CS 229, Public Course
Problem Set #1: Supervised Learning

---

1. **Newton's method for computing least squares**

   In this problem, we will prove that if we use Newton's method solve the least squares optimization problem, then we only need one iteration to converge to θ^*.

   (a) Find the Hessian of the cost function $J(\theta) = \frac{1}{2}\sum_{i=1}^{m}(\theta^T x^{(i)} - y^{(i)})^2$.

   (b) Show that the first iteration of Newton's method gives us $\theta^\star = (X^T X)^{-1} X^T \vec{y}$, the solution to our least squares problem.

2. **Locally-weighted logistic regression**

   In this problem you will implement a locally-weighted version of logistic regression, where we weight different training examples differently according to the query point. The locally-weighted logistic regression problem is to maximize

$$\ell(\theta) = -\frac{\lambda}{2}\theta^T\theta + \sum_{i=1}^{m} w^{(i)} \left[y^{(i)} \log h_\theta(x^{(i)}) + (1 - y^{(i)}) \log(1 - h_\theta(x^{(i)})) \right].$$

   The $-\frac{\lambda}{2}\theta^T\theta$ here is what is known as a regularization parameter, which will be discussed in a future lecture, but which we include here because it is needed for Newton's method to perform well on this task. For the entirety of this problem you can use the value $\lambda = 0.0001$.

   Using this definition, the gradient of $\ell(\theta)$ is given by

$$\nabla_\theta \ell(\theta) = X^T z - \lambda\theta$$

   where $z \in \mathbb{R}^m$ is defined by

$$z_i = w^{(i)}(y^{(i)} - h_\theta(x^{(i)}))$$

   and the Hessian is given by

$$H = X^T D X - \lambda I$$

   where $D \in \mathbb{R}^{m \times m}$ is a diagonal matrix with

$$D_{ii} = -w^{(i)} h_\theta(x^{(i)})(1 - h_\theta(x^{(i)}))$$

   For the sake of this problem you can just use the above formulas, but you should try to derive these results for yourself as well.

   Given a query point x, we choose compute the weights

$$w^{(i)} = \exp\left(-\frac{||x - x^{(i)}||^2}{2\tau^2} \right).$$

   Much like the locally weighted linear regression that was discussed in class, this weighting scheme gives more when the "nearby" points when predicting the class of a new example.

(a) Implement the Newton-Raphson algorithm for optimizing $\ell(\theta)$ for a new query point x, and use this to predict the class of x.

The `q2/` directory contains data and code for this problem. You should implement the `y = lwlr(X_train, y_train, x, tau)` function in the `lwlr.m` file. This function takes as input the training set (the `X_train` and `y_train` matrices, in the form described in the class notes), a new query point `x` and the weight bandwitdh `tau`. Given this input the function should 1) compute weights $w^{(i)}$ for each training example, using the formula above, 2) maximize $\ell(\theta)$ using Newton's method, and finally 3) output $y = 1\{h_\theta(x) > 0.5\}$ as the prediction.

We provide two additional functions that might help. The `[X_train, y_train] = load_data;` function will load the matrices from files in the `data/` folder. The function `plot_lwlr(X_train, y_train, tau, resolution)` will plot the resulting classifier (assuming you have properly implemented `lwlr.m`). This function evaluates the locally weighted logistic regression classifier over a large grid of points and plots the resulting prediction as blue (predicting $y = 0$) or red (predicting $y = 1$). Depending on how fast your `lwlr` function is, creating the plot might take some time, so we recommend debugging your code with `resolution = 50;` and later increase it to at least 200 to get a better idea of the decision boundary.

(b) Evaluate the system with a variety of different bandwidth parameters τ. In particular, try $\tau = 0.01, 0.050.1, 0.51.0, 5.0$. How does the classification boundary change when varying this parameter? Can you predict what the decision boundary of ordinary (unweighted) logistic regression would look like?

3. **Multivariate least squares**

So far in class, we have only considered cases where our target variable y is a scalar value. Suppose that instead of trying to predict a single output, we have a training set with multiple outputs for each example:

$$\{(x^{(i)}, y^{(i)}),\ i = 1, \ldots, m\},\ x^{(i)} \in \mathbb{R}^n,\ y^{(i)} \in \mathbb{R}^p.$$

Thus for each training example, $y^{(i)}$ is vector-valued, with p entries. We wish to use a linear model to predict the outputs, as in least squares, by specifying the parameter matrix Θ in

$$y = \Theta^T x,$$

where $\Theta \in \mathbb{R}^{n \times p}$.

(a) The cost function for this case is

$$J(\Theta) = \frac{1}{2} \sum_{i=1}^{m} \sum_{j=1}^{p} \left((\Theta^T x^{(i)})_j - y_j^{(i)} \right)^2.$$

Write $J(\Theta)$ in matrix-vector notation (i.e., without using any summations). [Hint: Start with the $m \times n$ design matrix

$$X = \begin{bmatrix} — & (x^{(1)})^T & — \\ — & (x^{(2)})^T & — \\ & \vdots & \\ — & (x^{(m)})^T & — \end{bmatrix}$$

and the $m \times p$ target matrix

$$Y = \begin{bmatrix} — & (y^{(1)})^T & — \\ — & (y^{(2)})^T & — \\ & \vdots & \\ — & (y^{(m)})^T & — \end{bmatrix}$$

and then work out how to express $J(\Theta)$ in terms of these matrices.]

(b) Find the closed form solution for Θ which minimizes $J(\Theta)$. This is the equivalent to the normal equations for the multivariate case.

(c) Suppose instead of considering the multivariate vectors $y^{(i)}$ all at once, we instead compute each variable $y_j^{(i)}$ separately for each $j = 1, \ldots, p$. In this case, we have a p individual linear models, of the form

$$y_j^{(i)} = \theta_j^T x^{(i)}, \ j = 1, \ldots, p.$$

(So here, each $\theta_j \in \mathbb{R}^n$). How do the parameters from these p independent least squares problems compare to the multivariate solution?

4. **Naive Bayes**

In this problem, we look at maximum likelihood parameter estimation using the naive Bayes assumption. Here, the input features $x_j, \ j = 1, \ldots, n$ to our model are discrete, binary-valued variables, so $x_j \in \{0, 1\}$. We call $x = [x_1 \ x_2 \ \cdots \ x_n]^T$ to be the input vector. For each training example, our output targets are a single binary-value $y \in \{0, 1\}$. Our model is then parameterized by $\phi_{j|y=0} = p(x_j = 1|y = 0)$, $\phi_{j|y=1} = p(x_j = 1|y = 1)$, and $\phi_y = p(y = 1)$. We model the joint distribution of (x, y) according to

$$\begin{aligned} p(y) &= (\phi_y)^y (1 - \phi_y)^{1-y} \\ p(x|y = 0) &= \prod_{j=1}^{n} p(x_j|y = 0) \\ &= \prod_{j=1}^{n} (\phi_{j|y=0})^{x_j} (1 - \phi_{j|y=0})^{1-x_j} \\ p(x|y = 1) &= \prod_{j=1}^{n} p(x_j|y = 1) \\ &= \prod_{j=1}^{n} (\phi_{j|y=1})^{x_j} (1 - \phi_{j|y=1})^{1-x_j} \end{aligned}$$

(a) Find the joint likelihood function $\ell(\varphi) = \log \prod_{i=1}^{m} p(x^{(i)}, y^{(i)}; \varphi)$ in terms of the model parameters given above. Here, φ represents the entire set of parameters $\{\phi_y, \ \phi_{j|y=0}, \ \phi_{j|y=1}, \ j = 1, \ldots, n\}$.

(b) Show that the parameters which maximize the likelihood function are the same as

those given in the lecture notes; i.e., that

$$\begin{aligned}
\phi_{j|y=0} &= \frac{\sum_{i=1}^{m} 1\{x_j^{(i)} = 1 \wedge y^{(i)} = 0\}}{\sum_{i=1}^{m} 1\{y^{(i)} = 0\}} \\
\phi_{j|y=1} &= \frac{\sum_{i=1}^{m} 1\{x_j^{(i)} = 1 \wedge y^{(i)} = 1\}}{\sum_{i=1}^{m} 1\{y^{(i)} = 1\}} \\
\phi_y &= \frac{\sum_{i=1}^{m} 1\{y^{(i)} = 1\}}{m}.
\end{aligned}$$

(c) Consider making a prediction on some new data point x using the most likely class estimate generated by the naive Bayes algorithm. Show that the hypothesis returned by naive Bayes is a linear classifier—i.e., if $p(y = 0|x)$ and $p(y = 1|x)$ are the class probabilities returned by naive Bayes, show that there exists some $\theta \in \mathbb{R}^{n+1}$ such that

$$p(y = 1|x) \geq p(y = 0|x) \text{ if and only if } \theta^T \begin{bmatrix} 1 \\ x \end{bmatrix} \geq 0.$$

(Assume θ_0 is an intercept term.)

5. **Exponential family and the geometric distribution**

(a) Consider the geometric distribution parameterized by ϕ:

$$p(y; \phi) = (1 - \phi)^{y-1}\phi, \ y = 1, 2, 3, \ldots.$$

Show that the geometric distribution is in the exponential family, and give $b(y)$, η, $T(y)$, and $a(\eta)$.

(b) Consider performing regression using a GLM model with a geometric response variable. What is the canonical response function for the family? You may use the fact that the mean of a geometric distribution is given by $1/\phi$.

(c) For a training set $\{(x^{(i)}, y^{(i)}); \ i = 1, \ldots, m\}$, let the log-likelihood of an example be $\log p(y^{(i)}|x^{(i)}; \theta)$. By taking the derivative of the log-likelihood with respect to θ_j, derive the stochastic gradient ascent rule for learning using a GLM model with goemetric responses y and the canonical response function.

CS 229, Public Course
Problem Set #1 Solutions: Supervised Learning

---

1. **Newton's method for computing least squares**

   In this problem, we will prove that if we use Newton's method solve the least squares optimization problem, then we only need one iteration to converge to θ^*.

   (a) Find the Hessian of the cost function $J(\theta) = \frac{1}{2}\sum_{i=1}^m (\theta^T x^{(i)} - y^{(i)})^2$.

   Answer: As shown in the class notes

   $$\frac{\partial J(\theta)}{\partial \theta_j} = \sum_{i=1}^m (\theta^T x^{(i)} - y^{(i)}) x_j^{(i)}.$$

   So

   $$\begin{aligned} \frac{\partial^2 J(\theta)}{\partial \theta_j \partial \theta_k} &= \sum_{i=1}^m \frac{\partial}{\partial \theta_k} (\theta^T x^{(i)} - y^{(i)}) x_j^{(i)} \\ &= \sum_{i=1}^m x_j^{(i)} x_k^{(i)} = (X^T X)_{jk} \end{aligned}$$

   Therefore, the Hessian of $J(\theta)$ is $H = X^T X$. This can also be derived by simply applying rules from the lecture notes on Linear Algebra.

   (b) Show that the first iteration of Newton's method gives us $\theta^\star = (X^T X)^{-1} X^T \vec{y}$, the solution to our least squares problem.

   Answer: Given any $\theta^{(0)}$, Newton's method finds $\theta^{(1)}$ according to

   $$\begin{aligned} \theta^{(1)} &= \theta^{(0)} - H^{-1} \nabla_\theta J(\theta^{(0)}) \\ &= \theta^{(0)} - (X^T X)^{-1} (X^T X \theta^{(0)} - X^T \vec{y}) \\ &= \theta^{(0)} - \theta^{(0)} + (X^T X)^{-1} X^T \vec{y} \\ &= (X^T X)^{-1} X^T \vec{y}. \end{aligned}$$

   Therefore, no matter what $\theta^{(0)}$ we pick, Newton's method always finds $\theta^\star$ after one iteration.

2. **Locally-weighted logistic regression**

   In this problem you will implement a locally-weighted version of logistic regression, where we weight different training examples differently according to the query point. The locally-weighted logistic regression problem is to maximize

   $$\ell(\theta) = -\frac{\lambda}{2}\theta^T\theta + \sum_{i=1}^m w^{(i)} \left[y^{(i)} \log h_\theta(x^{(i)}) + (1 - y^{(i)}) \log(1 - h_\theta(x^{(i)})) \right].$$

The $-\frac{\lambda}{2}\theta^T\theta$ here is what is known as a regularization parameter, which will be discussed in a future lecture, but which we include here because it is needed for Newton's method to perform well on this task. For the entirety of this problem you can use the value $\lambda = 0.0001$.

Using this definition, the gradient of $\ell(\theta)$ is given by

$$\nabla_\theta \ell(\theta) = X^T z - \lambda\theta$$

where $z \in \mathbb{R}^m$ is defined by

$$z_i = w^{(i)}(y^{(i)} - h_\theta(x^{(i)}))$$

and the Hessian is given by

$$H = X^T D X - \lambda I$$

where $D \in \mathbb{R}^{m\times m}$ is a diagonal matrix with

$$D_{ii} = -w^{(i)} h_\theta(x^{(i)})(1 - h_\theta(x^{(i)}))$$

For the sake of this problem you can just use the above formulas, but you should try to derive these results for yourself as well.

Given a query point x, we choose compute the weights

$$w^{(i)} = \exp\left(-\frac{||x - x^{(i)}||^2}{2\tau^2}\right).$$

Much like the locally weighted linear regression that was discussed in class, this weighting scheme gives more when the "nearby" points when predicting the class of a new example.

(a) Implement the Newton-Raphson algorithm for optimizing $\ell(\theta)$ for a new query point x, and use this to predict the class of x.

The `q2/` directory contains data and code for this problem. You should implement the `y = lwlr(X_train, y_train, x, tau)` function in the `lwlr.m` file. This function takes as input the training set (the `X_train` and `y_train` matrices, in the form described in the class notes), a new query point `x` and the weight bandwitdh `tau`. Given this input the function should 1) compute weights $w^{(i)}$ for each training example, using the formula above, 2) maximize $\ell(\theta)$ using Newton's method, and finally 3) output $y = 1\{h_\theta(x) > 0.5\}$ as the prediction.

We provide two additional functions that might help. The `[X_train, y_train] = load_data;` function will load the matrices from files in the `data/` folder. The function `plot_lwlr(X_train, y_train, tau, resolution)` will plot the resulting classifier (assuming you have properly implemented `lwlr.m`). This function evaluates the locally weighted logistic regression classifier over a large grid of points and plots the resulting prediction as blue (predicting $y = 0$) or red (predicting $y = 1$). Depending on how fast your `lwlr` function is, creating the plot might take some time, so we recommend debugging your code with `resolution = 50;` and later increase it to at least 200 to get a better idea of the decision boundary.

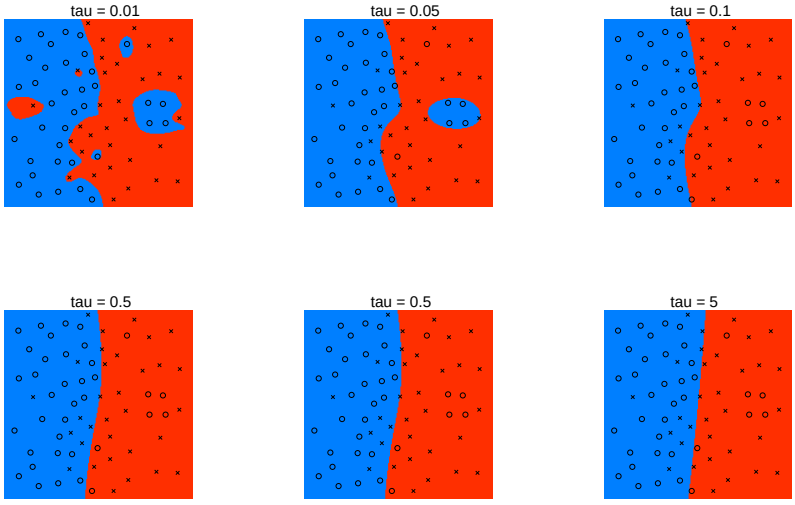
Answer: Our implementation of `lwlr.m`:

```
function y = lwlr(X_train, y_train, x, tau)

m = size(X_train,1);
n = size(X_train,2);
```

```
theta = zeros(n,1);

% compute weights
w = exp(-sum((X_train - repmat(x', m, 1)).^2, 2) / (2*tau));

% perform Newton's method
g = ones(n,1);
while (norm(g) > 1e-6)
  h = 1 ./ (1 + exp(-X_train * theta));
  g = X_train' * (w.*(y_train - h)) - 1e-4*theta;
  H = -X_train' * diag(w.*h.*(1-h)) * X_train - 1e-4*eye(n);
  theta = theta - H \ g;
end

% return predicted y
y = double(x'*theta > 0);
```

(b) Evaluate the system with a variety of different bandwidth parameters τ. In particular, try $\tau = 0.01, 0.050.1, 0.51.0, 5.0$. How does the classification boundary change when varying this parameter? Can you predict what the decision boundary of ordinary (unweighted) logistic regression would look like?

Answer: These are the resulting decision boundaries, for the different values of τ.



For smaller τ, the classifier appears to overfit the data set, obtaining zero training error, but outputting a sporadic looking decision boundary. As τ grows, the resulting decision boundary becomes smoother, eventually converging (in the limit as $\tau \to \infty$ to the unweighted linear regression solution).

3. **Multivariate least squares**

So far in class, we have only considered cases where our target variable y is a scalar value. Suppose that instead of trying to predict a single output, we have a training set with

multiple outputs for each example:

$$\{(x^{(i)}, y^{(i)}),\ i = 1, \ldots, m\},\ x^{(i)} \in \mathbb{R}^n,\ y^{(i)} \in \mathbb{R}^p.$$

Thus for each training example, $y^{(i)}$ is vector-valued, with p entries. We wish to use a linear model to predict the outputs, as in least squares, by specifying the parameter matrix Θ in

$$y = \Theta^T x,$$

where $\Theta \in \mathbb{R}^{n \times p}$.

(a) The cost function for this case is

$$J(\Theta) = \frac{1}{2} \sum_{i=1}^{m} \sum_{j=1}^{p} \left((\Theta^T x^{(i)})_j - y_j^{(i)} \right)^2.$$

Write $J(\Theta)$ in matrix-vector notation (i.e., without using any summations). [Hint: Start with the $m \times n$ design matrix

$$X = \begin{bmatrix} — & (x^{(1)})^T & — \\ — & (x^{(2)})^T & — \\ & \vdots & \\ — & (x^{(m)})^T & — \end{bmatrix}$$

and the $m \times p$ target matrix

$$Y = \begin{bmatrix} — & (y^{(1)})^T & — \\ — & (y^{(2)})^T & — \\ & \vdots & \\ — & (y^{(m)})^T & — \end{bmatrix}$$

and then work out how to express $J(\Theta)$ in terms of these matrices.]

Answer: The objective function can be expressed as

$$J(\Theta) = \frac{1}{2} \mathrm{tr} \left((X\Theta - Y)^T (X\Theta - Y) \right).$$

To see this, note that

$$\begin{aligned} J(\Theta) &= \frac{1}{2} \mathrm{tr} \left((X\Theta - Y)^T (X\Theta - Y) \right) \\ &= \frac{1}{2} \sum_i \left((X\Theta - Y)^T (X\Theta - Y) \right)_{ii} \\ &= \frac{1}{2} \sum_i \sum_j (X\Theta - Y)_{ij}^2 \\ &= \frac{1}{2} \sum_{i=1}^{m} \sum_{j=1}^{p} \left((\Theta^T x^{(i)})_j - y_j^{(i)} \right)^2 \end{aligned}$$

(b) Find the closed form solution for Θ which minimizes $J(\Theta)$. This is the equivalent to the normal equations for the multivariate case.

Answer: First we take the gradient of $J(\Theta)$ with respect to Θ.

$$\begin{aligned}
\nabla_\Theta J(\Theta) &= \nabla_\Theta \left[\frac{1}{2}\mathrm{tr}\left((X\Theta - Y)^T(X\Theta - Y)\right)\right] \\
&= \nabla_\Theta \left[\frac{1}{2}\mathrm{tr}\left(\Theta^T X^T X\Theta - \Theta^T X^T Y - Y^T X\Theta - Y^T T\right)\right] \\
&= \frac{1}{2}\nabla_\Theta \left[\mathrm{tr}(\Theta^T X^T X\Theta) - \mathrm{tr}(\Theta^T X^T Y) - \mathrm{tr}(Y^T X\Theta) + \mathrm{tr}(Y^T Y)\right] \\
&= \frac{1}{2}\nabla_\Theta \left[\mathrm{tr}(\Theta^T X^T X\Theta) - 2\mathrm{tr}(Y^T X\Theta) + \mathrm{tr}(Y^T Y)\right] \\
&= \frac{1}{2}\left[X^T X\Theta + X^T X\Theta - 2X^T Y\right] \\
&= X^T X\Theta - X^T Y
\end{aligned}$$

Setting this expression to zero we obtain

$$\Theta = (X^T X)^{-1} X^T Y.$$

This looks very similar to the closed form solution in the univariate case, except now Y is a $m \times p$ matrix, so then Θ is also a matrix, of size $n \times p$.

(c) Suppose instead of considering the multivariate vectors $y^{(i)}$ all at once, we instead compute each variable $y_j^{(i)}$ separately for each $j = 1, \ldots, p$. In this case, we have a p individual linear models, of the form

$$y_j^{(i)} = \theta_j^T x^{(i)},\ j = 1, \ldots, p.$$

(So here, each $\theta_j \in \mathbb{R}^n$). How do the parameters from these p independent least squares problems compare to the multivariate solution?

Answer: This time, we construct a set of vectors

$$\vec{y}_j = \begin{bmatrix} y_j^{(1)} \\ y_j^{(2)} \\ \vdots \\ y_j^{(m)} \end{bmatrix},\ j = 1, \ldots, p.$$

Then our j-th linear model can be solved by the least squares solution

$$\theta_j = (X^T X)^{-1} X^T \vec{y}_j.$$

If we line up our θ_j, we see that we have the following equation:

$$\begin{aligned}
[\theta_1\ \theta_2\ \cdots\ \theta_p] &= \left[(X^T X)^{-1} X^T \vec{y}_1\ (X^T X)^{-1} X^T \vec{y}_2\ \cdots\ (X^T X)^{-1} X^T \vec{y}_p\right] \\
&= (X^T X)^{-1} X^T \left[\vec{y}_1\ \vec{y}_2\ \cdots\ \vec{y}_p\right] \\
&= (X^T X)^{-1} X^T Y \\
&= \Theta.
\end{aligned}$$

Thus, our p individual least squares problems give the exact same solution as the multivariate least squares.

4. **Naive Bayes**

In this problem, we look at maximum likelihood parameter estimation using the naive Bayes assumption. Here, the input features $x_j,\ j = 1, \ldots, n$ to our model are discrete, binary-valued variables, so $x_j \in \{0, 1\}$. We call $x = [x_1\ x_2\ \cdots\ x_n]^T$ to be the input vector. For each training example, our output targets are a single binary-value $y \in \{0, 1\}$. Our model is then parameterized by $\phi_{j|y=0} = p(x_j = 1|y = 0)$, $\phi_{j|y=1} = p(x_j = 1|y = 1)$, and $\phi_y = p(y = 1)$. We model the joint distribution of (x, y) according to

$$\begin{aligned}
p(y) &= (\phi_y)^y(1-\phi_y)^{1-y} \\
p(x|y=0) &= \prod_{j=1}^{n} p(x_j|y=0) \\
&= \prod_{j=1}^{n} (\phi_{j|y=0})^{x_j}(1-\phi_{j|y=0})^{1-x_j} \\
p(x|y=1) &= \prod_{j=1}^{n} p(x_j|y=1) \\
&= \prod_{j=1}^{n} (\phi_{j|y=1})^{x_j}(1-\phi_{j|y=1})^{1-x_j}
\end{aligned}$$

(a) Find the joint likelihood function $\ell(\varphi) = \log \prod_{i=1}^{m} p(x^{(i)}, y^{(i)}; \varphi)$ in terms of the model parameters given above. Here, φ represents the entire set of parameters $\{\phi_y,\ \phi_{j|y=0},\ \phi_{j|y=1},\ j = 1, \ldots, n\}$.

Answer:

$$\begin{aligned}
\ell(\varphi) &= \log \prod_{i=1}^{m} p(x^{(i)}, y^{(i)}; \varphi) \\
&= \log \prod_{i=1}^{m} p(x^{(i)}|y^{(i)}; \varphi) p(y^{(i)}; \varphi) \\
&= \log \prod_{i=1}^{m} \left(\prod_{j=1}^{n} p(x_j^{(i)}|y^{(i)}; \varphi) \right) p(y^{(i)}; \varphi) \\
&= \sum_{i=1}^{m} \left(\log p(y^{(i)}; \varphi) + \sum_{j=1}^{n} \log p(x_j^{(i)}|y^{(i)}; \varphi) \right) \\
&= \sum_{i=1}^{m} \Bigg[y^{(i)} \log \phi_y + (1 - y^{(i)}) \log(1 - \phi_y) \\
&\qquad + \sum_{j=1}^{n} \left(x_j^{(i)} \log \phi_{j|y^{(i)}} + (1 - x_j^{(i)}) \log(1 - \phi_{j|y^{(i)}}) \right) \Bigg]
\end{aligned}$$

(b) Show that the parameters which maximize the likelihood function are the same as

those given in the lecture notes; i.e., that

$$\begin{aligned}
\phi_{j|y=0} &= \frac{\sum_{i=1}^m 1\{x_j^{(i)} = 1 \wedge y^{(i)} = 0\}}{\sum_{i=1}^m 1\{y^{(i)} = 0\}} \\
\phi_{j|y=1} &= \frac{\sum_{i=1}^m 1\{x_j^{(i)} = 1 \wedge y^{(i)} = 1\}}{\sum_{i=1}^m 1\{y^{(i)} = 1\}} \\
\phi_y &= \frac{\sum_{i=1}^m 1\{y^{(i)} = 1\}}{m}.
\end{aligned}$$

Answer: The only terms in $\ell(\varphi)$ which have non-zero gradient with respect to $\phi_{j|y=0}$ are those which include $\phi_{j|y^{(i)}}$. Therefore,

$$\begin{aligned}
\nabla_{\phi_{j|y=0}} \ell(\varphi) &= \nabla_{\phi_{j|y=0}} \sum_{i=1}^m \left(x_j^{(i)} \log \phi_{j|y^{(i)}} + (1 - x_j^{(i)}) \log(1 - \phi_{j|y^{(i)}}) \right) \\
&= \nabla_{\phi_{j|y=0}} \sum_{i=1}^m \Big(x_j^{(i)} \log(\phi_{j|y=0}) 1\{y^{(i)} = 0\} \\
&\qquad\qquad + (1 - x_j^{(i)}) \log(1 - \phi_{j|y=0}) 1\{y^{(i)} = 0\} \Big) \\
&= \sum_{i=1}^m \left(x_j^{(i)} \frac{1}{\phi_{j|y=0}} 1\{y^{(i)} = 0\} - (1 - x_j^{(i)}) \frac{1}{1 - \phi_{j|y=0}} 1\{y^{(i)} = 0\} \right).
\end{aligned}$$

Setting $\nabla_{\phi_{j|y=0}} \ell(\varphi) = 0$ gives

$$\begin{aligned}
0 &= \sum_{i=1}^m \left(x_j^{(i)} \frac{1}{\phi_{j|y=0}} 1\{y^{(i)} = 0\} - (1 - x_j^{(i)}) \frac{1}{1 - \phi_{j|y=0}} 1\{y^{(i)} = 0\} \right) \\
&= \sum_{i=1}^m \left(x_j^{(i)} (1 - \phi_{j|y=0}) 1\{y^{(i)} = 0\} - (1 - x_j^{(i)}) \phi_{j|y=0} 1\{y^{(i)} = 0\} \right) \\
&= \sum_{i=1}^m \left((x_j^{(i)} - \phi_{j|y=0}) 1\{y^{(i)} = 0\} \right) \\
&= \sum_{i=1}^m \left(x_j^{(i)} \cdot 1\{y^{(i)} = 0\} \right) - \phi_{j|y=0} \sum_{i=1}^m 1\{y^{(i)} = 0\} \\
&= \sum_{i=1}^m \left(1\{x_j^{(i)} = 1 \wedge y^{(i)} = 0\} \right) - \phi_{j|y=0} \sum_{i=1}^m 1\{y^{(i)} = 0\}.
\end{aligned}$$

We then arrive at our desired result

$$\phi_{j|y=0} = \frac{\sum_{i=1}^m 1\{x_j^{(i)} = 1 \wedge y^{(i)} = 0\}}{\sum_{i=1}^m 1\{y^{(i)} = 0\}}$$

The solution for $\phi_{j|y=1}$ proceeds in the identical manner.

To solve for ϕ_y,

$$\begin{aligned}\nabla_{\phi_y}\ell(\varphi) &= \nabla_{\phi_y}\sum_{i=1}^{m}\Big(y^{(i)}\log\phi_y + (1-y^{(i)})\log(1-\phi_y)\Big)\\ &= \sum_{i=1}^{m}\left(y^{(i)}\frac{1}{\phi_y} - (1-y^{(i)})\frac{1}{1-\phi_y}\right)\end{aligned}$$

Then setting $\nabla_{\phi_y} = 0$ gives us

$$\begin{aligned}0 &= \sum_{i=1}^{m}\left(y^{(i)}\frac{1}{\phi_y} - (1-y^{(i)})\frac{1}{1-\phi_y}\right)\\ &= \sum_{i=1}^{m}\Big(y^{(i)}(1-\phi_y) - (1-y^{(i)})\phi_y\Big)\\ &= \sum_{i=1}^{m}y^{(i)} - \sum_{i=1}^{m}\phi_y.\end{aligned}$$

Therefore,

$$\phi_y = \frac{\sum_{i=1}^{m}1\{y^{(i)}=1\}}{m}.$$

(c) Consider making a prediction on some new data point x using the most likely class estimate generated by the naive Bayes algorithm. Show that the hypothesis returned by naive Bayes is a linear classifier—i.e., if $p(y=0|x)$ and $p(y=1|x)$ are the class probabilities returned by naive Bayes, show that there exists some $\theta \in \mathbb{R}^{n+1}$ such that

$$p(y=1|x) \geq p(y=0|x) \text{ if and only if } \theta^T\begin{bmatrix}1\\ x\end{bmatrix} \geq 0.$$

(Assume θ_0 is an intercept term.)

Answer:

$$\begin{aligned}&p(y=1|x) \geq p(y=0|x)\\ \iff &\frac{p(y=1|x)}{p(y=0|x)} \geq 1\\ \iff &\frac{\left(\prod_{j=1}^{n}p(x_j|y=1)\right)p(y=1)}{\left(\prod_{j=1}^{n}p(x_j|y=0)\right)p(y=0)} \geq 1\\ \iff &\frac{\left(\prod_{j=1}^{n}(\phi_{j|y=0})^{x_j}(1-\phi_{j|y=0})^{1-x_j}\right)\phi_y}{\left(\prod_{j=1}^{n}(\phi_{j|y=1})^{x_j}(1-\phi_{j|y=1})^{1-x_j}\right)(1-\phi_y)} \geq 1\\ \iff &\sum_{j=1}^{n}\left(x_j\log\left(\frac{\phi_{j|y=1}}{\phi_{j|y=0}}\right) + (1-x_j)\log\left(\frac{1-\phi_{j|y=0}}{1-\phi_j|y=0}\right)\right) + \log\left(\frac{\phi_y}{1-\phi_y}\right) \geq 0\\ \iff &\sum_{j=1}^{n}x_j\log\left(\frac{(\phi_{j|y=1})(1-\phi_{j|y=0})}{(\phi_{j|y=0})(1-\phi_{j|y=1})}\right) + \sum_{j=1}^{n}\log\left(\frac{1-\phi_{j|y=1}}{1-\phi_{j|y=0}}\right) + \log\left(\frac{\phi_y}{1-\phi_y}\right) \geq 0\\ \iff &\theta^T\begin{bmatrix}1\\ x\end{bmatrix} \geq 0,\end{aligned}$$

where

$$\begin{aligned}\theta_0 &= \sum_{j=1}^{n} \log\left(\frac{1-\phi_{j|y=1}}{1-\phi_{j|y=0}}\right) + \log\left(\frac{\phi_y}{1-\phi_y}\right) \\ \theta_j &= \log\left(\frac{(\phi_{j|y=1})(1-\phi_{j|y=0})}{(\phi_{j|y=0})(1-\phi_{j|y=1})}\right),\ j = 1,\ldots,n.\end{aligned}$$

5. **Exponential family and the geometric distribution**

(a) Consider the geometric distribution parameterized by ϕ:

$$p(y;\phi) = (1-\phi)^{y-1}\phi,\ y = 1,2,3,\ldots.$$

Show that the geometric distribution is in the exponential family, and give $b(y)$, η, $T(y)$, and $a(\eta)$.

Answer:

$$\begin{aligned}p(y;\phi) &= (1-\phi)^{y-1}\phi \\ &= \exp\left[\log(1-\phi)^{y-1} + \log\phi\right] \\ &= \exp\left[(y-1)\log(1-\phi) + \log\phi\right] \\ &= \exp\left[y\log(1-\phi) - \log\left(\frac{1-\phi}{\phi}\right)\right]\end{aligned}$$

Then

$$\begin{aligned}b(y) &= 1 \\ \eta &= \log(1-\phi) \\ T(y) &= y \\ a(\eta) &= \log\left(\frac{1-\phi}{\phi}\right) = \log\left(\frac{e^\eta}{1-e^\eta}\right),\end{aligned}$$

where the last line follows becuase $\eta = \log(1-\phi) \Rightarrow e^\eta = 1-\phi \Rightarrow \phi = 1-e^\eta$.

(b) Consider performing regression using a GLM model with a geometric response variable. What is the canonical response function for the family? You may use the fact that the mean of a geometric distribution is given by $1/\phi$.

Answer:

$$g(\eta) = E[y;\phi] = \frac{1}{\phi} = \frac{1}{1-e^\eta}.$$

(c) For a training set $\{(x^{(i)}, y^{(i)});\ i = 1,\ldots,m\}$, let the log-likelihood of an example be $\log p(y^{(i)}|x^{(i)};\theta)$. By taking the derivative of the log-likelihood with respect to θ_j, derive the stochastic gradient ascent rule for learning using a GLM model with goemetric responses y and the canonical response function.

Answer: The log-likelihood of an example $(x^{(i)}, y^{(i)})$ is defined as $\ell(\theta) = \log p(y^{(i)}|x^{(i)};\theta)$. To derive the stochastic gradient ascent rule, use the results from previous parts and the standard GLM assumption that $\eta = \theta^T x$.

$$\begin{aligned}
\ell_i(\theta) &= \log\left[\exp\left(\theta^T x^{(i)} \cdot y^{(i)} - \log\left(\frac{e^{\theta^T x^{(i)}}}{1 - e^{\theta^T x^{(i)}}}\right)\right)\right] \\
&= \log\left[\exp\left(\theta^T x^{(i)} \cdot y^{(i)} - \log\left(\frac{1}{e^{-\theta^T x^{(i)}} - 1}\right)\right)\right] \\
&= \theta^T x^{(i)} \cdot y^{(i)} + \log\left(e^{-\theta^T x^{(i)}} - 1\right) \\
\frac{\partial}{\partial \theta_j}\ell_i(\theta) &= x_j^{(i)} y^{(i)} + \frac{e^{-\theta^T x^{(i)}}}{e^{-\theta^T x^{(i)}} - 1}(-x_j^{(i)}) \\
&= x_j^{(i)} y^{(i)} - \frac{1}{1 - e^{-\theta^T x^{(i)}}} x_j^{(i)} \\
&= \left(y^{(i)} - \frac{1}{1 - e^{\theta^T x^{(i)}}}\right) x_j^{(i)}.
\end{aligned}$$

Thus the stochastic gradient ascent update rule should be

$$\theta_j := \theta_j + \alpha \frac{\partial \ell_i(\theta)}{\partial \theta_j},$$

which is

$$\theta_j := \theta_j + \alpha \left(y^{(i)} - \frac{1}{1 - e^{\theta^T x^{(i)}}}\right) x_j^{(i)}.$$

CS 229, Public Course
Problem Set #2: Kernels, SVMs, and Theory

---

1. **Kernel ridge regression**

   In contrast to ordinary least squares which has a cost function

   $$J(\theta) = \frac{1}{2}\sum_{i=1}^{m}(\theta^T x^{(i)} - y^{(i)})^2,$$

   we can also add a term that penalizes large weights in θ. In *ridge regression*, our least squares cost is regularized by adding a term $\lambda\|\theta\|^2$, where $\lambda > 0$ is a fixed (known) constant (regularization will be discussed at greater length in an upcoming course lecutre). The ridge regression cost function is then

   $$J(\theta) = \frac{1}{2}\sum_{i=1}^{m}(\theta^T x^{(i)} - y^{(i)})^2 + \frac{\lambda}{2}\|\theta\|^2.$$

   (a) Use the vector notation described in class to find a closed-form expreesion for the value of θ which minimizes the ridge regression cost function.

   (b) Suppose that we want to use kernels to implicitly represent our feature vectors in a high-dimensional (possibly infinite dimensional) space. Using a feature mapping ϕ, the ridge regression cost function becomes

   $$J(\theta) = \frac{1}{2}\sum_{i=1}^{m}(\theta^T \phi(x^{(i)}) - y^{(i)})^2 + \frac{\lambda}{2}\|\theta\|^2.$$

   Making a prediction on a new input x_{new} would now be done by computing $\theta^T\phi(x_{\text{new}})$. Show how we can use the "kernel trick" to obtain a closed form for the prediction on the new input without ever explicitly computing $\phi(x_{\text{new}})$. You may assume that the parameter vector θ can be expressed as a linear combination of the input feature vectors; i.e., $\theta = \sum_{i=1}^{m}\alpha_i\phi(x^{(i)})$ for some set of parameters α_i.
   [Hint: You may find the following identity useful:

   $$(\lambda I + BA)^{-1}B = B(\lambda I + AB)^{-1}.$$

   If you want, you can try to prove this as well, though this is not required for the problem.]

2. **ℓ_2 norm soft margin SVMs**

   In class, we saw that if our data is not linearly separable, then we need to modify our support vector machine algorithm by introducing an error margin that must be minimized. Specifically, the formulation we have looked at is known as the ℓ_1 norm soft margin SVM. In this problem we will consider an alternative method, known as the ℓ_2 norm soft margin SVM. This new algorithm is given by the following optimization problem (notice that the slack penalties are now squared):

   $$\begin{aligned} \min_{w,b,\xi} \quad & \tfrac{1}{2}\|w\|^2 + \tfrac{C}{2}\textstyle\sum_{i=1}^{m}\xi_i^2 \\ \text{s.t.} \quad & y^{(i)}(w^T x^{(i)} + b) \geq 1 - \xi_i,\ i = 1, \ldots, m \end{aligned}.$$

(a) Notice that we have dropped the $\xi_i \geq 0$ constraint in the ℓ_2 problem. Show that these non-negativity constraints can be removed. That is, show that the optimal value of the objective will be the same whether or not these constraints are present.

(b) What is the Lagrangian of the ℓ_2 soft margin SVM optimization problem?

(c) Minimize the Lagrangian with respect to w, b, and ξ by taking the following gradients: $\nabla_w \mathcal{L}$, $\frac{\partial \mathcal{L}}{\partial b}$, and $\nabla_\xi \mathcal{L}$, and then setting them equal to 0. Here, $\xi = [\xi_1, \xi_2, \ldots, \xi_m]^T$.

(d) What is the dual of the ℓ_2 soft margin SVM optimization problem?

3. **SVM with Gaussian kernel**

Consider the task of training a support vector machine using the Gaussian kernel $K(x, z) = \exp(-\|x - z\|^2/\tau^2)$. We will show that as long as there are no two identical points in the training set, we can always find a value for the bandwidth parameter τ such that the SVM achieves zero training error.

(a) Recall from class that the decision function learned by the support vector machine can be written as

$$f(x) = \sum_{i=1}^{m} \alpha_i y^{(i)} K(x^{(i)}, x) + b.$$

Assume that the training data $\{(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})\}$ consists of points which are separated by at least a distance of ϵ; that is, $||x^{(j)} - x^{(i)}|| \geq \epsilon$ for any $i \neq j$. Find values for the set of parameters $\{\alpha_1, \ldots, \alpha_m, b\}$ and Gaussian kernel width τ such that $x^{(i)}$ is correctly classified, for all $i = 1, \ldots, m$. [Hint: Let $\alpha_i = 1$ for all i and $b = 0$. Now notice that for $y \in \{-1, +1\}$ the prediction on $x^{(i)}$ will be correct if $|f(x^{(i)}) - y^{(i)}| < 1$, so find a value of τ that satisfies this inequality for all i.]

(b) Suppose we run a SVM with slack variables using the parameter τ you found in part (a). Will the resulting classifier necessarily obtain zero training error? Why or why not? A short explanation (without proof) will suffice.

(c) Suppose we run the SMO algorithm to train an SVM with slack variables, under the conditions stated above, using the value of τ you picked in the previous part, and using some arbitrary value of C (which you do not know beforehand). Will this necessarily result in a classifier that achieve zero training error? Why or why not? Again, a short explanation is sufficient.

4. **Naive Bayes and SVMs for Spam Classification**

In this question you'll look into the Naive Bayes and Support Vector Machine algorithms for a spam classification problem. However, instead of implementing the algorithms yourself, you'll use a freely available machine learning library. There are many such libraries available, with different strengths and weaknesses, but for this problem you'll use the WEKA machine learning package, available at `http://www.cs.waikato.ac.nz/ml/weka/`. WEKA implements many standard machine learning algorithms, is written in Java, and has both a GUI and a command line interface. It is not the best library for very large-scale data sets, but it is very nice for playing around with many different algorithms on medium size problems.

You can download and install WEKA by following the instructions given on the website above. To use it from the command line, you first need to install a java runtime environment, then add the `weka.jar` file to your `CLASSPATH` environment variable. Finally, you

can call WEKA using the command:
`java <classifier> -t <training file> -T <test file>`
For example, to run the Naive Bayes classifier (using the multinomial event model) on our provided spam data set by running the command:
`java weka.classifiers.bayes.NaiveBayesMultinomial -t spam_train_1000.arff -T spam_test.arff`

The spam classification dataset in the `q4/` directory was provided courtesy of Christian Shelton (cshelton@cs.ucr.edu). Each example corresponds to a particular email, and each feature correspondes to a particular word. For privacy reasons we have removed the actual words themselves from the data set, and instead label the features generically as `f1`, `f2`, etc. However, the data set is from a real spam classification task, so the results demonstrate the performance of these algorithms on a real-world problem. The `q4/` directory actually contains several different training files, named `spam_train_50.arff`, `spam_train_100.arff`, etc (the ".arff" format is the default format by WEKA), each containing the corresponding number of training examples. There is also a single test set `spam_test.arff`, which is a hold out set used for evaluating the classifier's performance.

(a) Run the `weka.classifiers.bayes.NaiveBayesMultinomial` classifier on the dataset and report the resulting error rates. Evaluate the performance of the classifier using each of the different training files (but each time using the same test file, `spam_test.arff`). Plot the error rate of the classifier versus the number of training examples.

(b) Repeat the previous part, but using the `weka.classifiers.functions.SMO` classifier, which implements the SMO algorithm to train an SVM. How does the performance of the SVM compare to that of Naive Bayes?

5. **Uniform convergence**

In class we proved that for any finite set of hypotheses $\mathcal{H} = \{h_1, \ldots, h_k\}$, if we pick the hypothesis $\hat{h}$ that minimizes the training error on a set of m examples, then with probability at least $(1 - \delta)$,

$$\varepsilon(\hat{h}) \leq \left(\min_i \varepsilon(h_i)\right) + 2\sqrt{\frac{1}{2m} \log \frac{2k}{\delta}},$$

where $\varepsilon(h_i)$ is the generalization error of hypothesis h_i. Now consider a special case (often called the *realizable* case) where we know, a priori, that there is some hypothesis in our class $\mathcal{H}$ that achieves zero error on the distribution from which the data is drawn. Then we could obviously just use the above bound with $\min_i \varepsilon(h_i) = 0$; however, we can prove a better bound than this.

(a) Consider a learning algorithm which, after looking at m training examples, chooses some hypothesis $\hat{h} \in \mathcal{H}$ that makes zero mistakes on this training data. (By our assumption, there is at least one such hypothesis, possibly more.) Show that with probability $1 - \delta$

$$\varepsilon(\hat{h}) \leq \frac{1}{m} \log \frac{k}{\delta}.$$

Notice that since we do not have a square root here, this bound is much tighter. [Hint: Consider the probability that a hypothesis with generalization error greater than γ makes no mistakes on the training data. Instead of the Hoeffding bound, you might also find the following inequality useful: $(1 - \gamma)^m \leq e^{-\gamma m}$.]

(b) Rewrite the above bound as a sample complexity bound, i.e., in the form: for fixed δ and γ, for $\varepsilon(\hat{h}) \leq \gamma$ to hold with probability at least $(1 - \delta)$, it suffices that $m \geq f(k, \gamma, \delta)$ (i.e., $f(\cdot)$ is some function of k, γ, and δ).

CS 229, Public Course
Problem Set #2 Solutions: Kernels, SVMs, and Theory

---

1. **Kernel ridge regression**

   In contrast to ordinary least squares which has a cost function

   $$J(\theta) = \frac{1}{2}\sum_{i=1}^{m}(\theta^T x^{(i)} - y^{(i)})^2,$$

   we can also add a term that penalizes large weights in θ. In *ridge regression*, our least squares cost is regularized by adding a term $\lambda\|\theta\|^2$, where $\lambda > 0$ is a fixed (known) constant (regularization will be discussed at greater length in an upcoming course lecutre). The ridge regression cost function is then

   $$J(\theta) = \frac{1}{2}\sum_{i=1}^{m}(\theta^T x^{(i)} - y^{(i)})^2 + \frac{\lambda}{2}\|\theta\|^2.$$

   (a) Use the vector notation described in class to find a closed-form expreesion for the value of θ which minimizes the ridge regression cost function.

   Answer: Using the design matrix notation, we can rewrite $J(\theta)$ as

   $$J(\theta) = \frac{1}{2}(X\theta - \vec{y})^T(X\theta - \vec{y}) + \frac{\lambda}{2}\theta^T\theta.$$

   Then the gradient is

   $$\nabla_\theta J(\theta) = X^T X\theta - X^T\vec{y} + \lambda\theta.$$

   Setting the gradient to 0 gives us

   $$\begin{aligned} 0 &= X^T X\theta - X^T\vec{y} + \lambda\theta \\ \theta &= (X^T X + \lambda I)^{-1}X^T\vec{y}. \end{aligned}$$

   (b) Suppose that we want to use kernels to implicitly represent our feature vectors in a high-dimensional (possibly infinite dimensional) space. Using a feature mapping ϕ, the ridge regression cost function becomes

   $$J(\theta) = \frac{1}{2}\sum_{i=1}^{m}(\theta^T\phi(x^{(i)}) - y^{(i)})^2 + \frac{\lambda}{2}\|\theta\|^2.$$

   Making a prediction on a new input x_{new} would now be done by computing $\theta^T\phi(x_{\text{new}})$. Show how we can use the "kernel trick" to obtain a closed form for the prediction on the new input without ever explicitly computing $\phi(x_{\text{new}})$. You may assume that the parameter vector θ can be expressed as a linear combination of the input feature vectors; i.e., $\theta = \sum_{i=1}^{m}\alpha_i\phi(x^{(i)})$ for some set of parameters α_i.

[Hint: You may find the following identity useful:

$$(\lambda I + BA)^{-1}B = B(\lambda I + AB)^{-1}.$$

If you want, you can try to prove this as well, though this is not required for the problem.]

Answer: Let Φ be the design matrix associated with the feature vectors $\phi(x^{(i)})$. Then from parts (a) and (b),

$$\begin{aligned}\theta &= \left(\Phi^T\Phi + \lambda I\right)^{-1}\Phi^T\vec{y} \\ &= \Phi^T\left(\Phi\Phi^T + \lambda I\right)^{-1}\vec{y} \\ &= \Phi^T(K + \lambda I)^{-1}\vec{y}.\end{aligned}$$

where K is the kernel matrix for the training set (since $\Phi_{i,j} = \phi(x^{(i)})^T\phi(x^{(j)}) = K_{ij}$.) To predict a new value y_{new}, we can compute

$$\begin{aligned}\vec{y}_{\text{new}} &= \theta^T\phi(x_{\text{new}}) \\ &= \vec{y}^T(K + \lambda I)^{-1}\Phi\phi(x_{\text{new}}) \\ &= \sum_{i=1}^{m}\alpha_i K(x^{(i)}, x_{\text{new}}).\end{aligned}$$

where $\alpha = (K + \lambda I)^{-1}\vec{y}$. All these terms can be efficiently computing using the kernel function.

To prove the identity from the hint, we left-multiply by $\lambda(I + BA)$ and right-multiply by $\lambda(I + AB)$ on both sides. That is,

$$\begin{aligned}(\lambda I + BA)^{-1}B &= B(\lambda I + AB)^{-1} \\ B &= (\lambda I + BA)B(\lambda I + AB)^{-1} \\ B(\lambda I + AB) &= (\lambda I + BA)B \\ \lambda B + BAB &= \lambda B + BAB.\end{aligned}$$

This last line clearly holds, proving the identity.

2. ℓ_2 **norm soft margin SVMs**

In class, we saw that if our data is not linearly separable, then we need to modify our support vector machine algorithm by introducing an error margin that must be minimized. Specifically, the formulation we have looked at is known as the ℓ_1 norm soft margin SVM. In this problem we will consider an alternative method, known as the ℓ_2 norm soft margin SVM. This new algorithm is given by the following optimization problem (notice that the slack penalties are now squared):

$$\begin{aligned}\min_{w,b,\xi} \quad & \tfrac{1}{2}\|w\|^2 + \tfrac{C}{2}\textstyle\sum_{i=1}^{m}\xi_i^2 \\ \text{s.t.} \quad & y^{(i)}(w^Tx^{(i)} + b) \geq 1 - \xi_i,\ i = 1, \ldots, m\end{aligned}.$$

(a) Notice that we have dropped the $\xi_i \geq 0$ constraint in the ℓ_2 problem. Show that these non-negativity constraints can be removed. That is, show that the optimal value of the objective will be the same whether or not these constraints are present.

Answer: Consider a potential solution to the above problem with some $\xi < 0$. Then the constraint $y^{(i)}(w^Tx^{(i)} + b) \geq 1 - \xi_i$ would also be satisfied for $\xi_i = 0$, and the objective function would be lower, proving that this could not be an optimal solution.

(b) What is the Lagrangian of the ℓ_2 soft margin SVM optimization problem?
Answer:

$$\mathcal{L}(w, b, \xi, \alpha) = \frac{1}{2} w^T w + \frac{C}{2} \sum_{i=1}^{m} \xi_i^2 - \sum_{i=1}^{m} \alpha_i [y^{(i)}(w^T x^{(i)} + b) - 1 + \xi_i],$$

where $\alpha_i \geq 0$ for $i = 1, \ldots, m$.

(c) Minimize the Lagrangian with respect to w, b, and ξ by taking the following gradients: $\nabla_w \mathcal{L}$, $\frac{\partial \mathcal{L}}{\partial b}$, and $\nabla_\xi \mathcal{L}$, and then setting them equal to 0. Here, $\xi = [\xi_1, \xi_2, \ldots, \xi_m]^T$.
Answer: Taking the gradient with respect to w, we get

$$0 = \nabla_w \mathcal{L} = w - \sum_{i=1}^{m} \alpha_i y^{(i)} x^{(i)},$$

which gives us

$$w = \sum_{i=1}^{m} \alpha_i y^{(i)} x^{(i)}.$$

Taking the derivative with respect to b, we get

$$0 = \frac{\partial \mathcal{L}}{\partial b} = -\sum_{i=1}^{m} \alpha_i y^{(i)},$$

giving us

$$0 = \sum_{i=1}^{m} \alpha_i y^{(i)}.$$

Finally, taking the gradient with respect to ξ, we have

$$0 = \nabla_\xi \mathcal{L} = C\xi - \alpha,$$

where $\alpha = [\alpha_1, \alpha_2, \ldots, \alpha_m]^T$. Thus, for each $i = 1, \ldots, m$, we get

$$0 = C\xi_i - \alpha_i \quad \Rightarrow \quad C\xi_i = \alpha_i.$$

(d) What is the dual of the ℓ_2 soft margin SVM optimization problem?

Answer: The objective function for the dual is

$$\begin{aligned}
W(\alpha) &= \min_{w,b,\xi} \mathcal{L}(w,b,\xi,\alpha) \\
&= \frac{1}{2}\sum_{i=1}^{m}\sum_{j=1}^{m}(\alpha_i y^{(i)} x^{(i)})^T(\alpha_j y^{(j)} x^{(j)}) + \frac{1}{2}\sum_{i=1}^{m}\frac{\alpha_i}{\xi_i}\xi_i^2 \\
&\quad - \sum_{i=1}^{m}\alpha_i\left[y^{(i)}\left(\left(\sum_{j=1}^{m}\alpha_j y^{(j)} x^{(j)}\right)^T x^{(i)} + b\right) - 1 + \xi_i\right] \\
&= -\frac{1}{2}\sum_{i=1}^{m}\sum_{j=1}^{m}\alpha_i\alpha_j y^{(i)} y^{(j)}(x^{(i)})^T x^{(j)} + \frac{1}{2}\sum_{i=1}^{m}\alpha_i\xi_i \\
&\quad - \left(\sum_{i=1}^{m}\alpha_i y^{(i)}\right) b + \sum_{i=1}^{m}\alpha_i - \sum_{i=1}^{m}\alpha_i\xi_i \\
&= \sum_{i=1}^{m}\alpha_i - \frac{1}{2}\sum_{i=1}^{m}\sum_{j=1}^{m}\alpha_i\alpha_j y^{(i)} y^{(j)}(x^{(i)})^T x^{(j)} - \frac{1}{2}\sum_{i=1}^{m}\alpha_i\xi_i \\
&= \sum_{i=1}^{m}\alpha_i - \frac{1}{2}\sum_{i=1}^{m}\sum_{j=1}^{m}\alpha_i\alpha_j y^{(i)} y^{(j)}(x^{(i)})^T x^{(j)} - \frac{1}{2}\sum_{i=1}^{m}\frac{\alpha_i^2}{C}.
\end{aligned}$$

Then the dual formulation of our problem is

$$\begin{array}{ll}
\max_\alpha & \sum_{i=1}^{m}\alpha_i - \frac{1}{2}\sum_{i=1}^{m}\sum_{j=1}^{m}\alpha_i\alpha_j y^{(i)} y^{(j)}(x^{(i)})^T x^{(j)} - \frac{1}{2}\sum_{i=1}^{m}\frac{\alpha_i^2}{C} \\
\text{s.t.} & \alpha_i \geq 0,\ i = 1,\ldots,m \\
& \sum_{i=1}^{m}\alpha_i y^{(i)} = 0
\end{array}.$$

3. **SVM with Gaussian kernel**

Consider the task of training a support vector machine using the Gaussian kernel $K(x,z) = \exp(-\|x-z\|^2/\tau^2)$. We will show that as long as there are no two identical points in the training set, we can always find a value for the bandwidth parameter τ such that the SVM achieves zero training error.

(a) Recall from class that the decision function learned by the support vector machine can be written as

$$f(x) = \sum_{i=1}^{m}\alpha_i y^{(i)} K(x^{(i)}, x) + b.$$

Assume that the training data $\{(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})\}$ consists of points which are separated by at least a distance of ϵ; that is, $||x^{(j)} - x^{(i)}|| \geq \epsilon$ for any $i \neq j$. Find values for the set of parameters $\{\alpha_1, \ldots, \alpha_m, b\}$ and Gaussian kernel width τ such that $x^{(i)}$ is correctly classified, for all $i = 1, \ldots, m$. [Hint: Let $\alpha_i = 1$ for all i and $b = 0$. Now notice that for $y \in \{-1, +1\}$ the prediction on $x^{(i)}$ will be correct if $|f(x^{(i)}) - y^{(i)}| < 1$, so find a value of τ that satisfies this inequality for all i.]

Answer: First we set $\alpha_i = 1$ for all $i = 1, \ldots, m$ and $b = 0$. Then, for a training example $\{x^{(i)}, y^{(i)}\}$, we get

$$\begin{aligned}
\left| f(x^{(i)}) - y^{(i)} \right| &= \left| \sum_{j=1}^{m} y^{(j)} K(x^{(j)}, x^{(i)}) - y^{(i)} \right| \\
&= \left| \sum_{j=1}^{m} y^{(j)} \exp\left(-\|x^{(j)} - x^{(i)}\|^2 / \tau^2 \right) - y^{(i)} \right| \\
&= \left| y^{(i)} + \sum_{j \neq i} y^{(j)} \exp\left(\|x^{(j)} - x^{(i)}\|^2 / \tau^2 \right) - y^{(i)} \right| \\
&= \left| \sum_{j \neq i} y^{(j)} \exp\left(-\|x^{(j)} - x^{(i)}\|^2 / \tau^2 \right) \right| \\
&\leq \sum_{j \neq i} \left| y^{(j)} \exp\left(-\|x^{(j)} - x^{(i)}\|^2 / \tau^2 \right) \right| \\
&= \sum_{j \neq i} \left| y^{(j)} \right| \cdot \exp\left(\|x^{(j)} - x^{(i)}\|^2 / \tau^2 \right) \\
&= \sum_{j \neq i} \exp\left(-\|x^{(j)} - x^{(i)}\|^2 / \tau^2 \right) \\
&\leq \sum_{j \neq i} \exp\left(-\epsilon^2 / \tau^2 \right) \\
&= (m-1) \exp\left(-\epsilon^2 / \tau^2 \right).
\end{aligned}$$

The first inequality comes from repeated application of the triangle inequality $|a + b| \leq |a| + |b|$, and the second inequality (1) from the assumption that $||x^{(j)} - x^{(i)}|| \geq \epsilon$ for all $i \neq j$. Thus we need to choose a γ such that

$$(m-1) \exp(-\epsilon^2 / \tau^2) < 1,$$

or

$$\tau < \frac{\epsilon}{\log(m-1)}.$$

By choosing, for example, $\tau = \epsilon / \log m$ we are done.

(b) Suppose we run a SVM with slack variables using the parameter τ you found in part (a). Will the resulting classifier necessarily obtain zero training error? Why or why not? A short explanation (without proof) will suffice.

Answer: The classifier will obtain zero training error. The SVM without slack variables will always return zero training error if it is able to find a solution, so all that remains to be shown is that there exists at least one feasible point.

Consider the constraint $y^{(i)}(w^T x^{(i)} + b)$ for some i, and let $b = 0$. Then

$$y^{(i)}(w^T x^{(i)} + b) = y^{(i)} \cdot f(x^{(i)}) > 0$$

since $f(x^{(i)})$ and $y^{(i)}$ have the same sign, and shown above. Therefore, as we choose all the α_i's large enough, $y^{(i)}(w^T x^{(i)} + b) > 1$, so the optimization problem is feasible.

(c) Suppose we run the SMO algorithm to train an SVM with slack variables, under the conditions stated above, using the value of τ you picked in the previous part, and using some arbitrary value of C (which you do not know beforehand). Will this necessarily result in a classifier that achieve zero training error? Why or why not? Again, a short explanation is sufficient.

Answer: The resulting classifier will not necessarily obtain zero training error. The C parameter controls the relative weights of the $(C\sum_{i=1}^{m}\xi_i)$ and $(\frac{1}{2}||w||^2)$ terms of the SVM training objective. If the C parameter is sufficiently small, then the former component will have relatively little contribution to the objective. In this case, a weight vector which has a very small norm but does not achieve zero training error may achieve a lower objective value than one which achieves zero training error. For example, you can consider the extreme case where $C = 0$, and the objective is just the norm of w. In this case, $w = 0$ is the solution to the optimization problem regardless of the choise of τ, this this may not obtain zero training error.

4. **Naive Bayes and SVMs for Spam Classification**

In this question you'll look into the Naive Bayes and Support Vector Machine algorithms for a spam classification problem. However, instead of implementing the algorithms yourself, you'll use a freely available machine learning library. There are many such libraries available, with different strengths and weaknesses, but for this problem you'll use the WEKA machine learning package, available at `http://www.cs.waikato.ac.nz/ml/weka/`. WEKA implements many standard machine learning algorithms, is written in Java, and has both a GUI and a command line interface. It is not the best library for very large-scale data sets, but it is very nice for playing around with many different algorithms on medium size problems.

You can download and install WEKA by following the instructions given on the website above. To use it from the command line, you first need to install a java runtime environment, then add the `weka.jar` file to your `CLASSPATH` environment variable. Finally, you can call WEKA using the command:

```
java <classifier> -t <training file> -T <test file>
```

For example, to run the Naive Bayes classifier (using the multinomial event model) on our provided spam data set by running the command:

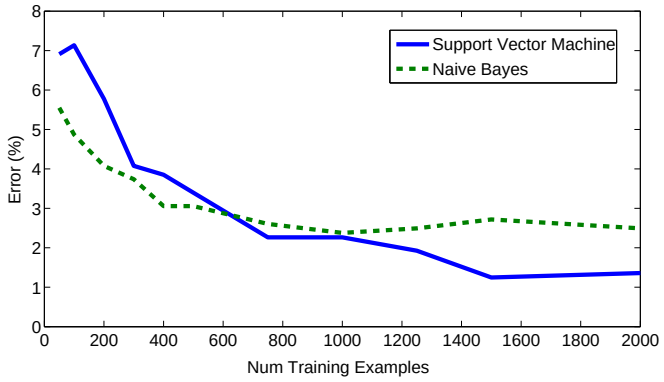
```
java weka.classifiers.bayes.NaiveBayesMultinomial -t spam_train_1000.arff -T spam_test.arff
```

The spam classification dataset in the `q4/` directory was provided courtesy of Christian Shelton (cshelton@cs.ucr.edu). Each example corresponds to a particular email, and each feature correspondes to a particular word. For privacy reasons we have removed the actual words themselves from the data set, and instead label the features generically as `f1`, `f2`, etc. However, the data set is from a real spam classification task, so the results demonstrate the performance of these algorithms on a real-world problem. The `q4/` directory actually contains several different training files, named `spam_train_50.arff`, `spam_train_100.arff`, etc (the ".arff" format is the default format by WEKA), each containing the corresponding number of training examples. There is also a single test set `spam_test.arff`, which is a hold out set used for evaluating the classifier's performance.

(a) Run the `weka.classifiers.bayes.NaiveBayesMultinomial` classifier on the dataset and report the resulting error rates. Evaluate the performance of the classifier using each of the different training files (but each time using the same test file, `spam_test.arff`). Plot the error rate of the classifier versus the number of training examples.

(b) Repeat the previous part, but using the `weka.classifiers.functions.SMO` classifier, which implements the SMO algorithm to train an SVM. How does the performance of the SVM compare to that of Naive Bayes?

Answer: Using the above command line arguments to run the classifier, we obtain the following error rates for the two algorithms:



For small amounts of data, Naive Bayes performs better than the Support Vector Machine. However, as the amount of data grows, the SVM achieves a better error rate.

5. **Uniform convergence**

In class we proved that for any finite set of hypotheses $\mathcal{H} = \{h_1, \ldots, h_k\}$, if we pick the hypothesis $\hat{h}$ that minimizes the training error on a set of m examples, then with probability at least $(1 - \delta)$,

$$\varepsilon(\hat{h}) \leq \left(\min_i \varepsilon(h_i)\right) + 2\sqrt{\frac{1}{2m} \log \frac{2k}{\delta}},$$

where $\varepsilon(h_i)$ is the generalization error of hypothesis h_i. Now consider a special case (often called the *realizable* case) where we know, a priori, that there is some hypothesis in our class $\mathcal{H}$ that achieves zero error on the distribution from which the data is drawn. Then we could obviously just use the above bound with $\min_i \varepsilon(h_i) = 0$; however, we can prove a better bound than this.

(a) Consider a learning algorithm which, after looking at m training examples, chooses some hypothesis $\hat{h} \in \mathcal{H}$ that makes zero mistakes on this training data. (By our assumption, there is at least one such hypothesis, possibly more.) Show that with probability $1 - \delta$

$$\varepsilon(\hat{h}) \leq \frac{1}{m} \log \frac{k}{\delta}.$$

Notice that since we do not have a square root here, this bound is much tighter. [Hint: Consider the probability that a hypothesis with generalization error greater than γ makes no mistakes on the training data. Instead of the Hoeffding bound, you might also find the following inequality useful: $(1 - \gamma)^m \leq e^{-\gamma m}$.]

Answer: Let $h \in \mathcal{H}$ be a hypothesis with true error greater than γ. Then

$$P(\text{“}h \text{ predicts correctly”}) \leq 1 - \gamma,$$

so

$$P(\text{“}h \text{ predicts correctly } m \text{ times”}) \leq (1-\gamma)^m \leq e^{-\gamma m}.$$

Applying the union bound,

$$P(\exists h \in \mathcal{H}, \text{ s.t. } \varepsilon(h) > \gamma \text{ and “h predicts correct } m \text{ times”}) \leq ke^{-\gamma m}.$$

We want to make this probability equal to δ, so we set

$$ke^{-\gamma m} = \delta,$$

which gives us

$$\gamma = \frac{1}{m} \log \frac{k}{\delta}.$$

This impiles that with probability $1-\delta$,

$$\varepsilon(\hat{h}) \leq \frac{1}{m} \log \frac{k}{\delta}.$$

(b) Rewrite the above bound as a sample complexity bound, i.e., in the form: for fixed δ and γ, for $\varepsilon(\hat{h}) \leq \gamma$ to hold with probability at least $(1-\delta)$, it suffices that $m \geq f(k, \gamma, \delta)$ (i.e., $f(\cdot)$ is some function of k, γ, and δ).

Answer: From part (a), if we take the equation,

$$ke^{-\gamma m} = \delta$$

and solve for m, we obtain

$$m = \frac{1}{\gamma} \log \frac{k}{\delta}.$$

Therefore, for m larger than this, $\varepsilon(\hat{h}) \leq \gamma$ will hold with probability at least $1-\delta$.

CS 229, Public Course
Problem Set #3: Learning Theory and Unsupervised Learning

---

1. **Uniform convergence and Model Selection**

   In this problem, we will prove a bound on the error of a simple model selection procedure.

   Let there be a binary classification problem with labels $y \in \{0, 1\}$, and let $\mathcal{H}_1 \subseteq \mathcal{H}_2 \subseteq \ldots \subseteq \mathcal{H}_k$ be k different finite hypothesis classes ($|\mathcal{H}_i| < \infty$). Given a dataset S of m iid training examples, we will divide it into a training set S_{train} consisting of the first $(1-\beta)m$ examples, and a hold-out cross validation set S_{cv} consisting of the remaining βm examples. Here, $\beta \in (0, 1)$.

   Let $\hat{h}_i = \arg\min_{h \in \mathcal{H}_i} \hat{\varepsilon}_{S_{\text{train}}}(h)$ be the hypothesis in $\mathcal{H}_i$ with the lowest *training* error (on S_{train}). Thus, $\hat{h}_i$ would be the hypothesis returned by training (with empirical risk minimization) using hypothesis class $\mathcal{H}_i$ and dataset S_{train}. Also let $h_i^\star = \arg\min_{h \in \mathcal{H}_i} \varepsilon(h)$ be the hypothesis in $\mathcal{H}_i$ with the lowest *generalization* error.

   Suppose that our algorithm first finds all the $\hat{h}_i$'s using empirical risk minimization then uses the hold-out cross validation set to select a hypothesis from this the $\{\hat{h}_1, \ldots, \hat{h}_k\}$ with minimum training error. That is, the algorithm will output

   $$\hat{h} = \arg \min_{h \in \{\hat{h}_1, \ldots, \hat{h}_k\}} \hat{\varepsilon}_{S_{\text{cv}}}(h).$$

   For this question you will prove the following bound. Let any $\delta > 0$ be fixed. Then with probability at least $1 - \delta$, we have that

   $$\varepsilon(\hat{h}) \leq \min_{i=1,\ldots,k} \left(\varepsilon(h_i^*) + \sqrt{\frac{2}{(1-\beta)m} \log \frac{4|\mathcal{H}_i|}{\delta}} \right) + \sqrt{\frac{2}{2\beta m} \log \frac{4k}{\delta}}$$

   (a) Prove that with probability at least $1 - \frac{\delta}{2}$, for all $\hat{h}_i$,

   $$|\varepsilon(\hat{h}_i) - \hat{\varepsilon}_{S_{\text{cv}}}(\hat{h}_i)| \leq \sqrt{\frac{1}{2\beta m} \log \frac{4k}{\delta}}.$$

   (b) Use part (a) to show that with probability $1 - \frac{\delta}{2}$,

   $$\varepsilon(\hat{h}) \leq \min_{i=1,\ldots,k} \varepsilon(\hat{h}_i) + \sqrt{\frac{2}{\beta m} \log \frac{4k}{\delta}}.$$

   (c) Let $j = \arg\min_i \varepsilon(\hat{h}_i)$. We know from class that for $\mathcal{H}_j$, with probability $1 - \frac{\delta}{2}$

   $$|\varepsilon(\hat{h}_j) - \hat{\varepsilon}_{S_{\text{train}}}(h_j^\star)| \leq \sqrt{\frac{2}{(1-\beta)m} \log \frac{4|\mathcal{H}_j|}{\delta}}, \quad \forall h_j \in \mathcal{H}_j.$$

   Use this to prove the final bound given at the beginning of this problem.

2. **VC Dimension**

Let the input domain of a learning problem be $\mathcal{X} = \mathbb{R}$. Give the VC dimension for each of the following classes of hypotheses. In each case, if you claim that the VC dimension is d, then you need to show that the hypothesis class can shatter d points, and explain why there are no $d+1$ points it can shatter.

- $h(x) = \mathbf{1}\{a < x\}$, with parameter $a \in \mathbb{R}$.
- $h(x) = \mathbf{1}\{a < x < b\}$, with parameters $a, b \in \mathbb{R}$.
- $h(x) = \mathbf{1}\{a \sin x > 0\}$, with parameter $a \in \mathbb{R}$.
- $h(x) = \mathbf{1}\{\sin(x + a) > 0\}$, with parameter $a \in \mathbb{R}$.

3. ℓ_1 **regularization for least squares**

In the previous problem set, we looked at the least squares problem where the objective function is augmented with an additional regularization term $\lambda\|\theta\|_2^2$. In this problem we'll consider a similar regularized objective but this time with a penalty on the ℓ_1 norm of the parameters $\lambda\|\theta\|_1$, where $\|\theta\|_1$ is defined as $\sum_i |\theta_i|$. That is, we want to minimize the objective

$$J(\theta) = \frac{1}{2}\sum_{i=1}^{m}(\theta^T x^{(i)} - y^{(i)})^2 + \lambda\sum_{i=1}^{n}|\theta_i|.$$

There has been a great deal of recent interest in ℓ_1 regularization, which, as we will see, has the benefit of outputting sparse solutions (i.e., many components of the resulting θ are equal to zero).

The ℓ_1 regularized least squares problem is more difficult than the unregularized or ℓ_2 regularized case, because the ℓ_1 term is not differentiable. However, there have been many efficient algorithms developed for this problem that work very well in practive. One very straightforward approach, which we have already seen in class, is the coordinate descent method. In this problem you'll derive and implement a coordinate descent algorithm for ℓ_1 regularized least squares, and apply it to test data.

(a) Here we'll derive the coordinate descent update for a given θ_i. Given the X and $\vec{y}$ matrices, as defined in the class notes, as well a parameter vector θ, how can we adjust θ_i so as to minimize the optimization objective? To answer this question, we'll rewrite the optimization objective above as

$$J(\theta) = \frac{1}{2}\|X\theta - \vec{y}\|_2^2 + \lambda\|\theta\|_1 = \frac{1}{2}\|X\bar{\theta} + X_i\theta_i - \vec{y}\|_2^2 + \lambda\|\bar{\theta}\|_1 + \lambda|\theta_i|$$

where $X_i \in \mathbb{R}^m$ denotes the ith column of X, and $\bar{\theta}$ is equal to θ except with $\bar{\theta}_i = 0$; all we have done in rewriting the above expression is to make the θ_i term explicit in the objective. However, this still contains the $|\theta_i|$ term, which is non-differentiable and therefore difficult to optimize. To get around this we make the observation that the sign of θ_i must either be non-negative or non-positive. But if we *knew* the sign of θ_i, then $|\theta_i|$ becomes just a linear term. That, is, we can rewrite the objective as

$$J(\theta) = \frac{1}{2}\|X\bar{\theta} + X_i\theta_i - \vec{y}\|_2^2 + \lambda\|\bar{\theta}\|_1 + \lambda s_i\theta_i$$

where s_i denotes the sign of θ_i, $s_i \in \{-1, 1\}$. In order to update θ_i, we can just compute the optimal θ_i for both possible values of s_i (making sure that we restrict

the optimal θ_i to obey the sign restriction we used to solve for it), then look to see which achieves the best objective value.

For each of the possible values of s_i, compute the resulting optimal value of θ_i. [Hint: to do this, you can fix s_i in the above equation, then differentiate with respect to θ_i to find the best value. Finally, clip θ_i so that it lies in the allowable range — i.e., for $s_i = 1$, you need to clip θ_i such that $\theta_i \geq 0$.]

(b) Implement the above coordinate descent algorithm using the updates you found in the previous part. We have provided a skeleton `theta = l1ls(X,y,lambda)` function in the `q3/` directory. To implement the coordinate descent algorithm, you should repeatedly iterate over all the θ_i's, adjusting each as you found above. You can terminate the process when θ changes by less than 10^-5 after all n of the updates.

(c) Test your implementation on the data provided in the `q3/` directory. The `[X, y, theta_true] = load_data;` function will load all the data — the data was generated by `y = X*theta_true + 0.05*randn(20,1)`, but `theta_true` is sparse, so that very few of the columns of `X` actually contain relevant features. Run your `l1ls.m` implementation on this data set, ranging λ from 0.001 to 10. Comment briefly on how this algorithm might be used for feature selection.

4. **K-Means Clustering**

In this problem you'll implement the K-means clustering algorithm on a synthetic data set. There is code and data for this problem in the `q4/` directory. Run `load 'X.dat';` to load the data file for clustering. Implement the `[clusters, centers] = k_means(X, k)` function in this directory. As input, this function takes the $m \times n$ data matrix `X` and the number of clusters `k`. It should output a m element vector, `clusters`, which indicates which of the clusters each data point belongs to, and a $k \times n$ matrix, `centers`, which contains the centroids of each cluster. Run the algorithm on the data provided, with $k = 3$ and $k = 4$. Plot the cluster assignments and centroids for each iteration of the algorithm using the `draw_clusters(X, clusters, centroids)` function. For each k, be sure to run the algorithm several times using different initial centroids.

5. **The Generalized EM algorithm**

When attempting to run the EM algorithm, it may sometimes be difficult to perform the M step exactly — recall that we often need to implement numerical optimization to perform the maximization, which can be costly. Therefore, instead of finding the global maximum of our lower bound on the log-likelihood, and alternative is to just increase this lower bound a little bit, by taking one step of gradient ascent, for example. This is commonly known as the Generalized EM (GEM) algorithm.

Put slightly more formally, recall that the M-step of the standard EM algorithm performs the maximization

$$\theta := \arg\max_\theta \sum_i \sum_{z^{(i)}} Q_i(z^{(i)}) \log \frac{p(x^{(i)}, z^{(i)}; \theta)}{Q_i(z^{(i)})}.$$

The GEM algorithm, in constrast, performs the following update in the M-step:

$$\theta := \theta + \alpha \nabla_\theta \sum_i \sum_{z^{(i)}} Q_i(z^{(i)}) \log \frac{p(x^{(i)}, z^{(i)}; \theta)}{Q_i(z^{(i)})}$$

where α is a learning rate which we assume is choosen small enough such that we do not decrease the objective function when taking this gradient step.

(a) Prove that the GEM algorithm described above converges. To do this, you should show that the the likelihood is monotonically improving, as it does for the EM algorithm — i.e., show that $\ell(\theta^{(t+1)}) \geq \ell(\theta^{(t)})$.

(b) Instead of using the EM algorithm at all, suppose we just want to apply gradient ascent to maximize the log-likelihood directly. In other words, we are trying to maximize the (non-convex) function

$$\ell(\theta) = \sum_i \log \sum_{z^{(i)}} p(x^{(i)}, z^{(i)}; \theta)$$

so we could simply use the update

$$\theta := \theta + \alpha \nabla_\theta \sum_i \log \sum_{z^{(i)}} p(x^{(i)}, z^{(i)}; \theta).$$

Show that this procedure in fact gives the same update as the GEM algorithm described above.

CS 229, Public Course
Problem Set #3 Solutions: Learning Theory and Unsupervised Learning

---

1. **Uniform convergence and Model Selection**

   In this problem, we will prove a bound on the error of a simple model selection procedure.

   Let there be a binary classification problem with labels $y \in \{0, 1\}$, and let $\mathcal{H}_1 \subseteq \mathcal{H}_2 \subseteq \ldots \subseteq \mathcal{H}_k$ be k different finite hypothesis classes ($|\mathcal{H}_i| < \infty$). Given a dataset S of m iid training examples, we will divide it into a training set S_{train} consisting of the first $(1-\beta)m$ examples, and a hold-out cross validation set S_{cv} consisting of the remaining βm examples. Here, $\beta \in (0, 1)$.

   Let $\hat{h}_i = \arg\min_{h \in \mathcal{H}_i} \hat{\varepsilon}_{S_{\text{train}}}(h)$ be the hypothesis in $\mathcal{H}_i$ with the lowest *training* error (on S_{train}). Thus, $\hat{h}_i$ would be the hypothesis returned by training (with empirical risk minimization) using hypothesis class $\mathcal{H}_i$ and dataset S_{train}. Also let $h_i^\star = \arg\min_{h \in \mathcal{H}_i} \varepsilon(h)$ be the hypothesis in $\mathcal{H}_i$ with the lowest *generalization* error.

   Suppose that our algorithm first finds all the $\hat{h}_i$'s using empirical risk minimization then uses the hold-out cross validation set to select a hypothesis from this the $\{\hat{h}_1, \ldots, \hat{h}_k\}$ with minimum training error. That is, the algorithm will output

$$\hat{h} = \arg \min_{h \in \{\hat{h}_1, \ldots, \hat{h}_k\}} \hat{\varepsilon}_{S_{\text{cv}}}(h).$$

   For this question you will prove the following bound. Let any $\delta > 0$ be fixed. Then with probability at least $1 - \delta$, we have that

$$\varepsilon(\hat{h}) \leq \min_{i=1,\ldots,k} \left(\varepsilon(h_i^*) + \sqrt{\frac{2}{(1-\beta)m} \log \frac{4|\mathcal{H}_i|}{\delta}} \right) + \sqrt{\frac{2}{2\beta m} \log \frac{4k}{\delta}}$$

   (a) Prove that with probability at least $1 - \frac{\delta}{2}$, for all $\hat{h}_i$,

$$|\varepsilon(\hat{h}_i) - \hat{\varepsilon}_{S_{\text{cv}}}(\hat{h}_i)| \leq \sqrt{\frac{1}{2\beta m} \log \frac{4k}{\delta}}.$$

   Answer: For each $\hat{h}_i$, the empirical error on the cross-validation set, $\hat{\varepsilon}(\hat{h}_i)$ represents the average of βm random variables with mean $\varepsilon(\hat{h}_i)$, so by the Hoeffding inequality for any $\hat{h}_i$,

$$P(|\varepsilon(\hat{h}_i) - \hat{\varepsilon}_{S_{\text{cv}}}(\hat{h}_i)| \geq \gamma) \leq 2\exp(-2\gamma^2 \beta m).$$

   As in the class notes, to insure that this holds for all $\hat{h}_i$, we need to take the union over all k of the $\hat{h}_i$'s.

$$P(\exists i, s.t. |\varepsilon(\hat{h}_i) - \hat{\varepsilon}_{S_{\text{cv}}}(\hat{h}_i)| \geq \gamma) \leq 2k\exp(-2\gamma^2 \beta m).$$

Setting this term equal to $\delta/2$ and solving for γ yields

$$\gamma = \sqrt{\frac{1}{2\beta m} \log \frac{4k}{\delta}}$$

proving the desired bound.

(b) Use part (a) to show that with probability $1 - \frac{\delta}{2}$,

$$\varepsilon(\hat{h}) \leq \min_{i=1,\ldots,k} \varepsilon(\hat{h}_i) + \sqrt{\frac{2}{\beta m} \log \frac{4k}{\delta}}.$$

Answer: Let $j = \arg\min_i \varepsilon(\hat{h}_i)$. Using part (a), with probability at least $1 - \frac{\delta}{2}$

$$\begin{aligned}
\varepsilon(\hat{h}) &\leq \hat{\varepsilon}_{S_{\text{cv}}}(\hat{h}) + \sqrt{\frac{1}{2\beta m} \log \frac{4k}{\delta}} \\
&= \min_i \hat{\varepsilon}_{S_{\text{cv}}}(\hat{h}_i) + \sqrt{\frac{1}{2\beta m} \log \frac{4k}{\delta}} \\
&\leq \hat{\varepsilon}_{S_{\text{cv}}}(\hat{h}_j) + \sqrt{\frac{1}{2\beta m} \log \frac{4k}{\delta}} \\
&\leq \varepsilon(\hat{h}_j) + 2\sqrt{\frac{1}{2\beta m} \log \frac{4k}{\delta}} \\
&= \min_{i=1,\ldots,k} \varepsilon(\hat{h}_i) + \sqrt{\frac{2}{\beta m} \log \frac{4k}{\delta}}
\end{aligned}$$

(c) Let $j = \arg\min_i \varepsilon(\hat{h}_i)$. We know from class that for $\mathcal{H}_j$, with probability $1 - \frac{\delta}{2}$

$$|\varepsilon(\hat{h}_j) - \hat{\varepsilon}_{S_{\text{train}}}(h_j^\star)| \leq \sqrt{\frac{2}{(1-\beta)m} \log \frac{4|\mathcal{H}_j|}{\delta}}, \;\; \forall h_j \in \mathcal{H}_j.$$

Use this to prove the final bound given at the beginning of this problem.

Answer: The bounds in parts (a) and (c) both hold simultaneously with probability $(1 - \frac{\delta}{2})^2 = 1 - \delta + \frac{\delta^2}{4} > 1 - \delta$, so with probablity greater than $1 - \delta$,

$$\varepsilon(\hat{h}) \leq \varepsilon(h_j^\star) + 2\sqrt{\frac{1}{2(1-\gamma)m} \log \frac{2|\mathcal{H}_j|}{\frac{\delta}{2}}} + 2\sqrt{\frac{1}{2\gamma m} \log \frac{2k}{\frac{\delta}{2}}}$$

which is equivalent to the bound we want to show.

2. **VC Dimension**

Let the input domain of a learning problem be $\mathcal{X} = \mathbb{R}$. Give the VC dimension for each of the following classes of hypotheses. In each case, if you claim that the VC dimension is d, then you need to show that the hypothesis class can shatter d points, and explain why there are no $d + 1$ points it can shatter.

- $h(x) = \mathbf{1}\{a < x\}$, with parameter $a \in \mathbb{R}$.

  Answer: VC-dimension = 1.

  (a) It can shatter point $\{0\}$, by choosing a to be 2 and -2.

  (b) It cannot shatter any two points $\{x_1, x_2\}$, $x_1 < x_2$, because the labelling $x_1 = 1$ and $x_2 = 0$ cannot be realized.

- $h(x) = \mathbf{1}\{a < x < b\}$, with parameters $a, b \in \mathbb{R}$.

  Answer: VC-dimension = 2.

  (a) It can shatter points $\{0, 2\}$ by choosing (a, b) to be $(3, 5)$, $(-1, 1)$, $(1, 3)$, $(-1, 3)$.

  (b) It cannot shatter any three points $\{x_1, x_2, x_3\}$, $x_1 < x_2 < x_3$, because the labelling $x_1 = x_3 = 1$, $x_2 = 0$ cannot be realized.

- $h(x) = \mathbf{1}\{a \sin x > 0\}$, with parameter $a \in \mathbb{R}$.

  Answer: VC-dimension = 1. a controls the *amplitude* of the sine curve.

  (a) It can shatter point $\{\frac{\pi}{2}\}$ by choosing a to be 1 and -1.

  (b) It cannot shatter any two points $\{x_1, x_2\}$, since, the labellings of x_1 and x_2 will flip together. If $x_1 = x_2 = 1$ for some a, then we cannot achieve $x_1 \neq x_2$. If $x_1 \neq x_2$ for some a, then we cannot achieve $x_1 = x_2 = 1$ ($x_1 = x_2 = 0$ can be achieved by setting $a = 0$).

- $h(x) = \mathbf{1}\{\sin(x + a) > 0\}$, with parameter $a \in \mathbb{R}$.

  Answer: VC-dimension = 2. a controls the *phase* of the sine curve.

  (a) It can shatter points $\{\frac{\pi}{4}, \frac{3\pi}{4}\}$, by choosing a to be 0, $\frac{\pi}{2}$, π, and $\frac{3\pi}{2}$.

  (b) It cannot shatter any three points $\{x_1, x_2, x_3\}$. Since sine has a preiod of 2π, let's define $x'_i = x_i \mod 2\pi$. W.l.o.g., assume $x'_1 < x'_2 < x'_3$. If the labelling of $x_1 = x_2 = x_3 = 1$ can be realized, then the labelling of $x_1 = x_3 = 1$, $x_2 = 0$ will not be realizable. Notice the similarity to the second question.

3. **ℓ_1 regularization for least squares**

   In the previous problem set, we looked at the least squares problem where the objective function is augmented with an additional regularization term $\lambda\|\theta\|_2^2$. In this problem we'll consider a similar regularized objective but this time with a penalty on the ℓ_1 norm of the parameters $\lambda\|\theta\|_1$, where $\|\theta\|_1$ is defined as $\sum_i |\theta_i|$. That is, we want to minimize the objective

$$J(\theta) = \frac{1}{2}\sum_{i=1}^{m}(\theta^T x^{(i)} - y^{(i)})^2 + \lambda\sum_{i=1}^{n}|\theta_i|.$$

   There has been a great deal of recent interest in ℓ_1 regularization, which, as we will see, has the benefit of outputting sparse solutions (i.e., many components of the resulting θ are equal to zero).

   The ℓ_1 regularized least squares problem is more difficult than the unregularized or ℓ_2 regularized case, because the ℓ_1 term is not differentiable. However, there have been many efficient algorithms developed for this problem that work very well in practive. One very straightforward approach, which we have already seen in class, is the coordinate descent method. In this problem you'll derive and implement a coordinate descent algorithm for ℓ_1 regularized least squares, and apply it to test data.

(a) Here we'll derive the coordinate descent update for a given θ_i. Given the X and $\vec{y}$ matrices, as defined in the class notes, as well a parameter vector θ, how can we adjust θ_i so as to minimize the optimization objective? To answer this question, we'll rewrite the optimization objective above as

$$J(\theta) = \frac{1}{2}\|X\theta - \vec{y}\|_2^2 + \lambda\|\theta\|_1 = \frac{1}{2}\|X\bar{\theta} + X_i\theta_i - \vec{y}\|_2^2 + \lambda\|\bar{\theta}\|_1 + \lambda|\theta_i|$$

where $X_i \in \mathbb{R}^m$ denotes the ith column of X, and $\bar{\theta}$ is equal to θ except with $\bar{\theta}_i = 0$; all we have done in rewriting the above expression is to make the θ_i term explicit in the objective. However, this still contains the $|\theta_i|$ term, which is non-differentiable and therefore difficult to optimize. To get around this we make the observation that the sign of θ_i must either be non-negative or non-positive. But if we *knew* the sign of θ_i, then $|\theta_i|$ becomes just a linear term. That, is, we can rewrite the objective as

$$J(\theta) = \frac{1}{2}\|X\bar{\theta} + X_i\theta_i - \vec{y}\|_2^2 + \lambda\|\bar{\theta}\|_1 + \lambda s_i\theta_i$$

where s_i denotes the sign of θ_i, $s_i \in \{-1, 1\}$. In order to update θ_i, we can just compute the optimal θ_i for both possible values of s_i (making sure that we restrict the optimal θ_i to obey the sign restriction we used to solve for it), then look to see which achieves the best objective value.

For each of the possible values of s_i, compute the resulting optimal value of θ_i. [Hint: to do this, you can fix s_i in the above equation, then differentiate with respect to θ_i to find the best value. Finally, clip θ_i so that it lies in the allowable range — i.e., for $s_i = 1$, you need to clip θ_i such that $\theta_i \geq 0$.]

Answer: For $s_i = 1$,

$$\begin{aligned} J(\theta) &= \frac{1}{2}\mathrm{tr}(X\bar{\theta} + X_i\theta_i - \vec{y})^T(X\bar{\theta} + X_i\theta_i - \vec{y}) + \lambda\|\bar{\theta}\|_1 + \lambda\theta_i \\ &= \frac{1}{2}\left(X_i^T X_i\theta_i^2 + 2X_i^T(X\bar{\theta} - \vec{y})\theta_i + \|X\bar{\theta} - \vec{y}\|_2^2\right) + \lambda\|\bar{\theta}\|_1 + \lambda\theta_i, \end{aligned}$$

so

$$\frac{\partial J(\theta)}{\partial \theta_i} = X_i^T X_i\theta + X_i^T(X\bar{\theta} - \vec{y}) + \lambda$$

which means the optimal θ_i is given by

$$\theta_i = \max\left\{\frac{-X_i^T(X\bar{\theta} - \vec{y}) - \lambda}{X_i^T X_i}, 0\right\}.$$

Similarly, for $s_i = -1$, the optimal θ_i is given by

$$\theta_i = \min\left\{\frac{-X_i^T(X\bar{\theta} - \vec{y}) + \lambda}{X_i^T X_i}, 0\right\}.$$

(b) Implement the above coordinate descent algorithm using the updates you found in the previous part. We have provided a skeleton `theta = l1ls(X,y,lambda)` function in the `q3/` directory. To implement the coordinate descent algorithm, you should repeatedly iterate over all the θ_i's, adjusting each as you found above. You can terminate the process when θ changes by less than 10^-5 after all n of the updates.

Answer: The following is our implementation of `l1ls.m`:

```
function theta = l1l2(X,y,lambda)

m = size(X,1);
n = size(X,2);
theta = zeros(n,1);
old_theta = ones(n,1);

while (norm(theta - old_theta) > 1e-5)
  old_theta = theta;
  for i=1:n,
    % compute possible values for theta
    theta(i) = 0;
    theta_i(1) = max((-X(:,i)'*(X*theta - y) - lambda) / (X(:,i)'*X(:,i)), 0);
    theta_i(2) = min((-X(:,i)'*(X*theta - y) + lambda) / (X(:,i)'*X(:,i)), 0);

    % get objective value for both possible thetas
    theta(i) = theta_i(1);
    obj_theta(1) = 0.5*norm(X*theta - y)^2 + lambda*norm(theta,1);
    theta(i) = theta_i(2);
    obj_theta(2) = 0.5*norm(X*theta - y)^2 + lambda*norm(theta,1);

    % pick the theta which minimizes the objective
    [min_obj, min_ind] = min(obj_theta);
    theta(i) = theta_i(min_ind);
  end
end
```

(c) Test your implementation on the data provided in the `q3/` directory. The `[X, y, theta_true] = load_data;` function will load all the data — the data was generated by `y = X*theta_true + 0.05*randn(20,1)`, but `theta_true` is sparse, so that very few of the columns of `X` actually contain relevant features. Run your `l1ls.m` implementation on this data set, ranging λ from 0.001 to 10. Comment briefly on how this algorithm might be used for feature selection.

Answer: For $\lambda = 1$, our implementation of l_1 regularized least squares recovers the exact sparsity pattern of the true parameter that generated the data. In constrast, using any amount of l_2 regularization still leads to θ's that contain no zeros. This suggests that the l_1 regularization could be very useful as a feature selection algorithm: we could run l_1 regularized least squares to see which coefficients are non-zero, then select only these features for use with either least-squares or possibly a completely different machine learning algorithm.

4. **K-Means Clustering**

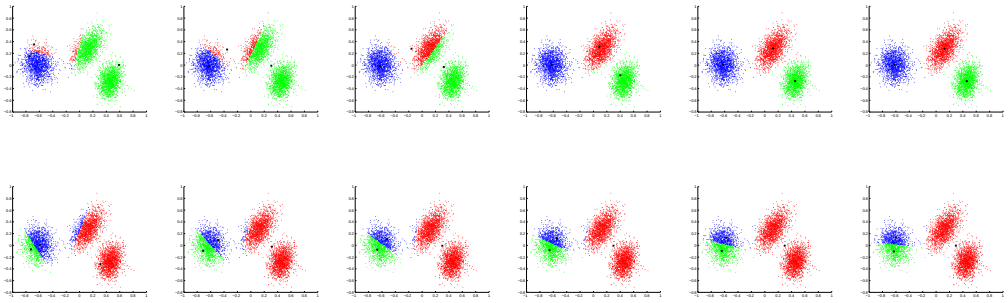
In this problem you'll implement the K-means clustering algorithm on a synthetic data set. There is code and data for this problem in the `q4/` directory. Run `load 'X.dat';` to load the data file for clustering. Implement the `[clusters, centers] = k_means(X, k)` function in this directory. As input, this function takes the $m \times n$ data matrix `X` and the number of clusters `k`. It should output a m element vector, `clusters`, which indicates which of the clusters each data point belongs to, and a $k \times n$ matrix, `centers`, which contains the centroids of each cluster. Run the algorithm on the data provided, with $k = 3$

and $k = 4$. Plot the cluster assignments and centroids for each iteration of the algorithm using the `draw_clusters(X, clusters, centroids)` function. For each k, be sure to run the algorithm several times using different initial centroids.

Answer: The following is our implementation of `k_means.m`:

```
function [clusters, centroids] = k_means(X, k)

m = size(X,1);
n = size(X,2);
oldcentroids = zeros(k,n);
centroids = X(ceil(rand(k,1)*m),:);

while (norm(oldcentroids - centroids) > 1e-15)
  oldcentroids = centroids;
  % compute cluster assignments
  for i=1:m,
    dists = sum((repmat(X(i,:), k, 1) - centroids).^2, 2);
    [min_dist, clusters(i,1)] = min(dists);
  end

  draw_clusters(X, clusters, centroids);
  pause(0.1);

  % compute cluster centroids
  for i=1:k,
    centroids(i,:) = mean(X(clusters == i, :));
  end
end
```

Below we show the centroid evolution for two typical runs with $k = 3$. Note that the different starting positions of the clusters lead to do different final clusterings.



5. **The Generalized EM algorithm**

When attempting to run the EM algorithm, it may sometimes be difficult to perform the M step exactly — recall that we often need to implement numerical optimization to perform the maximization, which can be costly. Therefore, instead of finding the global maximum of our lower bound on the log-likelihood, and alternative is to just increase this lower bound a little bit, by taking one step of gradient ascent, for example. This is commonly known as the Generalized EM (GEM) algorithm.

Put slightly more formally, recall that the M-step of the standard EM algorithm performs the maximization

$$\theta := \arg\max_\theta \sum_i \sum_{z^{(i)}} Q_i(z^{(i)}) \log \frac{p(x^{(i)}, z^{(i)}; \theta)}{Q_i(z^{(i)})}.$$

The GEM algorithm, in constrast, performs the following update in the M-step:

$$\theta := \theta + \alpha \nabla_\theta \sum_i \sum_{z^{(i)}} Q_i(z^{(i)}) \log \frac{p(x^{(i)}, z^{(i)}; \theta)}{Q_i(z^{(i)})}$$

where α is a learning rate which we assume is choosen small enough such that we do not decrease the objective function when taking this gradient step.

(a) Prove that the GEM algorithm described above converges. To do this, you should show that the the likelihood is monotonically improving, as it does for the EM algorithm — i.e., show that $\ell(\theta^{(t+1)}) \geq \ell(\theta^{(t)})$.

Answer: We use the same logic as for the standard EM algorithm. Specifically, just as for EM, we have for the GEM algorithm that

$$\begin{aligned} \ell(\theta^{(t+1)}) &\geq \sum_i \sum_{z^{(i)}} Q_i^{(t)}(z^{(i)}) \log \frac{p(x^{(i)}, z^{(i)}; \theta^{(t+1)})}{Q_i^{(t)}(z^{(i)})} \\ &\geq \sum_i \sum_{z^{(i)}} Q_i^{(t)}(z^{(i)}) \log \frac{p(x^{(i)}, z^{(i)}; \theta^{(t)})}{Q_i^{(t)}(z^{(i)})} \\ &= \ell(\theta^{(t)}) \end{aligned}$$

where as in EM the first line holds due to Jensen's equality, and the last line holds because we choose the Q distribution to make this hold with equality. The only difference between EM and GEM is the logic as to why the second line holds: for EM it held because $\theta^{(t+1)}$ was chosen to maximize this quantity, but for GEM it holds by our assumption that we take a gradient step small enough so as not to decrease the objective function.

(b) Instead of using the EM algorithm at all, suppose we just want to apply gradient ascent to maximize the log-likelihood directly. In other words, we are trying to maximize the (non-convex) function

$$\ell(\theta) = \sum_i \log \sum_{z^{(i)}} p(x^{(i)}, z^{(i)}; \theta)$$

so we could simply use the update

$$\theta := \theta + \alpha \nabla_\theta \sum_i \log \sum_{z^{(i)}} p(x^{(i)}, z^{(i)}; \theta).$$

Show that this procedure in fact gives the same update as the GEM algorithm described above.

Answer: Differentiating the log likelihood directly we get

$$\begin{aligned} \frac{\partial}{\partial \theta_j} \sum_i \log \sum_{z^{(i)}} p(x^{(i)}, z^{(i)}; \theta) &= \sum_i \frac{1}{\sum_{z^{(i)}} p(x^{(i)}, z^{(i)}; \theta)} \sum_{z^{(i)}} \frac{\partial}{\partial \theta_j} p(x^{(i)}, z^{(i)}; \theta) \\ &= \sum_i \sum_{z^{(i)}} \frac{1}{p(x^{(i)}; \theta)} \cdot \frac{\partial}{\partial \theta_j} p(x^{(i)}, z^{(i)}; \theta). \end{aligned}$$

For the GEM algorithm,

$$\frac{\partial}{\partial\theta_j}\sum_i\sum_{z^{(i)}} Q_i(z^{(i)})\log\frac{p(x^{(i)},z^{(i)};\theta)}{Q_i(z^{(i)})} \quad = \quad \sum_i\sum_{z^{(i)}}\frac{Q_i(z^{(i)})}{p(x^{(i)},z^{(i)};\theta)}\cdot\frac{\partial}{\partial\theta_j}p(x^{(i)},z^{(i)};\theta).$$

But the E-step of the GEM algorithm chooses

$$Q_i(z^{(i)}) = p(z^{(i)}|x^{(i)};\theta) = \frac{p(x^{(i)},z^{(i)};\theta)}{p(x^{(i)};\theta)},$$

so

$$\sum_i\sum_{z^{(i)}}\frac{Q_i(z^{(i)})}{p(x^{(i)},z^{(i)};\theta)}\cdot\frac{\partial}{\partial\theta_j}p(x^{(i)},z^{(i)};\theta) = \sum_i\sum_{z^{(i)}}\frac{1}{p(x^{(i)};\theta)}\cdot\frac{\partial}{\partial\theta_j}p(x^{(i)},z^{(i)};\theta)$$

which is the same as the derivative of the log likelihood.

CS 229, Public Course
Problem Set #4: Unsupervised Learning and Reinforcement Learning

---

1. **EM for supervised learning**

   In class we applied EM to the unsupervised learning setting. In particular, we represented $p(x)$ by marginalizing over a latent random variable

   $$p(x) = \sum_z p(x, z) = \sum_z p(x|z)p(z).$$

   However, EM can also be applied to the supervised learning setting, and in this problem we discuss a "mixture of linear regressors" model; this is an instance of what is often call the Hierarchical Mixture of Experts model. We want to represent $p(y|x)$, $x \in \mathbb{R}^n$ and $y \in \mathbb{R}$, and we do so by again introducing a discrete latent random variable

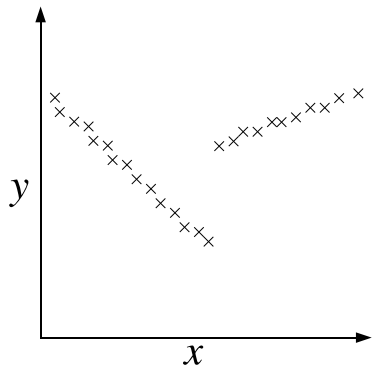
   $$p(y|x) = \sum_z p(y, z|x) = \sum_z p(y|x, z)p(z|x).$$

   For simplicity we'll assume that z is binary valued, that $p(y|x, z)$ is a Gaussian density, and that $p(z|x)$ is given by a logistic regression model. More formally

   $$\begin{aligned} p(z|x;\phi) &= g(\phi^T x)^z (1 - g(\phi^T x))^{1-z} \\ p(y|x, z = i;\theta_i) &= \frac{1}{\sqrt{2\pi}\sigma} \exp\left(\frac{-(y - \theta_i^T x)^2}{2\sigma^2}\right) \quad i = 1, 2 \end{aligned}$$

   where σ is a known parameter and $\phi, \theta_0, \theta_1 \in \mathbb{R}^n$ are parameters of the model (here we use the subscript on θ to denote two different parameter vectors, not to index a particular entry in these vectors).

   Intuitively, the process behind model can be thought of as follows. Given a data point x, we first determine whether the data point belongs to one of two hidden classes $z = 0$ or $z = 1$, using a logistic regression model. We then determine y as a linear function of x (different linear functions for different values of z) plus Gaussian noise, as in the standard linear regression model. For example, the following data set could be well-represented by the model, but not by standard linear regression.

(a) Suppose x, y, and z are all observed, so that we obtain a training set $\{(x^{(1)}, y^{(1)}, z^{(1)}), \ldots, (x^{(m)}, y^{(m)}, z^{(m)})\}$. Write the log-likelihood of the parameters, and derive the maximum likelihood estimates for ϕ, θ_0, and θ_1. Note that because $p(z|x)$ is a logistic regression model, there will not exist a closed form estimate of ϕ. In this case, derive the gradient and the Hessian of the likelihood with respect to ϕ; in practice, these quantities can be used to numerically compute the ML esimtate.

(b) Now suppose z is a latent (unobserved) random variable. Write the log-likelihood of the parameters, and derive an EM algorithm to maximize the log-likelihood. Clearly specify the E-step and M-step (again, the M-step will require a numerical solution, so find the appropriate gradients and Hessians).

2. **Factor Analysis and PCA**

In this problem we look at the relationship between two unsupervised learning algorithms we discussed in class: Factor Analysis and Principle Component Analysis.

Consider the following joint distribution over (x, z) where $z \in \mathbb{R}^k$ is a latent random variable

$$\begin{aligned} z &\sim \mathcal{N}(0, I) \\ x|z &\sim \mathcal{N}(Uz, \sigma^2 I). \end{aligned}$$

where $U \in \mathbb{R}^{n \times k}$ is a model parameters and σ^2 is assumed to be a known constant. This model is often called Probabilistic PCA. Note that this is nearly identical to the factor analysis model except we assume that the variance of $x|z$ is a known scaled identity matrix rather than the diagonal parameter matrix, Φ, and we do not add an additional μ term to the mean (though this last difference is just for simplicity of presentation). However, as we will see, it turns out that as $\sigma^2 \to 0$, this model is equivalent to PCA.

For simplicity, you can assume for the remainder of the problem that $k = 1$, i.e., that U is a column vector in $\mathbb{R}^n$.

(a) Use the rules for manipulating Gaussian distributions to determine the joint distribution over (x, z) and the conditional distribution of $z|x$. [Hint: for later parts of this problem, it will help significantly if you simplify your soluting for the conditional distribution using the identity we first mentioned in problem set #1: $(\lambda I + BA)^{-1}B = B(\lambda I + AB)^{-1}$.]

(b) Using these distributions, derive an EM algorithm for the model. Clearly state the E-step and the M-step of the algorithm.

(c) As $\sigma^2 \to 0$, show that if the EM algorithm convergences to a parameter vector $U^\star$ (and such convergence is guarenteed by the argument presented in class), then $U^\star$ must be an eigenvector of the sample covariance matrix $\Sigma = \frac{1}{m}\sum_{i=1}^m x^{(i)} {x^{(i)}}^T$ — i.e., $U^\star$ must satisfy

$$\lambda U^\star = \Sigma U^\star.$$

[Hint: When $\sigma^2 \to 0$, $\Sigma_{z|x} \to 0$, so the E step only needs to compute the means $\mu_{z|x}$ and not the variances. Let $w \in \mathbb{R}^m$ be a vector containing all these means, $w_i = \mu_{z^{(i)}|x^{(i)}}$, and show that the E step and M step can be expressed as

$$w = \frac{XU}{U^T U}, \quad U = \frac{X^T w}{w^T w}$$

respectively. Finally, show that if U doesn't change after this update, it must satisfy the eigenvector equation shown above.]

3. **PCA and ICA for Natural Images**

In this problem we'll apply Principal Component Analysis and Independent Component Analysis to images patches collected from "natural" image scenes (pictures of leaves, grass, etc). This is one of the classical applications of the ICA algorithm, and sparked a great deal of interest in the algorithm; it was observed that the bases recovered by ICA closely resemble image filters present in the first layer of the visual cortex.

The `q3/` directory contains the data and several useful pieces of code for this problem. The raw images are stored in the *images/* subdirectory, though you will not need to work with these directly, since we provide code for loading and normalizing the images.

Calling the function `[X_ica, X_pca] = load_images;` will load the images, break them into 16x16 images patches, and place all these patches into the columns of the matrices `X_ica` and `X_pca`. We create two different data sets for PCA and ICA because the algorithms require slightly different methods of preprocessing the data.[1]

For this problem you'll implement the `ica.m` and `pca.m` functions, using the PCA and ICA algorithms described in the class notes. While the PCA implementation should be straightforward, getting a good implementation of ICA can be a bit trickier. Here is some general advice to getting a good implementation on this data set:

- Picking a good learning rate is important. In our experiments we used $\alpha = 0.0005$ on this data set.
- Batch gradient descent doesn't work well for ICA (this has to do with the fact that ICA objective function is not concave), but the pure stochastic gradient described in the notes can be slow (There are about 20,000 16x16 images patches in the data set, so one pass over the data using the stochastic gradient rule described in the notes requires inverting the 256x256 W matrix 20,000 times). Instead, a good compromise is to use a hybrid stochastic/batch gradient descent where we calculate the gradient with respect to several examples at a time (100 worked well for us), and use this to update W. Our implementation makes 10 total passes over the entire data set.
- It is a good idea to randomize the order of the examples presented to stochastic gradient descent before each pass over the data.
- Vectorize your Matlab code as much as possible. For general examples of how to do this, look at the Matlab review session.

For reference, computing the ICA W matrix for the entire set of image patches takes about 5 minutes on a 1.6 Ghz laptop using our implementation.

After you've learned the U matrix for PCA (the columns of U should contain the principal components of the data) and the W matrix of ICA, you can plot the basis functions using the `plot_ica_bases(W);` and `plot_pca_bases(U);` functions we have provide. Comment briefly on the difference between the two sets of basis functions.

[1] Recall that the first step of performing PCA is to subtract the mean and normalize the variance of the features. For the image data we're using, the preprocessing step for the ICA algorithm is slightly different, though the precise mechanism and justification is not imporant for the sake of this problem. Those who are curious about the details should read Bell and Sejnowki's paper "The 'Independent Components' of Natural Scenes are Edge Filters," which provided the basis for the implementation we use in this problem.

4. **Convergence of Policy Iteration**

   In this problem we show that the Policy Iteration algorithm, described in the lecture notes, is guarenteed to find the optimal policy for an MDP. First, define B^π to be the Bellman operator for policy π, defined as follows: if $V' = B(V)$, then

   $$V'(s) = R(s) + \gamma \sum_{s' \in \mathcal{S}} P_{s\pi(s)}(s')V(s').$$

   (a) Prove that if $V_1(s) \leq V_2(s)$ for all $s \in \mathbb{S}$, then $B(V_1)(s) \leq B(V_2)(s)$ for all $s \in \mathbb{S}$.

   (b) Prove that for any V,

   $$\|B^\pi(V) - V^\pi\|_\infty \leq \gamma \|V - V^\pi\|_\infty$$

   where $\|V\|_\infty = \max_{s \in \mathcal{S}} |V(s)|$. Intuitively, this means that applying the Bellman operator B^π to any value function V, brings that value function "closer" to the value function for π, V^π. This also means that applying B^π repeatedly (an infinite number of times)

   $$B^\pi(B^\pi(\ldots B^\pi(V)\ldots))$$

   will result in the value function V^π (a little bit more is needed to make this completely formal, but we won't worry about that here).

   [Hint: Use the fact that for any $\alpha, x \in \mathbb{R}^n$, if $\sum_i \alpha_i = 1$ and $\alpha_i \geq 0$, then $\sum_i \alpha_i x_i \leq \max_i x_i$.]

   (c) Now suppose that we have some policy π, and use Policy Iteration to choose a new policy π' according to

   $$\pi'(s) = \arg\max_{a \in \mathcal{A}} \sum_{s' \in \mathcal{S}} P_{sa}(s')V^\pi(s').$$

   Show that this policy will never perform worse that the previous one — i.e., show that for all $s \in \mathcal{S}$, $V^\pi(s) \leq V^{\pi'}(s)$.

   [Hint: First show that $V^\pi(s) \leq B^{\pi'}(V^\pi)(s)$, then use the proceeding excercises to show that $B^{\pi'}(V^\pi)(s) \leq V^{\pi'}(s)$.]

   (d) Use the proceeding exercises to show that policy iteration will eventually converge (i.e., produce a policy $\pi' = \pi$). Furthermore, show that it must converge to the optimal policy $\pi^\star$. For the later part, you may use the property that if some value function satisfies

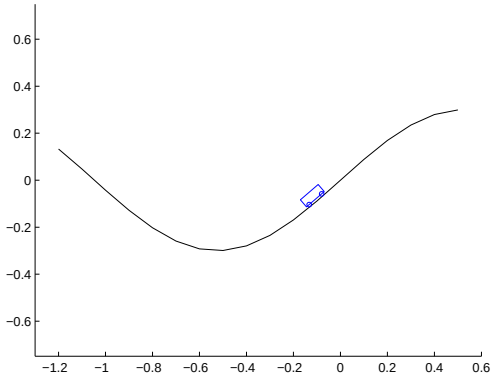
   $$V(s) = R(s) + \gamma \max_{a \in \mathcal{A}} \sum s' \in \mathcal{S} P_{sa}(s')V(s')$$

   then $V = V^\star$.

5. **Reinforcement Learning: The Mountain Car**

   In this problem you will implement the Q-Learning reinforcement learning algorithm described in class on a standard control domain known as the Mountain Car.[2] The Mountain Car domain simulates a car trying to drive up a hill, as shown in the figure below.

[2]The dynamics of this domain were taken from Sutton and Barto, 1998.



All states except those at the top of the hill have a constant reward $R(s) = -1$, while the goal state at the hilltop has reward $R(s) = 0$; thus an optimal agent will try to get to the top of the hill as fast as possible (when the car reaches the top of the hill, the episode is over, and the car is reset to its initial position). However, when starting at the bottom of the hill, the car does not have enough power to reach the top by driving forward, so it must first accerlaterate accelerate backwards, building up enough momentum to reach the top of the hill. This strategy of moving away from the goal in order to reach the goal makes the problem difficult for many classical control algorithms.

As discussed in class, Q-learning maintains a table of Q-values, $Q(s, a)$, for each state and action. These Q-values are useful because, in order to select an action in state s, we only need to check to see which Q-value is greatest. That is, in state s we take the action

$$\arg\max_{a \in \mathcal{A}} Q(s, a).$$

The Q-learning algorithm adjusts its estimates of the Q-values as follows. If an agent is in state s, takes action a, then ends up in state s', Q-learning will update $Q(s, a)$ by

$$Q(s, a) = (1 - \alpha)Q(s, a) + \gamma(R(s') + \gamma \max_{a' \in \mathcal{A}} Q(s', a').$$

At each time, your implementation of Q-learning can execute the greedy policy $\pi(s) = \arg\max_{a \in \mathcal{A}} Q(s, a)$

Implement the `[q, steps_per_episode] = qlearning(episodes)` function in the `q5/` directory. As input, the function takes the total number of episodes (each episode starts with the car at the bottom of the hill, and lasts until the car reaches the top), and outputs a matrix of the Q-values and a vector indicating how many steps it took before the car was able to reach the top of the hill. You should use the `[x, s, absorb] = mountain_car(x, actions(a))` function to simulate one control cycle for the task — the `x` variable describes the true (continuous) state of the system, whereas the `s` variable describes the discrete index of the state, which you'll use to build the Q values.

Plot a graph showing the average number of steps before the car reaches the top of the hill versus the episode number (there is quite a bit of variation in this quantity, so you will probably want to average these over a large number of episodes, as this will give you a better idea of how the number of steps before reaching the hilltop is decreasing). You can also visualize your resulting controller by calling the `draw_mountain_car(q)` function.

CS 229, Public Course
Problem Set #4 Solutions: Unsupervised Learning and Reinforcement Learning

---

1. **EM for supervised learning**

   In class we applied EM to the unsupervised learning setting. In particular, we represented $p(x)$ by marginalizing over a latent random variable

   $$p(x) = \sum_z p(x, z) = \sum_z p(x|z)p(z).$$

   However, EM can also be applied to the supervised learning setting, and in this problem we discuss a "mixture of linear regressors" model; this is an instance of what is often call the Hierarchical Mixture of Experts model. We want to represent $p(y|x)$, $x \in \mathbb{R}^n$ and $y \in \mathbb{R}$, and we do so by again introducing a discrete latent random variable

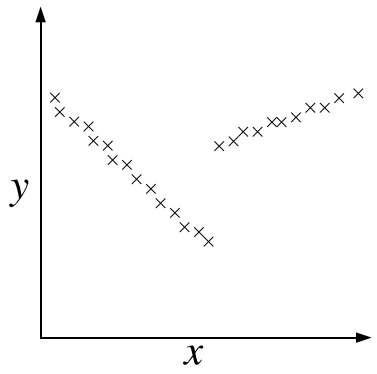
   $$p(y|x) = \sum_z p(y, z|x) = \sum_z p(y|x, z)p(z|x).$$

   For simplicity we'll assume that z is binary valued, that $p(y|x, z)$ is a Gaussian density, and that $p(z|x)$ is given by a logistic regression model. More formally

   $$\begin{aligned} p(z|x;\phi) &= g(\phi^T x)^z (1 - g(\phi^T x))^{1-z} \\ p(y|x, z = i;\theta_i) &= \frac{1}{\sqrt{2\pi}\sigma} \exp\left(\frac{-(y - \theta_i^T x)^2}{2\sigma^2}\right) \quad i = 1, 2 \end{aligned}$$

   where σ is a known parameter and $\phi, \theta_0, \theta_1 \in \mathbb{R}^n$ are parameters of the model (here we use the subscript on θ to denote two different parameter vectors, not to index a particular entry in these vectors).

   Intuitively, the process behind model can be thought of as follows. Given a data point x, we first determine whether the data point belongs to one of two hidden classes $z = 0$ or $z = 1$, using a logistic regression model. We then determine y as a linear function of x (different linear functions for different values of z) plus Gaussian noise, as in the standard linear regression model. For example, the following data set could be well-represented by the model, but not by standard linear regression.

(a) Suppose x, y, and z are all observed, so that we obtain a training set $\{(x^{(1)}, y^{(1)}, z^{(1)}), \ldots, (x^{(m)}, y^{(m)}, z^{(m)})\}$. Write the log-likelihood of the parameters, and derive the maximum likelihood estimates for ϕ, θ_0, and θ_1. Note that because $p(z|x)$ is a logistic regression model, there will not exist a closed form estimate of ϕ. In this case, derive the gradient and the Hessian of the likelihood with respect to ϕ; in practice, these quantities can be used to numerically compute the ML esimtate.

Answer: The log-likelihood is given by

$$\begin{aligned}
\ell(\phi, \theta_0, \theta 1) &= \log \prod_{i=1}^{m} p(y^{(i)}|x^{(i)}, z^{(i)}; \theta_0, \theta_1) p(z^{(i)}|x^{(i)}; \phi) \\
&= \sum_{i:z^{(i)}=0} \log \left((1 - g(\phi^T x)) \frac{1}{\sqrt{2\pi}\sigma} \exp\left(\frac{-(y^{(i)} - \theta_0^T x^{(i)})^2}{2\sigma^2} \right) \right) \\
&\quad + \sum_{i:z^{(i)}=1} \log \left((g(\phi^T x) \frac{1}{\sqrt{2\pi}\sigma} \exp\left(\frac{-(y^{(i)} - \theta_1^T x^{(i)})^2}{2\sigma^2} \right) \right)
\end{aligned}$$

Differentiating with respect to θ_1 and setting it to 0,

$$\begin{aligned}
0 &\stackrel{\text{set}}{=} \nabla_{\theta_0} \ell(\phi, \theta_0, \theta_1) \\
&= \nabla_\theta \sum_{i:z^{(i)}=0} -(y^{(i)} - \theta_0^T x^{(i)})^2
\end{aligned}$$

But this is just a least-squares problem on a subset of the data. In particular, if we let X_0 and $\vec{y}_0$ be the design matrices formed by considering only those examples with $z^{(i)} = 0$, then using the same logic as for the derivation of the least squares solution we get the maximum likelihood estimate of θ_0,

$$\theta_0 = (X_0^T X_0)^{-1} X_0^T \vec{y}_0.$$

The derivation for θ_1 proceeds in the identical manner.

Differentiating with respect to ϕ, and ignoring terms that do not depend on ϕ

$$\begin{aligned}
\nabla_\phi \ell(\phi, \theta_0, \theta_1) &= \nabla_\phi \sum i : z^{(i)} = 0 \log(1 - g(\phi^T x)) + \sum i : z^{(i)} = 1 \log g(\phi^T x) \\
&= \nabla_\phi \sum_{i=1}^{m} (1 - z^{(i)}) \log(1 - g(\phi^T x)) + z^{(i)} \log g(\phi^T x)
\end{aligned}$$

This is just the standard logistic regression objective function, for which we already know the gradient and Hessian

$$\nabla_\phi \ell(\phi, \theta_0, \theta_1) = X^T(\vec{z} - \vec{h}), \quad \vec{h}_i = g(\phi^T x^{(i)}),$$

$$H = X^T D X, \quad D_{ii} = g(\phi^T x^{(i)})(1 - g(\phi^T x^{(i)})).$$

(b) Now suppose z is a latent (unobserved) random variable. Write the log-likelihood of the parameters, and derive an EM algorithm to maximize the log-likelihood. Clearly specify the E-step and M-step (again, the M-step will require a numerical solution, so find the appropriate gradients and Hessians).

Answer: The log likelihood is now:

$$\begin{aligned}\ell(\phi,\theta_0,\theta_1) &= \log\prod_{i=1}^{m}\sum_{z^{(i)}} p(y^{(i)}|x^{(i)},z^{(i)};\theta_1,\theta_2)p(z^{(i)}|x^{(i)};\phi)\\ &= \sum_{i=1}^{m}\log\left((1-g(\phi^T x^{(i)}))^{1-z^{(i)}}\frac{1}{\sqrt{2\pi}\sigma}\exp\left(\frac{-(y^{(i)}-\theta_0^T x^{(i)})^2}{2\sigma^2}\right)\right.\\ &\qquad\left. + g(\phi^T x^{(i)})^{z^{(i)}}\frac{1}{\sqrt{2\pi}\sigma}\exp\left(\frac{-(y^{(i)}-\theta_1^T x^{(i)})^2}{2\sigma^2}\right)\right)\end{aligned}$$

In the E-step of the EM algorithm we compute

$$Q_i(z^{(i)}) = p(z^{(i)}|x^{(i)},y^{(i)};\phi,\theta_0,\theta_1) = \frac{p(y^{(i)}|x^{(i)},z^{(i)};\theta_0,\theta_1)p(z^{(i)}|x^{(i)};\phi)}{\sum_z p(y^{(i)}|x^{(i)},z;\theta_0,\theta_1)p(z|x^{(i)};\phi)}$$

Every propability in this term can be computed using the probability densities defined in the problem, so the E-step is tractable.

For the M-step, we first define $w_j^{(i)} = p(z^{(i)}=j|x^{(i)},y^{(i)};\phi,\theta_0,\theta_1)$ for $j=0,1$ as computed in the E-step (of course we only need to compute one of these terms in the real E-step, since $w_0^{(i)} = 1-w_1^{(i)}$, but we define both to simplify the expressions). Differentiating our lower bound on the likelihood with respect to θ_0, removing terms that don't depend on θ_0, and setting the expression equal to zero, we get

$$\begin{aligned}0 &\overset{\text{set}}{=} \nabla_{\theta_0}\sum_{i=1}^{m}\sum_{j=0,1} w_j^{(i)}\log\frac{p(y^{(i)}|x^{(i)},z^{(i)}=j;\theta_j)p(z^{(i)}=j|x^{(i)};\phi)}{w_j^{(i)}}\\ &= \nabla_{\theta_0}\sum_{i=1}^{m} w_0^{(i)}\log p(y^{(i)}|x^{(i)},z^{(i)}=j;\theta_j)\\ &= \nabla_{\theta_0}\sum_{i=1}^{m} -w_0^{(i)}(y^{(i)}-\theta_0^T x^{(i)})^2\end{aligned}$$

This is just a weighted least-squares problem, which has solution

$$\theta_0 = (X_0^T W X_0)^{-1}X_0^T W\vec{y}_0, \qquad W = \text{diag}(w_0^{(1)},\ldots,w_0^{(m)}.$$

The derivation for θ_1 proceeds similarly.

Finally, as before, we can't compute the M-step update for ϕ in closed form, so we instead find the gradient and Hessian. However, to do this we note that

$$\nabla_\phi\sum_{i=1}^{m}\sum_{j=0,1} w_j^{(i)}\log\frac{p(y^{(i)}|x^{(i)},z^{(i)}=j;\theta_j)p(z^{(i)}=j|x^{(i)};\phi)}{w_j^{(i)}} =$$

$$\nabla_\phi\sum_{i=1}^{m}\sum_{j=0,1} w_j^{(i)}\log p(z^{(i)}=j|x^{(i)};\phi) = \sum_{i=1}^{m}\left(w_0^{(i)}\log g(\phi^T x) + (1-w_0^{(i)})\log(1-g(\phi^T x^{(i)}))\right)$$

This term is the same as the objective for logistic regression task, but with the $w^{(i)}$ quantity replacing $y^{(i)}$. Therefore, the gradient and Hessian are given by

$$\nabla_\phi \sum_{i=1}^{m} \sum_{j=0,1} w_j^{(i)} \log p(z^{(i)} = j | x^{(i)}; \phi) = X^T(\vec{w} - \vec{h}), \quad \vec{h}_i = g(\phi^T x^{(i)}),$$

$$H = X^T D X, \quad D_{ii} = g(\phi^T x^{(i)})(1 - g(\phi^T x^{(i)})).$$

2. **Factor Analysis and PCA**

In this problem we look at the relationship between two unsupervised learning algorithms we discussed in class: Factor Analysis and Principle Component Analysis.

Consider the following joint distribution over (x, z) where $z \in \mathbb{R}^k$ is a latent random variable

$$\begin{aligned} z &\sim \mathcal{N}(0, I) \\ x|z &\sim \mathcal{N}(Uz, \sigma^2 I). \end{aligned}$$

where $U \in \mathbb{R}^{n \times k}$ is a model parameters and σ^2 is assumed to be a known constant. This model is often called Probabilistic PCA. Note that this is nearly identical to the factor analysis model except we assume that the variance of $x|z$ is a known scaled identity matrix rather than the diagonal parameter matrix, Φ, and we do not add an additional μ term to the mean (though this last difference is just for simplicity of presentation). However, as we will see, it turns out that as $\sigma^2 \to 0$, this model is equivalent to PCA.

For simplicity, you can assume for the remainder of the problem that $k = 1$, i.e., that U is a column vector in $\mathbb{R}^n$.

(a) Use the rules for manipulating Gaussian distributions to determine the joint distribution over (x, z) and the conditional distribution of $z|x$. [Hint: for later parts of this problem, it will help significantly if you simplify your soluting for the conditional distribution using the identity we first mentioned in problem set #1: $(\lambda I + BA)^{-1}B = B(\lambda I + AB)^{-1}$.]

Answer: To compute the joint distribution, we compute the means and covariances of x and z. First, $E[z] = 0$ and

$$E[x] = E[Uz + \epsilon] = UE[z] + E[\epsilon] = 0, \quad (\text{where } \epsilon \sim \mathcal{N}(0, \sigma^2 I)).$$

Since both x and z have zero mean

$$\begin{aligned} \Sigma_{zz} &= E[zz^T] = I \quad (= 1, \text{ since } z \text{ is a scalar when } k = 1) \\ \Sigma_{zx} &= E[(Uz + \epsilon)z^T] = UE[zz^T] + E[\epsilon z^T] = U \\ \Sigma_{xx} &= E[(Uz + \epsilon)(Uz + \epsilon)^T] = E[Uzz^TU^T + \epsilon z^T U^T + Uz\epsilon^T + \epsilon\epsilon^T] \\ &= UE[zz^T]U^T + E[\epsilon\epsilon^T] = UU^T + \sigma^2 I \end{aligned}$$

Therefore,

$$\begin{bmatrix} z \\ x \end{bmatrix} \sim \mathcal{N}\left(\begin{bmatrix} 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 1 & U^T \\ U & UU^T + \sigma^2 I \end{bmatrix} \right).$$

Using the rules for conditional Gaussian distributions, $z|x$ is also Gaussian with mean and covariance

$$\begin{aligned} \mu_{z|x} &= U^T(UU^T + \sigma^2 I)^{-1} x = \frac{U^T x}{U^T U + \sigma^2} \\ \Sigma_{z|x} &= 1 - U^T(UU^T + \sigma^2 I)^{-1} U = 1 - \frac{U^T U}{U^T U + \sigma^2} \end{aligned}$$

where in both cases the last equality comes from the identity in the hint.

(b) Using these distributions, derive an EM algorithm for the model. Clearly state the E-step and the M-step of the algorithm.

Answer: Even though $z^{(i)}$ is a scalar value, in this problem we continue to use the notation ${z^{(i)}}^T$, etc, to make the similarities to the Factor anlysis case obvious.

For the E-step, we compute the distribution $Q_i(z^{(i)}) = p(z^{(i)}|x^{(i)}; U)$ by computing $\mu_{z^{(i)}|x^{(i)}}$ and $\Sigma_{z^{(i)}|x^{(i)}}$ using the above formulas.

For the M-step, we need to maximize

$$\sum_{i=1}^{m} \int_{z^{(i)}} Q_i(z^{(i)}) \log \frac{p(x^{(i)}, |z^{(i)}; U) p(z^{(i)})}{Q_i(z^{(i)})}$$

$$= \sum_{i=1}^{m} E_{z^{(i)} \sim Q_i} \left[\log p(x^{(i)}|z^{(i)}; U) + \log p(z^{(i)}) - \log Q_i(z^{(i)}) \right].$$

Taking the gradient with respect to U equal to zero, dropping terms that don't depend on U, and omitting the subscript on the expectation, this becomes

$$\begin{aligned}
\nabla_U \sum_{i=1}^{m} E\left[\log p(x^{(i)}|z^{(i)}; U)\right] &= \nabla_U \sum_{i=1}^{m} E\left[-\frac{1}{2\sigma^2}(x^{(i)} - Uz^{(i)})^T(x^{(i)} - Uz^{(i)})\right] \\
&= -\frac{1}{2\sigma^2} \sum_{i=1}^{m} \nabla_U E\left[\mathrm{tr} {z^{(i)}}^T U^T U z^{(i)} - 2\mathrm{tr} {z^{(i)}}^T U^T x^{(i)}\right] \\
&= -\frac{1}{2\sigma^2} \sum_{i=1}^{m} E\left[U z^{(i)} {z^{(i)}}^T - x^{(i)} {z^{(i)}}^T\right] \\
&= \frac{1}{2\sigma^2} \sum_{i=1}^{m} \left[-U E[z^{(i)} {z^{(i)}}^T] + x^{(i)} E[{z^{(i)}}^T]\right]
\end{aligned}$$

using the same reasoning as in the Factor Analysis class notes. Setting this derivative to zero gives

$$\begin{aligned}
U &= \left(\sum_{i=1}^{m} x^{(i)} E[{z^{(i)}}^T]\right) \left(\sum_{i=1}^{m} E[z^{(i)} {z^{(i)}}^T]\right)^{-1} \\
&= \left(\sum_{i=1}^{m} x^{(i)} \mu_{z^{(i}|x^{(i)}}^T\right) \left(\sum_{i=1}^{m} \Sigma_{z^{(i)}|x^{(i)}} + \mu_{z^{(i}|x^{(i)}} \mu_{z^{(i}|x^{(i)}}^T\right)^{-1}
\end{aligned}$$

All these terms were calcuated in the E step, so this is our final M step update.

(c) As $\sigma^2 \to 0$, show that if the EM algorithm convergences to a parameter vector $U^\star$ (and such convergence is guarenteed by the argument presented in class), then $U^\star$ must be an eigenvector of the sample covariance matrix $\Sigma = \frac{1}{m} \sum_{i=1}^{m} x^{(i)} {x^{(i)}}^T$ — i.e., $U^\star$ must satisfy

$$\lambda U^\star = \Sigma U^\star.$$

[Hint: When $\sigma^2 \to 0$, $\Sigma_{z|x} \to 0$, so the E step only needs to compute the means $\mu_{z|x}$ and not the variances. Let $w \in \mathbb{R}^m$ be a vector containing all these means,

$w_i = \mu_{z^{(i)}|x^{(i)}}$, and show that the E step and M step can be expressed as

$$w = \frac{XU}{U^T U}, \quad U = \frac{X^T w}{w^T w}$$

respectively. Finally, show that if U doesn't change after this update, it must satisfy the eigenvector equation shown above.]

Answer: For the E step, when $\sigma^2 \to 0$, $\mu_{z^{(i)}|x^{(i)}} = \frac{U^T x^{(i)}}{U^T U}$, so using w as defined in the hint we have

$$w = \frac{XU}{U^T U}$$

as desired.

As mentioned in the hint, when $\sigma^2 \to 0$, $\Sigma_{z^{(i)}|x^{(i)}} = 0$, so

$$\begin{aligned} U &= \left(\sum_{i=1}^m x^{(i)} \mu_{z^{(i}|x^{(i)}}^T\right) \left(\sum_{i=1}^m \Sigma_{z^{(i)}|x^{(i)}} + \mu_{z^{(i}|x^{(i)}} \mu_{z^{(i}|x^{(i)}}^T\right)^{-1} \\ &= \left(\sum_{i=1}^m x^{(i)} w_i\right) (\sum_{i=1}^m w_i w_i)^{-1} = \frac{X^T w}{w^T w} \end{aligned}$$

For U to remain unchanged after an update requires that

$$U = \frac{X^T \frac{XU}{U^T U}}{\frac{U^T X^T}{U^T U} \frac{XU}{U^T U}} = X^T XU \frac{U^T U}{U^T X^T XU} = X^T XU \frac{1}{\lambda}$$

proving the desired equation.

3. **PCA and ICA for Natural Images**

In this problem we'll apply Principal Component Analysis and Independent Component Analysis to images patches collected from "natural" image scenes (pictures of leaves, grass, etc). This is one of the classical applications of the ICA algorithm, and sparked a great deal of interest in the algorithm; it was observed that the bases recovered by ICA closely resemble image filters present in the first layer of the visual cortex.

The `q3/` directory contains the data and several useful pieces of code for this problem. The raw images are stored in the *images/* subdirectory, though you will not need to work with these directly, since we provide code for loading and normalizing the images.

Calling the function `[X_ica, X_pca] = load_images;` will load the images, break them into 16x16 images patches, and place all these patches into the columns of the matrices `X_ica` and `X_pca`. We create two different data sets for PCA and ICA because the algorithms require slightly different methods of preprocessing the data.[1]

For this problem you'll implement the `ica.m` and `pca.m` functions, using the PCA and ICA algorithms described in the class notes. While the PCA implementation should be straightforward, getting a good implementation of ICA can be a bit trickier. Here is some general advice to getting a good implementation on this data set:

[1] Recall that the first step of performing PCA is to subtract the mean and normalize the variance of the features. For the image data we're using, the preprocessing step for the ICA algorithm is slightly different, though the precise mechanism and justification is not imporant for the sake of this problem. Those who are curious about the details should read Bell and Sejnowki's paper "The 'Independent Components' of Natural Scenes are Edge Filters," which provided the basis for the implementation we use in this problem.

- Picking a good learning rate is important. In our experiments we used $\alpha = 0.0005$ on this data set.
- Batch gradient descent doesn't work well for ICA (this has to do with the fact that ICA objective function is not concave), but the pure stochastic gradient described in the notes can be slow (There are about 20,000 16x16 images patches in the data set, so one pass over the data using the stochastic gradient rule described in the notes requires inverting the 256x256 W matrix 20,000 times). Instead, a good compromise is to use a hybrid stochastic/batch gradient descent where we calculate the gradient with respect to several examples at a time (100 worked well for us), and use this to update W. Our implementation makes 10 total passes over the entire data set.
- It is a good idea to randomize the order of the examples presented to stochastic gradient descent before each pass over the data.
- Vectorize your Matlab code as much as possible. For general examples of how to do this, look at the Matlab review session.

For reference, computing the ICA W matrix for the entire set of image patches takes about 5 minutes on a 1.6 Ghz laptop using our implementation.

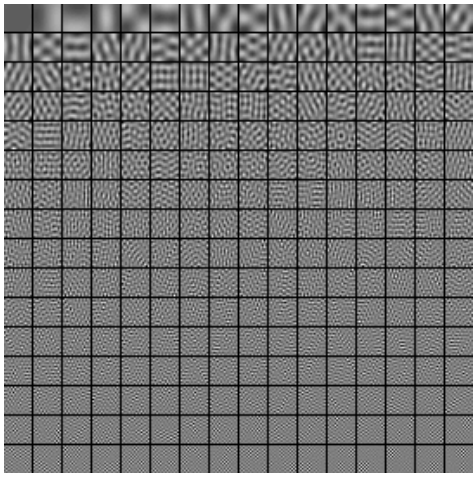
After you've learned the U matrix for PCA (the columns of U should contain the principal components of the data) and the W matrix of ICA, you can plot the basis functions using the `plot_ica_bases(W);` and `plot_pca_bases(U);` functions we have provide. Comment briefly on the difference between the two sets of basis functions.

Answer: The following are our implementations of `pca.m` and `ica.m`:

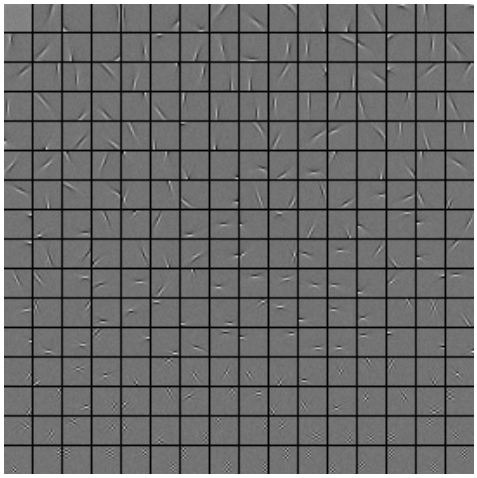
```
function U = pca(X)

[U,S,V] = svd(X*X');
```

```
function W = ica(X)

[n,m] = size(X);
chunk = 100;
alpha = 0.0005;
W = eye(n);

for iter=1:10,
  disp([num2str(iter)]);
  X = X(:,randperm(m));
  for i=1:floor(m/chunk),
    Xc = X(:,(i-1)*chunk+1:i*chunk);
    dW = (1 - 2./(1+exp(-W*Xc)))*Xc' + chunk*inv(W');
    W = W + alpha*dW;
  end
end
```

PCA produces the following bases:



while ICA produces the following bases



The PCA bases capture global features of the images, while the ICA bases capture more local features.

4. **Convergence of Policy Iteration**

   In this problem we show that the Policy Iteration algorithm, described in the lecture notes, is guarenteed to find the optimal policy for an MDP. First, define B^π to be the Bellman operator for policy π, defined as follows: if $V' = B(V)$, then

$$V'(s) = R(s) + \gamma \sum_{s' \in \mathcal{S}} P_{s\pi(s)}(s')V(s').$$

(a) Prove that if $V_1(s) \leq V_2(s)$ for all $s \in \mathbb{S}$, then $B(V_1)(s) \leq B(V_2)(s)$ for all $s \in \mathbb{S}$.
Answer:

$$\begin{aligned} B(V_1)(s) &= R(s) + \gamma \sum_{s' \in \mathcal{S}} P_{s\pi(s)}(s') V_1(s') \\ &\leq R(s) + \gamma \sum_{s' \in \mathcal{S}} P_{s\pi(s)}(s') V_2(s') = B(V_2)(s) \end{aligned}$$

where the inequality holds because $P_{s\pi(s)}(s') \geq 0$.

(b) Prove that for any V,

$$\|B^\pi(V) - V^\pi\|_\infty \leq \gamma \|V - V^\pi\|_\infty$$

where $\|V\|_\infty = \max_{s \in \mathcal{S}} |V(s)|$. Intuitively, this means that applying the Bellman operator B^π to any value function V, brings that value function "closer" to the value function for π, V^π. This also means that applying B^π repeatedly (an infinite number of times)

$$B^\pi(B^\pi(\ldots B^\pi(V)\ldots))$$

will result in the value function V^π (a little bit more is needed to make this completely formal, but we won't worry about that here).
[Hint: Use the fact that for any $\alpha, x \in \mathbb{R}^n$, if $\sum_i \alpha_i = 1$ and $\alpha_i \geq 0$, then $\sum_i \alpha_i x_i \leq \max_i x_i$.] **Answer:**

$$\begin{aligned} \|B^\pi(V) - V^\pi\|_\infty &= \max_{s' \in \mathcal{S}} \left| R(s) + \gamma \sum_{s' \in \mathcal{S}} P_{s\pi(s)}(s') V(s') - R(s) - \gamma \sum_{s' \in \mathcal{S}} P_{s\pi(s)}(s') V^\pi(s') \right| \\ &= \gamma \max_{s' \in J} \left| \sum_{s' \in \mathcal{S}} P_{s\pi(s)}(s') \left(V(s') - V^\pi(s') \right) \right| \\ &\leq \gamma \|V - V^\pi\|_\infty \end{aligned}$$

where the inequality follows from the hint above.

(c) Now suppose that we have some policy π, and use Policy Iteration to choose a new policy π' according to

$$\pi'(s) = \arg\max_{a \in \mathcal{A}} \sum_{s' \in \mathcal{S}} P_{sa}(s') V^\pi(s').$$

Show that this policy will never perform worse that the previous one — i.e., show that for all $s \in \mathcal{S}$, $V^\pi(s) \leq V^{\pi'}(s)$.
[Hint: First show that $V^\pi(s) \leq B^{\pi'}(V^\pi)(s)$, then use the proceeding excercises to show that $B^{\pi'}(V^\pi)(s) \leq V^{\pi'}(s)$.]
Answer:

$$\begin{aligned} V^\pi(s) &= R(s) + \gamma \sum_{s' \in \mathcal{S}} P_{s\pi(s)}(s') V^\pi(s') \\ &\leq R(s) + \gamma \max_{a \in \mathcal{A}} \sum_{s' \in \mathcal{S}} P_{sa}(s') V^\pi(s') \\ &= R(s) + \gamma \sum_{s' \in \mathcal{S}} P_{s\pi'(s)}(s') V^\pi(s') = B^{\pi'}(V^\pi)(s) \end{aligned}$$

Applying part (a),

$$V^{\pi}(s) \leq B^{\pi'}(V^{\pi})(s) \Rightarrow B^{\pi'}(V^{\pi})(s) \leq B^{\pi'}(B^{\pi'}(V^{\pi}))(s)$$

Continually applying this property, and applying part (b), we obtain

$$V^{\pi}(s) \leq B^{\pi'}(V^{\pi})(s) \leq B^{\pi'}(B^{\pi'}(V^{\pi}))(s) \leq \ldots \leq B^{\pi'}(B^{\pi'}(\ldots B^{\pi'}(V^{\pi})\ldots))(s) = V^{\pi'}(s).$$

(d) Use the proceeding exercises to show that policy iteration will eventually converge (i.e., produce a policy $\pi' = \pi$). Furthermore, show that it must converge to the optimal policy $\pi^{\star}$. For the later part, you may use the property that if some value function satisfies

$$V(s) = R(s) + \gamma \max_{a \in \mathcal{A}} \sum s' \in \mathcal{S} P_{sa}(s') V(s')$$

then $V = V^{\star}$.

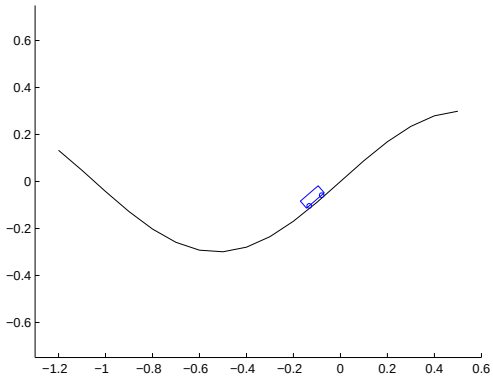
Answer: We know that policy iteration must converge because there are only a finite number of possible policies (if there are $|S|$ states, each with $|A|$ actions, then that leads to a $|S|^{|A|}$ total possible policies). Since the policies are monotonically improving, as we showed in part (c), at some point we must stop generating new policies, so the algorithm must produce $\pi' = \pi$. Using the assumptions stated in the question, it is easy to show convergence to the optimal policy. If $\pi' = \pi$, then using the same logic as in part (c)

$$V^{\pi}(s) = V^{\pi'}(s) = R(s) + \gamma \max_{a \in \mathcal{A}} \sum_{s' \in \int} P_{sa}(s') V^{\pi}(s),$$

So $V = V^{\star}$, and therefore $\pi = \pi^{\star}$.

5. **Reinforcement Learning: The Mountain Car**

In this problem you will implement the Q-Learning reinforcement learning algorithm described in class on a standard control domain known as the Mountain Car.[2] The Mountain Car domain simulates a car trying to drive up a hill, as shown in the figure below.



[2]The dynamics of this domain were taken from Sutton and Barto, 1998.

All states except those at the top of the hill have a constant reward $R(s) = -1$, while the goal state at the hilltop has reward $R(s) = 0$; thus an optimal agent will try to get to the top of the hill as fast as possible (when the car reaches the top of the hill, the episode is over, and the car is reset to its initial position). However, when starting at the bottom of the hill, the car does not have enough power to reach the top by driving forward, so it must first accerlaterate accelerate backwards, building up enough momentum to reach the top of the hill. This strategy of moving away from the goal in order to reach the goal makes the problem difficult for many classical control algorithms.

As discussed in class, Q-learning maintains a table of Q-values, $Q(s, a)$, for each state and action. These Q-values are useful because, in order to select an action in state s, we only need to check to see which Q-value is greatest. That is, in state s we take the action

$$\arg\max_{a \in \mathcal{A}} Q(s, a).$$

The Q-learning algorithm adjusts its estimates of the Q-values as follows. If an agent is in state s, takes action a, then ends up in state s', Q-learning will update $Q(s, a)$ by

$$Q(s, a) = (1 - \alpha)Q(s, a) + \gamma(R(s') + \gamma \max_{a' \in \mathcal{A}} Q(s', a').$$

At each time, your implementation of Q-learning can execute the greedy policy $\pi(s) = \arg\max_{a \in \mathcal{A}} Q(s, a)$

Implement the `[q, steps_per_episode] = qlearning(episodes)` function in the `q5/` directory. As input, the function takes the total number of episodes (each episode starts with the car at the bottom of the hill, and lasts until the car reaches the top), and outputs a matrix of the Q-values and a vector indicating how many steps it took before the car was able to reach the top of the hill. You should use the `[x, s, absorb] = mountain_car(x, actions(a))` function to simulate one control cycle for the task — the `x` variable describes the true (continuous) state of the system, whereas the `s` variable describes the discrete index of the state, which you'll use to build the Q values.

Plot a graph showing the average number of steps before the car reaches the top of the hill versus the episode number (there is quite a bit of variation in this quantity, so you will probably want to average these over a large number of episodes, as this will give you a better idea of how the number of steps before reaching the hilltop is decreasing). You can also visualize your resulting controller by calling the `draw_mountain_car(q)` function.

Answer: The following is our implementation of `qlearning.m`:

```
function [q, steps_per_episode] = qlearning(episodes)

% set up parameters and initialize q values
alpha = 0.05;
gamma = 0.99;
num_states = 100;
num_actions = 2;
actions = [-1, 1];
q = zeros(num_states, num_actions);

for i=1:episodes,
```

```
  [x, s, absorb] =  mountain_car([0.0 -pi/6], 0);
  [maxq, a] = max(q(s,:));
  if (q(s,1) == q(s,2)) a = ceil(rand*num_actions); end;
  steps = 0;

  while (~absorb)
    % execute the best action or a random action
    [x, sn, absorb] = mountain_car(x, actions(a));
    reward = -double(absorb == 0);

    % find the best action for the next state and update q value
    [maxq, an] = max(q(sn,:));
    if (q(sn,1) == q(sn,2)) an = ceil(rand*num_actions); end
    q(s,a) = (1 - alpha)*q(s,a) + alpha*(reward + gamma*maxq);
    a = an;
    s = sn;
    steps = steps + 1;
  end
  steps_per_episode(i) = steps;
end
```

Within 10000 episodes, the algorithm converges to a policy that usually gets the car up the hill in around 52-53 steps. The following plot shows the number of steps per episode (averaged over 500 episodes) versus the number of episodes. We generated the plot using the following code:

```
for i=1:10,
  [q, ep_steps] = qlearning(10000);
  all_ep_steps(i,:) = ep_steps;
end
plot(mean(reshape(mean(all_ep_steps), 500, 20)));
```