN KERNEL SVM RESULTS
  - Tuning parameter C: 8 or 11 (give identical training and test errors)
  - Tuning parameter γ: 0.105 or 0.11 (give identical training and test errors)
  - 10-fold crossvalidated training error rate: 0.02583333
  - Final test error rate: 0.07692308

(d) (2 pts) Which of the three methods you applied (logistic regression, SVM with linear kernel, SVM with Gaussian kernel) worked best? Do your results make sense?

The method that gives the best results is the SVM with linear kernel. However, I'd be very willing to bet that I could get equally good results from the SVM with Gaussian kernel if I spent some more time finding tuning parameters C and γ. Since there were two of them, I think it made it trickier to narrow down the best combination, whereas the linear kernel only requires honing in on one individual hyperparameter. Also, since the dataset wasn't very big (195 instances in overall data, with 156 instances in training data and 39 instances in test data), there is only so much room for improvement in accuracy when attempting to classify 39 observations. The logistic regression was misclassifying $0.1794872 \cdot 39 = 7$ of the 39 test dataset. The linear kernel SVM only missed $0.05128205 \cdot 39 = 2$ instances of the 39, and the Gaussian kernel SVM missed $0.07692308 \cdot 39 = 3$ of the 39 test instances. The fine line between $2/39$ and $3/39$ is fairly slim, so I don't think we can conclusively say that the linear kernel SVM is better than the Gaussian kernel SVM.

If I understand the way that SVMs work, I think it makes complete sense that we see better results by using SVMs over logistic regression. Essentially, the instances of these data aren't nicely linearly separable in the untouched feature space; this is why logistic regression had a test error of 0.1794872. By employing the kernel trick on the data (via SVM), we transform the feature space into a different dimensional space where the data can be linearly separated by the SVM.

BELOW IS MY CODE FOR THIS PROBLEM

Comments:

- The linear kernel for SVM is equivalent to applying the optimal soft- margin hyperplane.
- You will probably want to choose a dataset that does not have any categorical features.
- Depending on which code you use, you may want to choose data with a binary classification problem.
- The SVM can take a while to train so you may want to consider sample size when choosing a dataset.

```r
## STAT LEARNING 2
## HOMEWORK 4, Problem 6

# Libraries needed
library(caret)
library(magrittr)
library("e1071")

# Set working directory
setwd("C:/Users/jrdha/OneDrive/Desktop/USU_Fa2018/Moon__SLDM2/hw4/Problem6")




#=============================================================================
#==== INITIAL STEPS ==========================================================
#=============================================================================

# Read in the data
parkData <- read.csv("parkinsonsData.csv", header = TRUE)

# Subset, get rid of first column
parkData <- parkData[ , -1]

# Split into training and test data sets. Going with the typical 80/20 split.
lengthTrain <- round(nrow(parkData) * 0.8)
lengthTest <- nrow(parkData) - lengthTrain

# Set seed
set.seed(1234)

# Make indices for training data subsetting
train_ind <- sample(seq_len(nrow(parkData)), size = lengthTrain)

# Split up the data
trainData <- parkData[train_ind, ]
testData <- parkData[-train_ind, ]

# Make the response a factor (rather than integer) so the algorithms perform
# classification instead of regression.
trainData$status <- as.factor(trainData$status)
```

```r
#=============================================================================
#==== LOGISTIC REGRESSION ====================================================
#=============================================================================

# This specifies that we'll be doing 10-fold crossvalidation
ctrl <- trainControl(method = "repeatedcv",
                     number = 10,
                     savePredictions = TRUE)

# Train the model
mod_fit <- train(status ~.,
                 data = trainData,
                 method= "glm",
                 family= binomial(),
                 trControl = ctrl,
                 tuneLength = 10)

# This gives the 10-fold crossvalidated training error
trainError_logReg <- 1 - mod_fit$results$Accuracy
trainError_logReg
```

```
## [1] 0.1533929
```

```r
# Generates the predictions for our testData
testData$pred = predict(mod_fit, newdata=testData)

# Generates a column that says whether or not a test observation was correctly
# classified (1==correct, 0==incorrect)
testData$correct <- with(testData, ifelse(status == pred, 1, 0))

# Generate the test error
totalCorrect <- sum(testData$correct)
testError_logReg <- 1 - (totalCorrect / lengthTest)
testError_logReg
```

```
## [1] 0.1794872
```

```r
# RETURNS A TEST ERROR RATE OF 0.1794872, WHICH IS PRECISELY 7
# MISCLASSIFICATIONS. WE WILL BEAT THIS WITH BOTH KERNEL-VERSIONS OF SVM.

# Remove the "correct" and "pred" columns so that we'll have the original data
# to work with for the SVM portion
testData <- subset(testData, select =  -c(correct, pred))
```

```r
#=============================================================================
#==== SVM ====================================================================
#=============================================================================



#========== TUNING LINEAR-KERNEL MODEL ======================================
#=============================================================================


# Tuning LINEAR KERNEL: Round 1 (used this to get a set of values that did well)
# Ran it by changing cost = 1.digit*10^(-3:3) where I set digit = 0,1,2,....,9
set.seed(2345)
svm_tune_linear <- tune.svm(status~.,
                            data = trainData,
                            kernel = "linear",
                            cost = 1.9*10^(-3:3))
print(svm_tune_linear)
```

```
##
## Parameter tuning of 'svm':
##
## - sampling method: 10-fold cross validation
##
## - best parameters:
##  cost
##   1.9
##
## - best performance: 0.14125
```

```r
#====== RESULTS ==============================================================
# Parameter tuning of 'svm':
#   - sampling method: 10-fold cross validation
# - best parameters:
#   cost
# 1.9
# - best performance: 0.14125



# Having found 1.9 to be the best value from above (by trying 1.0, 1.1, 1.2,...
# ...,1.9) and seeing that most of these values were around 1.5 - 2.0, I decided
# to just look at values between 1 and 2.
# Tuning LINEAR KERNEL: Round 2 (looking only at values between 1 and 2.1)
set.seed(2345)
svm_tune_linear <- tune.svm(status~.,
                            data = trainData,
                            kernel = "linear",
                            cost = seq(1,2.1,0.01))
print(svm_tune_linear)
```

```
##
## Parameter tuning of 'svm':
##
## - sampling method: 10-fold cross validation
##
## - best parameters:
##  cost
##  1.52
##
## - best performance: 0.14125
```

```
#========= RESULTS ============================================================
# IRONICALLY, THIS COMES UP WITH THE SAME CROSSVALIDATED ACCURACY AS FOR C=1.9,
# SO WE SHOULDN'T EXPECT BETTER RESULTS FOR OUR TEST ERROR BY USING C=1.52 OVER
# C=1.9.
# Parameter tuning of 'svm':
#   - sampling method: 10-fold cross validation
# - best parameters:
#   cost
# 1.52
# - best performance: 0.14125
```

```
# THIS FITS A FINAL SVM MODEL, USING THE CROSSVALIDATED TUNING PARAMETER 1.52
svm_model <- svm(status ~., trainData, kernel = "linear", cost = 1.52)
testData$pred <- predict(svm_model, testData)
testData$correct <- with(testData, ifelse(testData$status == pred, 1, 0))
# Generate the test error
# Total number of correct classifications
totalCorrect <- sum(testData$correct)
testError_SVM <- 1 - (totalCorrect / lengthTest)
testError_SVM
```

```
## [1] 0.05128205
```

```
# Remove the "correct" column for the SVM portion
testData <- subset(testData, select =  -c(correct, pred))
```

```
#========== RESULTS ============================================================
# We get the same error rate (0.05128205) as we did by using C=1.9. However,
# this makes sense, as we're already correctly classifying 37/39 values
# correctly with C=1.9, so there's very little room for improvement. Using
#C=1.52 gives just as good of results.
```

```r
# ADDITIONAL TUNING: WIDENING THE WINDOW, LOOKING BETWEEN 0.1-19
# ONCE AGAIN, WE GET A COST IN THE SWEET SPOT OF 1.5 - 1.9, RETURNING 1.6, AND
# THE IDENTICAL 0.14125 CROSS-VALIDATED ERROR RATE
set.seed(2345)
svm_tune_linear <- tune.svm(status~.,
                            data = trainData,
                            kernel = "linear",
                            cost = seq(0.1,19,0.1))
print(svm_tune_linear)
```

```
##
## Parameter tuning of 'svm':
##
## - sampling method: 10-fold cross validation
##
## - best parameters:
##  cost
##   1.6
##
## - best performance: 0.14125
```

```r
# Parameter tuning of 'svm':
#   - sampling method: 10-fold cross validation
# - best parameters:
#   cost
# 1.6
# - best performance: 0.14125



# THIS FITS A FINAL SVM MODEL, USING THE CROSSVALIDATED TUNING PARAMETER 1.6
svm_model <- svm(status ~., trainData, kernel = "linear", cost = 1.6)
testData$pred <- predict(svm_model, testData)
testData$correct <- with(testData, ifelse(testData$status == pred, 1, 0))
# Generate the test error
# Total number of correct classifications
totalCorrect <- sum(testData$correct)
testError_SVM <- 1 - (totalCorrect / lengthTest)
testError_SVM
```

```
## [1] 0.05128205
```

```r
# Remove the "correct" column for the SVM portion
testData <- subset(testData, select =  -c(correct, pred))

# COMMON TEST ERROR RATES:
# 0.05128205 for C values in the range of about 1.5 - 1.9
# 0.07692308 for other C values
# 0.1025641 for other C values
# THIS MAKES SENSE: our test data set has only 39 observations. Therefore, we
# can only achieve a certain level of granularity with our test error. The
# best-tuned values for C miss only 2/39 classifications.
```

```r
#========== TUNING GAUSSIAN-KERNEL MODEL ===================================
#============================================================================

# INITIAL PASS (leave base values at 10)
set.seed(2345)
svm_tune_gaussian <- tune.svm(status~., data = trainData,
                              cost = 10^(-3:7),
                              gamma = 10^(-5:2))
print(svm_tune_gaussian)
```

```
##
## Parameter tuning of 'svm':
##
## - sampling method: 10-fold cross validation
##
## - best parameters:
##  gamma cost
##    0.1   10
##
## - best performance: 0.03208333
```

```r
# Gives cost = 10 and gamma = 0.1 as best values, CV error of 0.03208333



# LEFT COST=1.6, SINCE THIS WAS THE BEST VALUE FROM LINEAR-KERNEL SVM
# THIS GIVES POORER RESULTS THAN WHEN WE LET THE COST BE BIGGER (and vary from
# c=1.6, see below)
# CHANGE THE BASE VALUES FOR GAMMA, TRYING 1.0, 1.1, 1.2,....,1.9
set.seed(2345)
svm_tune_gaussian <- tune.svm(status~., data = trainData,
                              cost = 1.6,
                              gamma = 1.9*10^(-3:2))
print(svm_tune_gaussian)
```

```
##
## Parameter tuning of 'svm':
##
## - sampling method: 10-fold cross validation
##
## - best parameters:
##  gamma cost
##   0.19  1.6
##
## - best performance: 0.06375
```

```r
# Gives cost = 1.6 and gamma = 0.19 as best values with CV error: 0.06375
```

```r
# MANUAL GRID SEARCH, TRYING ALL 100 POSSIBLE COMBINATIONS OF (1.digit,1.digit)
# for GAMMA and COST
# CHANGE THE BASE VALUES FOR GAMMA, TRYING 1.0, 1.1, 1.2,....,1.9
# CHANGE THE BASE VALUES FOR COST, TRYING 1.0, 1.1, 1.2,....,1.9
set.seed(2345)
svm_tune_gaussian <- tune.svm(status~., data = trainData,
                              cost = 1.0*10^(-2:2),
                              gamma = seq(0.1,0.19,0.01))
print(svm_tune_gaussian)
```

```
##
## Parameter tuning of 'svm':
##
## - sampling method: 10-fold cross validation
##
## - best parameters:
##  gamma cost
##   0.11   10
##
## - best performance: 0.02583333
```

```r
# for cost=1.0^power, was consistently picking cost=10, and gamma=0.1digit,
# so I set gamma to only be values of 0.1, 0.11, 0.12,....,0.19, and then tried
# different values for cost (changing base to be 1.0, 1.1, 1.2,...,1.9)
# For these restricted gamma, it was consistenly using gamma=0.16, and then cost
# of 12, 13, 14, and giving CV error of 0.03833333.
# However, the best combination turned out to be cost=10, gamma=0.11, giving a
# CV error of 0.02583333!



# TRYING TO NARROW DOWN EVEN FURTHER: IF WE KNOW THAT COST=10 AND GAMMA=0.11
# GIVES THE BEST CV ERROR, LET'S TRY OTHER VALUES CLOSE TO THESE ONES.
# I made the steps of the sequences more granular so we could try more values
# and potentially find better ones.
set.seed(2345)
svm_tune_gaussian <- tune.svm(status~., data = trainData,
                              cost = seq(5,15,0.5),
                              gamma = seq(0.1,0.19,0.005))
print(svm_tune_gaussian)
```

```
##
## Parameter tuning of 'svm':
##
## - sampling method: 10-fold cross validation
##
## - best parameters:
##  gamma cost
##  0.105    8
##
## - best performance: 0.02583333
```

```r
# HERE'S WHAT IS RETURNED. As we can see, we're still getting the same CV error
# RATE (0.02583333) WE WERE FOR GAMMA=0.11 AND COST=10.
# SO I THINK WE CAN SAFELY CONCLUDE THAT A COST IN 8-10 IS GOOD, AND TUNED
# GAMMA OF EITHER 0.105 OR 0.11 IS GOING TO BE OUR BEST BEST FOR THE
# GAUSSIAN-KERNEL SVM
# Parameter tuning of 'svm':
#   - sampling method: 10-fold cross validation
# - best parameters:
#   gamma cost
# 0.105    8
# - best performance: 0.02583333
```

```r
# THIS FITS A FINAL SVM MODEL, USING THE CROSSVALIDATED TUNING PARAMETERs
# of C = (range of 8,9,10), and gamma = 0.105 or 0.11
costVal = 8
gammaVal = 0.105
svm_model <- svm(status ~., trainData, kernel = "radial",
                 cost = costVal, gamma = gammaVal)
testData$pred <- predict(svm_model, testData)
testData$correct <- with(testData, ifelse(testData$status == pred, 1, 0))
# Generate the test error
# Total number of correct classifications
totalCorrect <- sum(testData$correct)
testError_SVM <- 1 - (totalCorrect / lengthTest)
testError_SVM
```

```
## [1] 0.07692308
```

```r
# Remove the "correct" column for the SVM portion
testData <- subset(testData, select =  -c(correct, pred))
#========= RESULTS ============================================================
# All different values here give error rate of 0.07692308, which is
# misclassifying precisely 3/39 of our test set, so 1/39 worse than for the
# linear kernel. Some of this is certainly due to there only being so much room
# to improve the error rate, since we can only pick off 1 or 2
# misclassifications in our test set.
```

7. How long did this assignment take you? (5 pts)

Date	Times			Day Total
Wed, 11/07	5:00 pm – 9:00 pm			**4.0** hrs
Thu, 11/08	10:30 am – 12:00 pm	4:30 pm – 8:30 pm		**5.5** hrs
Fri, 11/09	9:00 am – 10:00 am	5:00 pm – 7:30 pm		**3.5** hrs
Sat, 11/10	1:00 pm – 8:15 pm			**7.25** hrs
Sun, 11/11	10:30 am – 11:30 am	2:00 pm – 3:30 pm		**2.5** hrs
Mon, 11/12	9:15 am – 11:15 am	2:00 pm – 4:30 pm	6:45 pm – 8:45 pm	**6.5** hrs
Tue, 11/13	10:00 am – 11:00 am			**1.0** hrs
Wed, 11/14	4:00 pm – 6:15 pm			**2.25** hrs
Fri, 11/16	9:00 am – 11:00 am	5:15 pm – 9:30 pm		**6.25** hrs
				TOTAL: 38.75 HOURS

8. Type up homework solutions (5 pts)