# A Practical Companion to the ML Exercise Book

# 1 Regression Methods

**1.** **(Linear regression applied to predicting the level of PSA in the prostate gland, using a set of medical test results)**

● ○ ⋆ ⋆ *CMU, 2009 fall, Geoff Gordon, HW3, pr. 3*

**The linear regression method is widely used in the medical domain. In this question you will work on a prostate cancer data from a study by Stamey et al.[1] You can download the data from ... .**

**Your task is to predict the level of prostate-specific antigen (PSA) using a set of medical test results. PSA is a protein produced by the cells of the prostate gland. High levels of PSA often indicate the presence of prostate cancer or other prostate disorders.**

**The attributes are several clinical measurements on men who have prostate cancer. There are 8 attributes: log cancer volume lcavol, log prostate weight (`lweight`), log of the amount of benign prostatic hyperplasia (`lbph`), seminal vesicle invasion (`svi`), `age`, log of capsular penetration (`lcp`), Gleason score (`gleason`), and percent of Gleason scores of 4 or 5 (`pgg45`). `svi` and `gleason` are categorical, that is they take values either 1 or 0; others are real-valued. We will refer to these attributes as $A_1$ = `lcavol`, $A_2$ = `lweight`, $A_3$ = `age`, $A_4$ = `lbph`, $A_5$ = `svi`, $A_6$ = `lcp`, $A_7$ = `gleason`, $A_8$ = `pgg45`.**

**Each row of the input file describes one data point: the first column is the index of the data point, the following eight columns are attributes, and the tenth column gives the log PSA level `lpsa`, the response variable we are interested in. We already randomized the data and split it into three parts corresponding to training, validation and test sets. The last column of the file indicates whether the data point belongs to the training set, validation set or test set, indicated by '1' for training, '2' for validation and '3' for testing. The training data includes 57 examples; validation and test sets contain 20 examples each.**

Inspecting the Data

**a. Calculate the correlation matrix of the 8 attributes and report it in a table. The table should be 8-by-8. You can use Matlab functions.**

**b. Report the top 2 pairs of attributes that show the highest pairwise positive correlation and the top 2 pairs of attributes that show the highest pairwise negative correlation.**

Solving the Linear Regression Problem

**You will now try to find several models in order to predict the lpsa levels. The linear regression model is**

$$Y = f(X) + \epsilon$$

[1] Stamey TA, Kabalin JN, McNeal JE et al. Prostate specific antigen in the diagnosis and treatment of the prostate. II. Radical prostatectomy treated patients. J Urol 1989;141:107683.

**where $\epsilon$ is a Gaussian noise variable, and**

$$f(X) = \sum_{j=0}^{p} w_j \phi_j(X)$$

**where $p$ is the number of basis functions (features), $\phi_j$ is the $j$th basis function, and $w_j$ is the weight we wish to learn for the $j$th basis function. In the models below, we will always assume that $\phi_0(X) = 1$ represents the *intercept term*.**

**c. Write a Matlab function that takes the data matrix $\Phi$ and the column vector of responses $y$ as an input and produces the least squares fit $w$ as the output (refer to the lecture notes for the calculation of $w$).**

**d. You will create the following three models. Note that before solving each regression problem below, you should scale each feature vector to have a zero mean and unit variance. Don't forget to include the intercept column, $\phi_0(X) = 1$, after scaling the other features. Notice that since you shifted the attributes to have zero mean, in your solutions, the intercept term will be the mean of the response variable.**

- **Model1: Features are equal to input attributes, with the addition of a constant feature $\phi_0$. That is, $\phi_0(X) = 1$, $\phi_1(X) = A_1, \ldots, \phi_8(X) = A_8$. Solve the linear regression problem and report the resulting feature weights. Discuss what it means for a feature to have a large negative weight, a large positive weight, or a small weight. Would you be able to comment on the weights, if you had not scaled the predictors to have the same variance? Report mean squared error (MSE) on the training and validation data.**
- **Model2: Include additional features corresponding to pairwise products of the first six of the original attributes,[2] i.e., $\phi_9(X) = A_1 \cdot A_2$, ..., $\phi_{13}(X) = A_1 \cdot A_6$, $\phi_{15}(X) = A_2 \cdot A_3$, ..., $\phi_{23}(X) = A_5 \cdot A_6$. First compute the features according to the formulas above using the unnormalized values, then shift and scale the new features to have zero mean and unit variance and add the column for the intercept term $\phi_0(X) = 1$. Report the five features whose weights achieved the largest absolute values.**
- **Model3: Starting with the results of Model1, drop the four features with the lowest weights (in absolute values). Build a new model using only the remaining features. Report the resulting weights.**

**e. Make two bar charts, the first to compare the training errors of the three models, the second to compare the validation errors of the three models. Which model achieves the best performance on the training data? Which model achieves the best performance on the validation data? Comment on differences between training and validation errors for individual models.**

**f. Which of the models would you use for predicting the response variable? Explain.**

**Ridge Regression**

**For this question you will start with Model2 and employ regularization on it.**

[2] These features are also called *interactions*, because they attempt to account for the effect of two attributes being simultaneously high or simultaneously low.

**g. Write a Matlab function to solve Ridge regression. The function should take the data matrix $\Phi$, the column vector of responses $y$, and the regularization parameter $\lambda$ as the inputs and produce the least squares fit $w$ as the output (refer to the *lecture notes* for the calculation of $w$). Do not penalize $w_0$, the *intercept term*. (You can achieve this by replacing the first column of the $\lambda I$ matrix with zeros.)**

**h. You will create a plot exploring the effect of the regularization parameter on training and validation errors. The x-axis is the *regularization parameter* (on a log scale) and the y-axis is the *mean squared error*. Show two curves in the same graph, one for the *training error* and one for the *validation error*. Starting with $\lambda = 2^{-30}$, try 50 values: at each iteration increase $\lambda$ by a factor of 2, so that for example the second iteration uses $\lambda = 2^{-29}$. For each $\lambda$, you need to train a new model.**

**i. What happens to the training error as the regularization parameter increases? What about the validation error? Explain the curve in terms of overfitting, bias and variance.**

**j. What is the $\lambda$ that achieves the lowest validation error and what is the validation error at that point? Compare this validation error to the Model2 validation error when no regularization was applied (you solved this in part $e$). How does w differ in the regularized and unregularized versions, i.e., what effect did regularization have on the weights?**

**k. Is this validation error lower or higher than the validation error of the model you chose in part $f$? Which one should be your final model?**

**l. Now that you have decided on your model (features and possibly the regularization parameter), combine your training and validation data to make a *combined training set*, train your model on this combined training set, and calculate it on the test set. Report the *training* and *test errors*.**

**Solution:**

**a.**

| | lcavol | lweight | age | lbph | svi | lcp | gleason | pgg45 | lpsa |
|---|---|---|---|---|---|---|---|---|---|
| lcavol | 1.0000 | 0.2805 | 0.2249 | 0.0273 | 0.5388 | 0.6753 | 0.4324 | 0.4336 | 0.7344 |
| lweight | 0.2805 | 1.0000 | 0.3479 | 0.4422 | 0.1553 | 0.1645 | 0.0568 | 0.1073 | 0.4333 |
| age | 0.2249 | 0.3479 | 1.0000 | 0.3501 | 0.1176 | 0.1276 | 0.2688 | 0.2761 | 0.1695 |
| lbph | 0.0273 | 0.4422 | 0.3501 | 1.0000 | -0.0858 | -0.0069 | 0.0778 | 0.0784 | 0.1798 |
| svi | 0.5388 | 0.1553 | 0.1176 | -0.0858 | 1.0000 | 0.6731 | 0.3204 | 0.4576 | 0.5662 |
| lcp | 0.6753 | 0.1645 | 0.1276 | -0.0069 | 0.6731 | 1.0000 | 0.5148 | 0.6315 | 0.5488 |
| gleason | 0.4324 | 0.0568 | 0.2688 | 0.0778 | 0.3204 | 0.5148 | 1.0000 | 0.7519 | 0.3689 |
| pgg45 | 0.4336 | 0.1073 | 0.2761 | 0.0784 | 0.4576 | 0.6315 | 0.7519 | 1.0000 | 0.4223 |
| lpsa | 0.7344 | 0.4333 | 0.1695 | 0.1798 | 0.5662 | 0.5488 | 0.3689 | 0.4223 | 1.0000 |

**b. The top 2 pairs that show the highest pairwise positive correlation are `gleason` - `ppg4` (0.7519) and `lcavol` -`lcp` (0.6731). Highest negative correlations: `lbph` - `svi` (-0.0858) and `lph` - `lcp` (-0.0070).**

**c. See below:**

```
function what=lregress(Y,X)
% least square solution to linear regression
% X is the feature matrix
% Y is the response variable vector
what=inv(X'*X)*X'*Y;
end
```

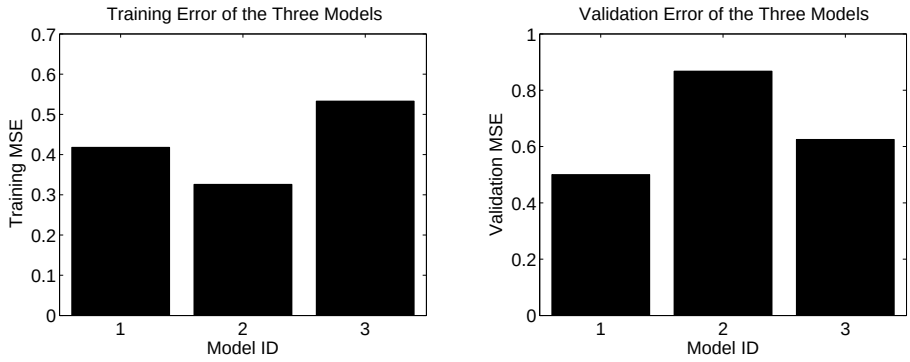
**d.**

**Model1:**
**the weight vector:**
$w = [2.68265, 0.71796, 0.17843, -0.21235, 0.25752, 0.42998, -0.14179, 0.08745, 0.02928]$.

**Model2:**
**The largest five absolute values in descending order:**
`lweight*age`, `lpbh`, `lweight`, `age`, `age*lpbh`.

**Model3:**
**The features with have the lowest absolute weights in Model1:**
`pgg45`, `gleason`, `lcp`, `lweight`.
**The resulting weights:** $w = [2.6827, 0.7164, -0.1735, 0.3441, 0.4095]$.

**e.**



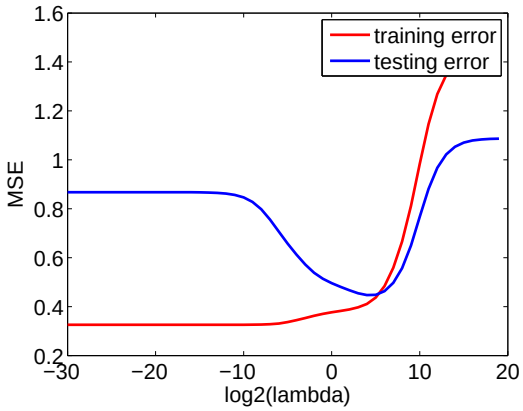

**Model2 achieves the best performance on the training data, whereas Model1 achieves the best performance on the validation data. Model2 suffer from *overfitting*, indicated by the very good training model but low validation error. Model3 seems to be too simple, it has a higher training and a higher validation error compared to Model1. The features that are dropped are informative, as indicated by the lower training and validation errors.**

**f. Model1, since it achieves the best performance on the validation data. Model2 overfits, and Model3 is too simple.**

**g. See below:**

```
function what = ridgeregress(Y,X,lambda)
% X is the feature matrix
% Y is the response vector
% what are the estimated weights
penal = lambda*eye(size(X,2));
penal(:,1) = 0;
what = inv(X'*X+penal)*X'*Y;
end
```

**h.**




**i. When the model is not regularized much (the left side of the graph), the training error is low and the validation error is high, indicating the model is too complex and *overfitting* to the training data. In that region, the *bias* is low and the *variance* is high.**
**As the regularization parameter increases, the bias increases and variance decreases. The overfitting problem is overcome as indicated by decreasing validation error and increasing training error.**
**As regularization penalty increase too much, the model becomes getting too simple and start suffering from *underfitting* as can be shown by the poor performance on the training data.**

**j.** $log\lambda = 4$**, i.e.,** $\lambda = 16$**, achieves the lowest validation error, which is 0.447. This validation error is much less than the validation error of the model without regularization, which was 0.867. Regularized weights are smaller than unregularized weights. Regularization decreases the magnitude of the weights.**

**k. The validation error of the penalized model (**$\lambda = 16$**) is 0.447, which is lower than Model1's validation error, 0.5005. Therefore, this model is chosen.**

**l. The final models' training error is 0.40661 and the test error is 0.58892.**

**2.** **([Weighted] Linear Regression applied to predicting the needed quatity of insulin, starting from the sugar level in the patient's blood)**

● ○ ***CMU, 2009 spring, Ziv Bar-Joseph, HW1, pr. 4***

**An automated insulin injector needs to calculate how much insulin it should inject into a patient based on the patient's blood sugar level. Let us formulate this as a linear regression problem as follows: let $y_i$ be the dependent predicted variable (blood sugar level), and let $\beta_0, \beta_1$ and $\beta_2$ be the unknown coefficients of the regression function. Thus, $y_i = \beta_0 + \beta_1 x_i + \beta_2 x_i^2$, and we can formulate the problem of finding the unknown $\beta \overset{not.}{=} (\beta_0, \beta_1, \beta_2)$ as:**

$$\hat{\beta} = (X^\top X)^{-1} X^\top y.$$

**See `data2.txt` (posted on website) for data based on the above scenario with space separated fields conforming to:**
`bloodsugarlevel insulinedose weightage`
**The purpose of the `weightage` field will be made clear in part $c$.**

**a. Write code in Matlab to estimate the regression coefficients given the dataset consisting of pairs of independent and dependent variables. Generate a space separated file with the estimated parameters from the entire dataset by writing out all the parameter.**

**b. Write code in Matlab to perform inference by predicting the insulin dosage given the blood sugar level based on training data using a leave one out cross validation scheme. Generate a space separated file with the predicted dosages in order. The predicted dosages:**

**c. However, it has been found that one group of patients are twice as sensitive to the insuline dosage than the other. In the training data, these particular patients are given a `weightage` of 2, while the others are given a `weightage` of 1. Is your goodness of fit function flexible enough to incorporate this information?**

**d. Show how to formulate the regression function, and correspondingly calculate the coefficients of regression under this new scenario, by incorporating the given weights.**

**e. Code up this variant of regressional analysis. Write out the new coefficients of regression you obtain by using the whole dataset as training data.**

<u>Solution:</u>

**a. The betas, in order:**
$-74.3825$ 13.4215 1.1941

**b.**
1417.0177 1501.4423 1966.3563 2833.7942 2953.4532 3075.472 3199.8566 3326.9663
3456.4704 5038.3777 5196.6767 5357.0094 7091.8113 7278.2709 7467.3604 7658.5256
7852.0808 9703.9748 9921.8604 10142.5438 10365.8512 12498.2968 12749.6202 13003.94
1798.3745 1869.1966 2024.7112 2265.6988 2392.0227 2756.1588 3915.9302 4004.465
4878.0057 5094.3282 6217.981 6485.8542 6544.4688 6805.7564 7073.8455 7207.5032
7285.3657 9393.5129 9515.9043 9704.0029 10060.5539 12037.6304 12361.3586 12903.5394

**c. Since we need to weigh each point differently, out current goodness of fit function is unable to work in this scenario. However, since the weights for this specific dataset are 1 and 2, we may just use the old formalism and double the data items with weightage 2. The changed formalism which enables us to assign weights of any precision to the data sample is shown below.**

**d. Let:**

$$\begin{aligned} y &= X\beta \\ y &= (y_1, y_2, \ldots, y_n)^\top \\ \beta &= (\beta_1, \beta_2, \ldots, \beta_m)^\top \\ X_{i,1} &= 1 \\ X_{i,j+1} &= x_{i,j} \end{aligned}$$

**Let us define the weight matrix as:**
$\Omega_{i,i} = \sqrt{w_i}$
$\Omega_{i,j} = 0 \quad (\text{for} \quad i \neq j)$
**So,** $\Omega y = \Omega X\beta$.

**To minimize the weighted square error, we have to take the derivative with respect to** $\beta$**:**

$$\begin{aligned} &\frac{\partial}{\partial\beta}((\Omega y - \Omega X\beta)^\top(\Omega y - \Omega X\beta)) \\ &= \frac{\partial}{\partial\beta}((\Omega y)^\top(\Omega y) - 2(\Omega y)^\top(\Omega X\beta) + (\Omega X\beta)^\top(\Omega X\beta) \\ &= \frac{\partial}{\partial\beta}((\Omega y)^\top(\Omega y) - 2(\Omega y)^\top(\Omega X\beta) + \beta^\top X^\top\Omega^\top\Omega X\beta \end{aligned}$$

**Therefore**

$$\begin{aligned} &\frac{\partial}{\partial\beta}((\Omega y - \Omega X\beta)^\top(\Omega y - \Omega X\beta)) = 0 \\ &\Leftrightarrow \quad 0 - 2((\Omega y)^\top(\Omega X))^\top + 2X^\top\Omega^\top\Omega X\hat{\beta} = 0 \\ &\Leftrightarrow \quad \hat{\beta} = (X^\top\Omega^\top\Omega X)^{-1}X^\top\Omega^\top\Omega y \end{aligned}$$

**e. The new beta coefficients are in order:**
$-57.808\ 13.821\ 1.199$

**3.** **(Logistic regression: application to text classification)**

● ○ ⋆ ***CMU, 2010 fall, Aarti Singh, HW1, pr. 5***

**In this problem you will implement Logistic Regression and evaluate its performance on a document classification task. The data for this task is taken from the 20 Newsgroups data set,[3] and is available from the course web page. The included `README.txt` describes the data set and file format.**

**Our model will use the *bag-of-words assumption*. This model assumes that each word in a document is drawn independently from a categorical distribution over possible words. (A categorical distribution is a generalization of a Bernoulli distribution to multiple values.) Although this model ignores the ordering of words in a document, it works surprisingly well for a number of tasks. We number the words in our vocabulary from 1 to $m$, where $m$ is the total number of distinct words in all of the documents. Documents from class $y$ are drawn from a class-specific categorical distribution parameterized by $\theta_y$. $\theta_y$ is a vector, where $\theta_{y,i}$ is the probability of drawing word $i$ and $\sum_{i=1}^{m} \theta_{y,i} = 1$. Therefore, the class-conditional probability of drawing document $x$ from our model is**

$$P(X = x|Y = y) = \prod_{i=1}^{m} \theta_{y,i}^{\text{count}_i(x)},$$

**where $\text{count}_i(x)$ is the number of times word $i$ appears in $x$.**

**a. Provide high-level descriptions of the Logistic Regression algorithm. Be sure to describe how to estimate the model parameters and how to classify a new example.**

**b. Implement Logistic Regression. Train the model on the provided training data and predict the labels of the test data. Report the training and test error.**

<u>Solution:</u>

**a. The logistic regression model is**

$$P(Y = 1|X = x, w) = \frac{\exp(w_0 + \sum_i w_i x_i)}{1 + \exp(w_0 + \sum_i w_i x_i)},$$

**where $w = (w_0, w_1, \ldots, w_m)^\top$ is our parameter vector. We will find $\hat{w}$ by maximizing the data loglikelihood $l(w)$:**

$$\begin{aligned} l(w) &= \log\left(\prod_j \frac{\exp(y^j(w_0 + \sum_i w_i x_i^j))}{1 + \exp(w_0 + \sum_i w_i x_i^j)}\right) \\ &= \sum_j \left(y^j(w_0 + \sum_i w_i x_i^j) - \log(1 + \exp(w_0 + \sum_i w_i x_i^j))\right) \end{aligned}$$

**We can estimate/learn the parameters ($w$) of logistic regression by optimizing $l(w)$, using *gradient ascent*. The *gradient* of $l(w)$ is the array of partial derivatives of $l(w)$:**

[3] Full version available from http://people.csail.mit.edu/jrennie/20Newsgroups/.

$$\begin{aligned}
\frac{\partial l(w)}{\partial w_0} &= \sum_j \left( y^j - \frac{\exp(w_0 + \sum_i w_i x_i^j)}{1 + \exp(w_0 + \sum_i w_i x_i^j)} \right) \\
&= \sum_j (y^j - P(Y = 1 | X = x^j)) \\
\frac{\partial l(w)}{\partial w_k} &= \sum_j \left( y^j x_k^j - \frac{x_k^j \exp(w_0 + \sum_i w_i x_i^j)}{1 + \exp(w_0 + \sum_i w_i x_i^j)} \right) \\
&= \sum_j x_k^j (y^j - P(Y = 1 | X = x^j))
\end{aligned}$$

**Let $w^{(t)}$ represent our parameter vector on the $t$-th iteration of gradient ascent. To perform gradient ascent, we first set $w^{(0)}$ to some arbitrary value (say $0$). We then repeat the following *updates* until convergence:**

$$\begin{aligned}
w_0^{(t+1)} &\leftarrow w_0^{(t)} + \alpha \sum_j \left( y^j - P(Y = 1 | X = x^j, w^{(t)}) \right) \\
w_k^{(t+1)} &\leftarrow w_k^{(t)} + \alpha \sum_j x_k^j \left( y^j - P(Y = 1 | X = x^j, w^{(t)}) \right)
\end{aligned}$$

**where $\alpha$ is a step size parameter which controls how far we move along our gradient at each step. We set $\alpha = 0.0001$. The algorithm converges when $||w^{(t)} - w^{(t+1)}|| < \delta$, that is when the weight vector doesn't change much during an iteration. We set $\delta = 0.001$.**

**b. Training error: $0.00$. Test error: $0.29$. The large difference between training and test error means that our model *overfits* our training data. A possible *reason* is that we do not have enough training data to estimate either model accurately.**

**4.** **(Logistic Regression and Rosenblatt's Perceptron: application on the Breast Cancer dataset)**

• ○ ⋆ ***(CMU, 2009 spring, Ziv Bar-Joseph, HW2, pr. 4.1-2)***

**For this exercise, you will use the Breast Cancer dataset, downloadable from the course web page. Given 9 different attributes, such as "uniformity of cell size", the *task* is to predict malignancy.[4] The archive from the course web page contains a Matlab method `loaddata.m`, so you can easily load in the data by typing (from the directory containing `loaddata.m`): `data = loaddata`. The variables in the resulting data structure relevant for you are:**

- **`data.X`: 683 9-dimensional data points, each element in the interval $[1, 10]$.**
- **`data.Y`: the 683 corresponding classes, either $0$ (benign), or $1$ (malignant).**

**Logistic Regression**

**a. Write code in Matlab to train the weights for logistic regression. To avoid dealing with the *intercept term* explicitly, you can add a nonzero-constant tenth dimension to `data.X`: `data.X(:,10)=1`. Your regression function thus becomes simply:**

$$\begin{aligned} P(Y=0|x;w) &= \frac{1}{1+\exp(\sum_{k=1}^{10} x_k w_k)} \\ P(Y=1|x;w) &= \frac{\exp(\sum_{k=1}^{10} x_k w_k)}{1+\exp(\sum_{k=1}^{10} x_k w_k)} \end{aligned}$$

**and the gradient-ascend update rule:**

$$w \leftarrow w + \alpha/683 \sum_{j=1}^{683} x^j (y^j - P(Y^j = 1|x^j; w))$$

**Use the learning rate $\alpha = 1/10$. Try different learning rates if you cannot get $w$ to converge.**

**b. To test your program, use 10-fold cross-validation, splitting [`data.X data.Y`] into 10 random approximately equal-sized portions, training on 9 concatenated parts, and testing on the remaining part. Report the mean classification accuracy over the 10 runs, and the 95% *confidence interval*.**

**Rosenblatt's Perceptron**

**A very simple and popular linear classifier is the perceptron algorithm of Rosenblatt (1962), a single-layer neural network model of the form**

$$y(x) = f(w^\top x),$$

**with the activation function**

$$f(a) = \begin{cases} 1 & \text{if } a \geq 0 \\ -1 & \text{otherwise.} \end{cases}$$

**For this classifier, we need our classes to be $-1$ (benign) and $1$ (malignant), which can be achieved with the Matlab command: `data.Y = data.Y * 2 - 1`.**

[4] For more information on what the individual attributes mean, see ftp://ftp.ics.uci.edu/pub/machine-learning-databases/breast-cancer-wisconsin/breastcancer-wisconsin.names.

**Weight training usually proceeds in an online fashion, iterating through the individual data points $x^j$ one or more times. For each $x^j$, we compute the predicted class $\hat{y}^j = f(w^\top x^j)$ for $x^j$ under the current parameters $w$, and update the weight vector as follows:**

$$w \leftarrow w + x^j[y^j - \hat{y}^j].$$

**Note how $w$ only changes if $x_j$ was misclassified under the current model.**

**c. Implement this training algorithm in Matlab. To avoid dealing with the *intercept term* explicitly, augment each point in `data.X` with a non-zero constant tenth element. In Matlab this can be done by typing: `data.X(:,10)=1`. Have your algorithm iterate through the whole training data 20 times and report the number of examples that were still mis-classified in the 20th iteration. Does it look like the training data is linearly separable? (Hint: The perceptron algorithm is guaranteed to converge if the data is linearly separable.)**

**d. To test your program, use 10-fold cross-validation, using the splits you obtained in part $b$. For each split, do 20 training iterations to train the weights. Report the mean classification accuracy over the 10 runs, and the 95% confidence interval.**

**e. If the data is not linearly separable, weights can toggle back and forth from iteration to iteration. Even in the linearly separable case, the learned model is often very dependent on which training data points come first in the training sequence. A simple improvement is the *weighted perceptron*: training proceeds as before, but the weight vector $w$ is saved after each update. After training, instead of the final $w$, the average of all saved $w$ is taken to be the learned weight vector. Report 10-fold CV accuracy for this variant and compare it to the simple perceptron's.**

**Solution:**

**You should have gotten something like this:**

**b. mean accuracy:** 0.965, **confidence interval:** (0.951217, 0.978783).

**c. 30 mis-classifications in the 20th iteration. (Note that using the trained weights *after* the 20th iteration results in only around 24 mis-classifications.) When running with 200 iterations, still more than 20 mis-classifications occur, so the data is unlikely to be linearly separable as otherwise the training error would become zero after many enough iterations.**

**d. Perceptron:**
**mean accuracy** = 0.956, **95% confidence interval:** (0.940618, 0.971382).

**e. Weighted perceptron:**
**mean accuracy** = 0.968, **95% confidence interval:** (0.954800, 0.981200).

**5.** **([Multi-class] Logistic Regression: application to hand-written digit recognition)**

● ○ ⋆ ***CMU, 2014 fall, W. Cohen, Z. Bar-Joseph, HW3, pr. 1***
***CMU, 2011 spring, Tom Mitchell, HW3, pr. 2***

**A. In this part of the exercise you will implement the two class Logistic Regression classifier and evaluate its performance on digit recognition.**

**The *dataset* we are using for this assignment is a subset of the MNIST handwritten digit database,[5] which is a set of 70,000 28 × 28 handwritten digits from a mixture of high school students and government Census Bureau employees.**

**Your *goal* will be to write a logistic regression classifier to distinguish between a collection of 4s and 7s, of which you can see some examples in the nearby figure.**

**The data is given to you in the form of a design matrix $X$ and a vector $y$ of labels indicating the class. There are two design matrices, one for training and one for evaluation.**

**The design matrix is of size $m \times n$, where $m$ is the number of examples and $n$ is the number of features. We will treat each pixel as a feature, giving us $n = 28 \times 28 = 784$.**

**Given a set of training points $x_1, x_2, \ldots, x_m$ and a set of labels $y_1, \ldots, y_m$ we want to estimate the parameters of the model $w$. We can do this by maximizing the log-likelihood function.[6]**

**Given the *sigmoid / logistic* function,**

$$\sigma(x) = \frac{1}{1+e^{-x}},$$

**the *cost function* and its *gradient* are**

$$\begin{aligned} J(w) &= \frac{\lambda}{2}||w||_2^2 - \sum_{i=1}^{m} y_i \log \sigma(w^\top x_i) + (1-y_i)\log(1-\sigma(w^\top x_i)) \\ \nabla J(w) &= \lambda w - \sum_{i=1}^{m} (y_i - \sigma(w^\top x_i))x_i \end{aligned}$$

***Note* (1): The cost function contains the *regularization term*, $\frac{\lambda}{2}||w||_2^2$. Regularization forces the parameters of the model to be pushed towards zero by penalizing large $w$ values. This helps to prevent *overfitting* and also makes**

[5] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, *Gradient-based learning applied to document recognition*. Proceedings of the IEEE 86, 11 (Nov 1998), pp. 2278–2324.

[6] We derived the cost function for logistic regression in class, so we will not repeat the derivation here.

Tabela 1: **Summary of notation used for Logistic Regression**

| Not. | Meaning | Type |
|---|---|---|
| $m$ | **number of training examples** | **scalar** |
| $n$ | **number of features** | **scalar** |
| $x_i$ | **$i$th augmented training data point (one digit example)** | $(n+1) \times 1$ |
| $X$ | **design matrix (all training examples)** | $m \times (n+1)$ |
| $y_i$ | **$i$th training label (is the digit a 7?)** | $\{0, 1\}$ |
| $Y$ **or** $y$ | **all training labels** | $m \times 1$ |
| $w$ | **parameter vector** | $(n+1) \times 1$ |
| $S$ | **sigmoid function,** $S(t) = (1 + e^{-t})^{-1}$ | $\dim(t) \to \dim(t)$ |
| $J$ | **cost (loss) function** | $\mathbb{R}^n \to \mathbb{R}$ |
| $\nabla J$ | **gradient of $J$ (vector of derivatives in each dimension)** | $\mathbb{R}^n \to \mathbb{R}^n$ |
| $\alpha$ | **(parameter) gradient descent learning rate** | **scalar** |
| $d$ | **(parameter) decay constant for $\alpha$ to decrease by every iteration** | **scalar** |
| $\lambda$ | **(parameter) regularization strength** | **scalar** |

**the objective function strictly concave, which means that there is a unique solution.**

***Note* (2): Please regularize the *intercept term* too, i.e., $w^{(0)}$ should also be regularized.[7] In order to keep the notation clean and make the implementation easier, we assume that each $x_i$ has been augmented with an extra $1$ at the beginning, i.e., $x_i' = [1; x_i]$. Therefore our *model* of the log-odds is**

$$\log \frac{1-p}{p} = w^{(0)} + w^{(1)}x^{(1)} + \ldots + w^{(n)}x^{(n)}.$$

***Note* (3): For models such as linear regression we were able to find a closed form solution for the parameters of the model. Unfortunately, for many machine learning models, including Logistic Regression, no such closed form solutions exist. Therefore we will use a *gradient-based method* to find our parameters.**

**The *update rule* for gradient ascent is**

$$w_{i+1} = w_i + \alpha d^i \nabla J(w_i),$$

**where $\alpha$ specifies the *learning rate*, or how large a step we wish to take, and $d$ is a *decay term* that we use to ensure that the step sizes we make will gradually get smaller, so long as we converge. The iteration stops when the change of $x$ or $f(x)$ is smaller than a threshold.**

**a. Implement the cost function and the gradient for logistic regression in `costLR.m`.[8] Implement gradient descent in `minimize.m`. Use your minimizer to complete `trainLR.m`.**

[7] Many resources about Logistic Regression on the web do not regularize the intercept term, so be aware if you see different objective functions.

[8] You can run `run_logit.m` to check whether your gradients match the cost. The script should pass the gradient checker and then stop.

**b. Once you have trained the model, you can then use it to make predictions. Implement `predictLR`, which will generate the most likely classes for a given $x_i$.**

**B. In this part of the exercise you will implement the multi-class class Logistic Regression classifier and evaluate its performance on another digit recognition, provided by USPS. In this dataset, each hand-written digital image is 16 by 16 pixels. If we treat the value of each pixel as a boolean feature (either 0 for black or 1 for white), then each example has $16 \times 16 = 256$ $\{0, 1\}$-valued features, and hence $x$ has 256 dimension. Each digit (i.e., 1,2,3,4,5,6,7,8,9,0) corresponds to a class label $y$ ($y = 1, \ldots, K,\ K = 10$). For each digit, we have 600 training samples and 500 testing samples.[9]**

**Please download the data from the website. Load the usps `digital.mat` file in `usps_digital.zip` into Matlab. You will have four matrices:**

- **`tr_X`: training input matrix with the dimension $6000 \times 256$.**
- **`tr_y`: training label of the length 6000, each element is from 1 to 10.**
- **`te_X`: testing input matrix with the dimension $5000 \times 256$.**
- **`te_y`: testing label of the length 5000, each element is from 1 to 10.**

**For those who do NOT want to use Matlab, we also provide the text file for these four matrices in `usps_digital.zip`. Note that if you want to view the image of a particular training/testing example in Matlab, say the 1000th training example, you may use the following Matlab command: `imshow(reshape(tr_X(1000,:),16,16))`.**

**c. Use the gradient ascent algorithm to train a multi-class logistic regression classifier. Plot (1) the objective value (log-likelihood), (2) the training accuracy, and (3) the testing accuracy versus the number of iterations. Report your final testing accuracy, i.e., the fraction of test images that are correctly classified.**

**Note that you must choose a suitable learning rate (i.e. stepsize) of the gradient ascent algorithm. A *hint* is that your learning rate cannot be too large otherwise your objective will increase only for the first few iterations.**

**In addition, you need to choose a suitable stopping criterion. You might use the number of iterations, the decrease of the objective value, or the maximum of the L2 norms of the gradient with respect to each $w_k$. Or you might watch the increase of the testing accuracy and stop the optimization when the accuracy is stable.**

**d. Now we add the regularization term $\frac{\lambda}{2}\sum_{i=1}^{K-1}||w_l||_2^2$. For $\lambda = 1, 10, 100, 1000$, report the final testing accuracies.**

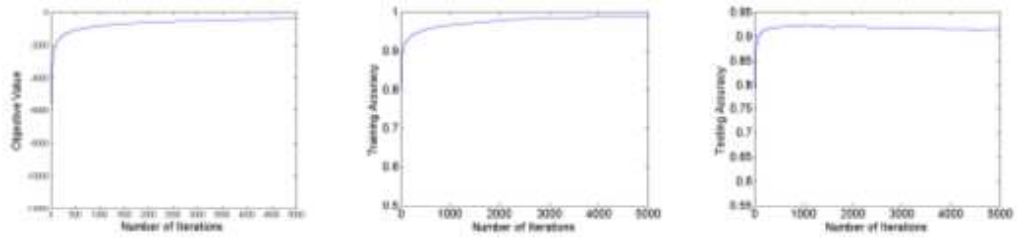
**e. What can you conclude from the above experiment? (Hint: the relationship between the regularization weight and the prediction performance.)**

**Solution:**

**b. You should get about 96% accuracy.**

[9] You can view these images at http://www.cs.nyu.edu/~roweis/data/usps_0.jpg, ..., http://www.cs.nyu.edu/~roweis/data/usps_9.jpg.

**c. I use the stepsize $\eta = 0.0001$ and run the gradient ascent method for 5000 iterations. The objective value vs. the number of iterations, training error vs. the number of iterations, testing error vs. the number of iterations are presented in figure below:**




**d. For $\lambda = 0, 1, 10, 100, 1000$, the comparison of the testing accuracy is presented in the next table:**

| $\lambda$ | 0 | 1 | 10 | 100 | 1000 |
|---|---|---|---|---|---|
| Testing accuracy | 91.44% | 91.58% | 91.92% | 89.74% | 79.78% |

**e. From the above result, we can see that adding the regularization could avoid overfitting and lead to better generalization performance (e.g., $\lambda = 1, 10$). However, the regularization cannot be too large. Although a larger regularization can decrease the variance, it introduces additional *bias* and may lead to worse generalization performance.**

**6.** **(Multinomial/Categorical Logistic Regression, Gaussian Naive Bayes, Gaussian Joint Bayes, and $k$-NN: application on the ORL Faces dataset)**

• · ***CMU 2010 spring, E. Xing, T. Mitchell, A. Singh, HW2, pr. 2***

**In this part, you are going to play with *The ORL Database of Faces*.**

*6 sample images from two persons*

**Each image is 92 by 112 pixels. If we treat the luminance of each pixel as a feature, each sample has $92 * 112 = 10304$ real value features, which can be written as a random vector $X$. We will treat each person as a class $Y$ ($Y = 1, \ldots, K,\ K = 10$). We use $X_i$ to refer the $i$-th feature. Given a set of training data $D = \{(y^l, x^l)\}$, we will train different classification models to classify images to their person id's. To simplify notation, we will use $P(y|x)$ in place of $P(Y = y|X = x)$.**

**We will select our models by 10-fold cross validation: partition the data for each face into 10 mutually exclusive sets (folds). In our case, exactly one image for each fold. Then, for $k = 1, \ldots, 10$, leave out the data from fold $k$ for all faces, train on the rest, and test on the left out data. Average the results of these 10 tests to estimate the training accuracy of your classifier.**

***Note*: Beware that we are actually not evaluating the *generalization errors* of the classifier here. When evaluating generalization error, we would need an independent test set that is not at all touched during the whole developing and tuning process.**

**For your convenience, a piece of code `loadFaces.m` is provided to help loading images as feature vectors.**

**From Tom Mitchell's additional book chapter,[10] page 13, you will see *a generalization of logistic regression*, which allows $Y$ to have more than two possible values.**

**a. Write down the objective function, and the first order derivatives of the *multinomial logistic regression model* (which is a binary classifier).[11]**

**Here we will consider a L2-norm *regularized objective function* (with a term $\lambda|\theta|_2$).**

**b. Implement the logistic regression model with *gradient ascent*. Show your evaluation result here. Use regularization parameter $\lambda = 0$.**

[10] www.cs.cmu.edu/∼tom/mlbook/NBayesLogReg.pdf.

[11] *Hint*: In order to do $k$-class classification with binary classifier, we use a voting scheme. At training time, a classifier is trained for any pair of classes. At testing time, all $k(k-1)/2$ classifiers are applied to the testing sample. Each classifier either vote for its first class or its second class. The class voted by most number of classifiers is chosen as the prediction.

***Hint*: The gradient ascent method (also known as *steepest ascent*) is a first-order optimization algorithm. It optimizes a function $f(x)$ by**

$$x_{t+1} = x_t + \alpha_t f'(x_t),$$

**where $\alpha_t$ is called the *step size*, which is often picked by *line search*. For example, we can initialize $\alpha_t = 1.0$. Then set $\alpha_t = \alpha_t/2$ while $f(x_t + \alpha_t f'(x_t)) < f(x_t)$. The iteration stops when the change of $x$ or $f(x)$ is smaller than a threshold.**

***Hint*: If the training time of your model is too long, you can consider use just a subset of the features (e.g., in Matlab `X = X(:,1:100:d)`).**

**c. Overfitting and Regularization**

**Now we test how *regularization* can help prevent *overfitting*. During cross-validation, let's use $m$ images from each person for training, and the rest for testing. Report your cross-validated result with varying $m = 1, \ldots, 9$ and varying regularization parameter $\lambda$.**

**d. Logistic Regression and Newton's method**

**Newton's method (also known as the Newton-Raphson method) is a first-order optimization algorithm, which often converges in a few iterations. It Optimize a function $f(x)$ by the update equation**

$$x_{t+1} = x_t - \frac{f'(x_t)}{f''(x_t)}$$

**The iteration stops when the change of $x$ or $f(x)$ is smaller than a threshold. Write down the second order derivatives and the update equation of the logistic regression model.**

**Implement the logistic regression model with Newton's method. Show your evaluation result here.**

**B. Implement the $k$-NN algorithm. Use L2-norm as the distance metric. Show your evaluation result here, and compare different values of $k$.**

**C. Conditional Gaussian Estimation**

**For a Gaussian model we have**

$$P(y|x) = \frac{P(x|y)P(y)}{P(x)},$$

**where**

$$P(x|y) = \frac{1}{(2\pi)^{d/2}|\Sigma_y|^{1/2}} \exp(-(x-\mu_y)^\top \Sigma^{-1}(x-\mu_y)/2),$$

**and $P(y) = \pi_y$. Please write down the MLE estimation of model parameters $\Sigma_y$, $\mu_y$, and $\pi_y$. Here we do not assume that $X_i$ are independent given $Y$.**

**D. Gaussian Naive Bayes is a form of Guassian model with assumption that $X_i$ are independent given $Y$. Implement the Gaussian NB model, and briefly describe your evaluation result.**

**E. Compare the above methods by training/testing time, and accuracy. Which method do you prefer?**

**7.** **(Model selection: sentiment analysis for music reviews using a dataset provided by Amazon)**

• · ***CMU, 2014 spring, B. Poczos, A. Singh, HW2, pr. 5***

**In this homework, you will perform model selection on a sentiment analysis dataset of music reviews.[12] The *dataset* consists of reviews from Amazon.com for musics. The ratings have been converted to a binary label, indicating a *negative review* or a *positive review*. We will use *lasso logistic regression* for this problem.[13] The lasso logistic regression *objective function* to minimize during training is:**

$$\mathcal{L}(\beta) = \log(1 + \exp(-y\beta^\top x)) + \lambda\|\beta\|_1$$

**In lasso logistic regression, we penalize the loss function by an $L_1$ norm of the feature coefficients. Penalization with an $L_1$ norm tends to produce solutions where some coefficients are exactly 0. This makes it attractive for high-dimensional data such as text, because in most cases most words can typically be ignored. Furthermore, since we are often left with only a few nonzero coefficients, the lasso solution is often easy to interpret.**

**The *goal* of *model selection* here is to choose $\lambda$ since each setting of $\lambda$ implies a different *model size* (number of non-zero coefficients).**

**You do not need to implement lasso logistic regression. You can *download* an implementation from** https://github.com/redpony/creg**, and the dataset can be found on the course web page. There are three *feature files* and three *response (label) files* (all response files end with `.res`). They are already in the format required by the implementation you will use. The files are:**

- **Training data:** `music.train` **and** `music.train.res`
- **Development data:** `music.dev` **and** `music.dev.res`
- **Test data:** `music.test` **and** `music.test.res`

***Important note*: The code outputs *accuracy*, whereas you need to plot *classification error* here. You can simply transform accuracy to error by using** $1 -$ **accuracy.**

**Error on development (validation) data**

**In the first part of the problem, we will use error on a development data to choose $\lambda$. Run the model with $\lambda = \{10^{-8}, 10^{-7}, 10^{-6}, \ldots, 10^{-1}, 1, 10, 100\}$.**

**a. Plot the error on training data and development data as a function of $\log \lambda$.**

**b. Plot the model size (number of nonzero coefficients) on development data as a function of $\log \lambda$.**

**c. Choose $\lambda$ that gives the lowest error on development data. Run it on the test data and report the test error.**

**Briefly discuss all the results.**

[12] John Blitzer, Mark Dredze, and Fernando Pereira. Biographies, Bollywood, Boom-boxes and Blenders: Domain Adaptation for Sentiment Classification. In *Proceedings of ACL*, 2007.

[13] Robert Tibshirani. Regression shrinkage and selection via the lasso, In *Journal of Royal Statistical Society B*, 58(1):267:288, 1996.

**Model Complexity and Bias-Variance Tradeoff**

**d. Give a high-level explanation on the relation between $\lambda$ and the *bias* and *variance* of parameter estimates $\hat{\beta}$. Does larger $\lambda$ correspond to higher or lower bias? What about the variance? Does larger $\lambda$ lead to a more complex or a less complex model?**

**Resolving a tie**

**e. If there are more than one $\lambda$ that minimizes the error on the development data, which one will you pick? Explain your choice.**

**Random search**

**f. An alternative way to search $\lambda$ is by randomly sampling its value from an interval.**

*i.* **Sample eleven random values *log* uniformly from an interval $[10^{-8}, 100]$ for $\lambda$ and train a lasso logistic regression model. Plot the error on development data as a function of $\log \lambda$.**

*ii.* **Choose $\lambda$ that gives the lowest error on development data. Run it on the test data and report the test error.**

**Random vs. grid search**

**g. Which one do you think is a better method for searching values to try for $\lambda$? Why?**

**8.** **(Gradient descent: comparison with another training algorithm on a function approximation task)**

• ***CMU, 2004 fall, Carlos Guestrin, HW4, pr. 2***

**In this problem you'll compare the Gradient Descent training algorithm with one other training algorithm of your choosing, on a particular function approximation problem. For this problem, the idea is to familiarize yourself with Gradient Descent and at least one other numerical solution technique.**

**The dataset `data.txt` contains a series of $(x, y)$ records, where $x \in [0, 5]$ and $y$ is a function of $x$ given by $y = a\sin(bx) + w$, where $a$ and $b$ are parameters to be learned and $w$ is a noise term such that $w \sim N(0, \sigma^2)$. We want to learn from the data the best values of $a$ and $b$ to minimize the sum of squared error:**

$$\arg\min_{a,b} \sum_{i=1}^{n} (y_i - a\sin(bx_i))^2.$$

**Use any programming language of your choice and implement two training techniques to learn these parameters. The first technique should be Gradient Descent with a fixed learning rate, as discussed in class. The second can be any of the other numerical solutions listed in class: Levenberg-Marquardt, Newton's Method, Conjugate Gradient, Gradient Descent with dynamic learning rate and/or momentum considerations, or one of your own choice not mentioned in class.**

**You may want to look at a scatterplot of the data to get rough initial values for the parameters $a$ and $b$. If you are getting a large sum of squared error after convergence (where large means $> 100$), you may want to try random restarts.**

**Write a short report detailing the method you chose and its relative performance in comparison to standard Gradient Descent (report the final solution obtained (values of $a$ and $b$) and some measure of the computation required to reach it and/or the resistance of the approach to local minima). If possible, explain the difference in performance based on the algorithmic difference between the two approaches you implemented and the function being learned.**

# 2 Decision Trees

**9.** **(Decision trees: analysing the relationship between the dataset size and model complexity)**

● ○ ***CMU, 2012 fall, T. Mitchell, Z. Bar-Joseph, HW1, pr. 2.e***

**Here we will use a synthetic dataset generated by the following algorithm: To generate an $(x, y)$ pair, first, six binary valued $x_1, \ldots, x_6$ are randomly generated, each independently with probability 0.5. This six-tuple is our $x$. Then, to generate the corresponding $y$ value:**

$$\begin{aligned} f(x) &= x_1 \vee (\neg x_1 \wedge x_2 \wedge x_6) \\ y &= \begin{cases} f(x) \text{ with probability } \theta, \\ \text{else } (1 - f(x)). \end{cases} \end{aligned}$$

**So $Y$ is a possibly corrupted version of $f(X)$, where the parameter $\theta$ controls the noisiness. ($\theta = 1$ is noise-free. $\theta = 0.51$ is very noisy.) Get *code* and *test data* from ... .**

**We will experimentally investigate the relationships between *model complexity*, *training size*, and *classifier accuracy*.**

**We provide a Matlab implementation of ID3, without pruning, but featuring a maxdepth parameter: *traintree(trainX, trainY, maxdepth)*. It returns an object representing the classifier, which can be viewed with *printtree(tree)*. Classify new data via *classifywithtree(tree, testX)*. We also provide the simulation function to generate the synthetic data: *generatedata(N, theta)*, that you can use to create training data. Finally, there is a fixed test set for all experiments (generated using $\theta = 0.9$). See `tt1.m` for sample code to get started. Include printouts of your code and graphs.**

**a. For a depth = 3 decision tree learner, learn classifiers for training sets size 10 and 100 (generate using $\theta = 0.9$). At each size, report training and test accuracies.**

**b. Let's track the learning curves for *simple* versus *complex classifiers*. For maxdepth = 1 and maxdepth = 3, perform the following experiment: For each training set size $\{2^1, 2^2, \ldots, 2^{10}\}$, generate a training set, fit a tree, and record the train and test accuracies. For each (depth, trainsize) combination, average the results over 20 different simulated training sets. Make three learning curve plots, where the horizontal axis is training size, and vertical axis is accuracy. First, plot the two testing accuracy curves, for each maxdepth setting, on the same graph. For the second and third graphs, have one for each maxdepth setting, and on each plot its training and testing accuracy curves. Place the graphs side-by-side, with identical axis scales. It may be helpful to use a log-scale for data size.**

**Next, answer several questions with no more than three sentences each:**

**c. When is the simpler model better? When is the more complex model better?**

**d. When are train and test accuracies different? If you're experimenting in the real world and find that train and test accuracies are substantially different, what should you do?**

**e. For a particular maxdepth, why do train and test accuracies converge to the same place? Comparing different maxdepths, why do test accuracies converge to different places? Why does it take smaller or larger amounts of data to do so?**

**f. For maxdepths 1 and 3, repeat the same vary-the-training-size experiment with $\theta = 0.6$ for the training data. Show the graphs. Compare to the previous ones: what is the effect of noisier data?**

**Solution:**

**a.**

**b. Pink stars: depth = 3. Black x's: depth = 1. Blue circles: training accuracy. Red squares: testing accuracy.**

**c. It is good to have high model complexity when there is lots of training data. When there is little training data, the simpler model is better.**

**d. They're different when you're overfitting. If this is happening you have two options: (1) decrease your model complexity, or (2) get more data.**

**e. They converge when the algorithm is learning the best possible model from the model class prescribed by maxdepth: this gets same accuracy on the training and test sets. (2) The higher complexity (maxdepth=3) model class learns the underlying function better, thus gets better accuracy. But, (3) the higher complexity model class has more parameters to learn, and thus takes more data to get to this point.**

**f. It's much harder for the complex model to do better. Also, it takes much longer for all test curves to converge. (Train/test curves don't converge to same place because noise levels are different.)**

**Note: Colors and styles are the same as for previous plots. These plots test all the way up to $2^{15}$ training examples: you can see where they converge to, which is not completely clear with only $2^{10}$ examples.**

**10.** **(Decision trees: experiment with an ID3 implementation (in C))**

● ○ ***CMU, 2012 spring, Roni Rosenfeld, HW3***

**This exercise gives you the opportunity to experiment with a decision tree learning program. You are first asked to experiment with the simple Play-Tennis data described in Chapter 3 of Tom Mitchell's *Machine Learning* book, and then to experiment with a considerably larger data set.**

**We provide most of the decision tree code. You will have to complete the code, test, and prune a decision tree based on the ID3 algorithm described in Chapter 3 of the textbook. You can obtain it as a gzipped archive from . . . .**

**To unzip and get started on a Linux/Mac machine, do the following:**
**1. Download the `hw3.tgz` file to your working directory.**
**2. Issue the command `tar -zxvf hw3.tgz` to unzip and untar the file. This will create a subdirectory `hw3` in the current directory.**
**3. Type `make` to compile and you are ready to go. The executable is called `dt`. There is a help file named `README.dt` which contains instructions on how to run the `dt` program.**

**If you work from a Windows machine, you can install Cygwin and that should give you a Linux environment. Remember to install the "Devel" category to get `gcc`. Depending on your machine some tweaks might be needed to make it work.**

**A. Play Tennis Data**

**The training data from Table 3.2 in the textbook is available in the file `tennis.ssv`. Notice that it contains the fourteen training examples repeated twice. For question A.2, please use it as given (with the 28 training examples). For question A.4, you will need to extract the fourteen *unique* training examples and use those in addition to the ones you invent.**

**A1. If you try running the code now it will not work because the function that calculates the entropy has not been implemented. (Remember that entropy is required in turn to compute the information gain.) It is your job to complete it. You will have to make your changes in the file `entropy.c`. After you correctly implement the entropy calculation, the program will produce the decision tree shown in Figure 3.1 of the textbook when run on `tennis.ssv` (with all examples used for training).**

***Hint*: When you implement the entropy function, be sure to deal with casts from `int` to `double` correctly. Note that `num_pos/num_total = 0` if `num_pos` and `num_total` are both `int`'s. You must do `((double)num_pos)/num_total` to get the desired result or, alternately, define `num_total` as a `double`.**

**A2. Try running the program a few times with half of the data set for training and the other half for testing (no pruning). Print out your command for running the program. Do you get a different result each time? Why? Report the average accuracy of 10 runs on both training and test data (use the `batch` option of `dt`). For this question please use `tennis.ssv` as given.**

**A3. If we add the following examples:**

| | | | | |
|---|---|---|---|---|
| 0 | sunny | hot | high | weak |
| 1 | sunny | cool | normal | weak |
| 0 | sunny | mild | high | weak |
| 0 | rain | mild | high | strong |
| 1 | sunny | mild | normal | strong |

**to the original `tennis.ssv` that has 28 examples, which attribute would be selected at the root node? Compute the information gain for each attribute and find out the max. Show how you calculate the gains.**

**A4. By now, you should be able to teach ID3 the concept represented by Figure 3.1; we call this the *correct concept*. If an example is correctly labeled by the correct concept, we call the example a *correct example*. For this question, you will need to extract the fourteen *unique* examples from `tennis.ssv`. In each of the questions below, you will add some your own examples to the original fourteen, and use all of them for training (You will not use a testing or pruning set here.). Turn in your datasets (named according to the *Hand-in* section at the end).**

**a. Duplicate some of the fourteen training examples in order to get ID3 to learn a different decision tree.**

**b. Add new *correct* samples to the original fourteen samples in order to get ID3 to include the attribute "temperature" in the tree.**

**A5. Use the fourteen unique examples from `tennis.ssv`. Run ID3 once using all fourteen for training, and note the structure (which is in Figure 3.1). Now, try flipping the label (0 to 1, or 1 to 0) of any one example, and run ID3 again. Note the structure of the new tree.**

**Now change the example's label back to correct. Do the same with another four samples. (Flip one label, run ID3, and then flip the label back.) Give some general observations about the structures (differences and similarities with the original tree) ID3 learns on these slightly *noisy* datasets.**

**B. Agaricus-Lepiota Data Set**

**The file `mushroom.ssv` contains records drawn from The Audobon Society Field Guide to North American Mushrooms (1981), G. H. Lincoff (Pres.), New York. Alfred A. Knopf, posted [LC: it on the] UCI Machine Learning Repository. Each record contains information about a mushroom and whether or not it is poisonous.**

**B1. First we will look at how the quality of the learned hypothesis varies with the size of the training set. Run the program with training sets of size 10%, 30%, 50%, 70%, and 90%, using 10% to test each time. Please run a particular size at least 10 times. You may want to use the batch mode option provided by `dt`.**

**Construct a graph with training set size on the x-axis and test set accuracy on the y-axis. Remember to place *errorbars* on each point extending one standard deviation above and below the point. You can do this in Matlab, Mathematica, GNUplot or by hand.**

**If you use `gnuplot`:**
**1. Create a file `data.txt` with your results with the following format:**
**2. Each line has `<training size> <accuracy> <standard deviation>`.**

**3. Type `gnuplot` to get to the gnuplot command prompt. At the prompt, type set terminal postscript followed by `set output graph.ps` and finally `plot data.txt with errorbars` to plot the graph.**

**B2. Now repeat the experiment, but with a *noisy* dataset, `noisy10.ssv`, in which each label in has been flipped with a chance of 10%. Run the program with training sizes from 30% to 70% at the step of 5% (9 sizes in total), using 10% at each step to test and at least 10 trials for each size. Plot the graph of test accuracy and compare it with the one from B.1. In addition, plot the number of nodes in the result trees against the training %. Note that the training accuracy decreases slightly after a certain point. You may also observe dips in the test accuracy. What could be causing this?**

**B3. One way to battle these phenomena is with *pruning*. For this question, you will need to complete the implementation of pruning that has been provided. As it stands, the pruning function considers only the root of the tree and does not recursively descend to the sub-trees. You will have to fix this by implementing the recursive call in `PruneDecisionTree()` (in `prune-dt.c`). Recall that pruning traverses the tree removing nodes which do not help classification over the validation set. Note that pruning a node entails removing the sub-tree(s) below a node, and not the node itself.**

**In order to implement the recursive call, you will need to familiarize yourself with the trees representation in C. In particular, how to get at the children of a node. Look at `dt.h` for details. A decision you will make is when to prune a sub-tree, before or after pruning the current node. Bottom-Up pruning is when you prune the subtree of a node, before considering the node as a pruning candidate. Top-down pruning is when you first consider the node as a pruning candidate and only prune the subtree should you decide not to eliminate the node. Please do NOT mix the two up. If you are in doubt, consult the book.**

**Write out on paper the code that you would need to add for both *bottom-up* and *top-down* pruning. Implement only the bottom-up code and repeat the experiments in B.2 using 20% of the data for pruning at each trial. Plot the graph of test accuracy and number of nodes and comment on the differences from B.2.**

**B4. Answer the following questions with explanation.**

**a. Which pruning strategy would make more *calls*? top-down or bottom-up? or is it data-dependent?**

**b. Which pruning strategy would result in better accuracy on the *pruning set*? top-down or bottom-up? or is it data-dependent?**

**c. Which pruning strategy would result in better accuracy on the *testing set*? top-down or bottom-up? or is it data-dependent?**

**Hand-In Instructions**

**Besides your assignment write-up, here is the additional materials you will need to hand in. Your write-up should include the graphs asked for.**

**A.1. Hand in your modified entropy.c.**
**A.2. Nothing to hand in.**
**A.3. Nothing to hand in.**

**A.4. For part $a$, hand in `tennis1.4.a.ssv`, which should contain the original data plus the samples you created appended at the end of the file. Likewise for $b$.**

**B.1. Nothing to hand in.**
**B.2. Nothing to hand in.**
**B.3. Hand in your modified `prune-dt.c`.**
**B.4. Nothing to hand in.**

**Hints:**

**– If you are unsure about your answer, play with code to see if you can experimentally verify your intuitions.**

**– It is very helpful to include explanations with your examples, or at least mention how you constructed the example, what was the reasoning behind your choices, etc.**

**– Please label the axes and specify what accuracy/performance metric you are measuring and on what dataset: e.g. training, testing, validation, noisy10 etc.**

**Solution:**

**A1. The Entropy function should be like:**

```
double Entropy( int num_pos, int num_neg )
{
    if (num_pos == 0 || num_neg == 0)
    return 0.0;

    double entropy = 0.0;
    double total = (double) (num_pos + num_neg);

    entropy = - (num_pos / total) * LogBase2( num_pos / total )
        - (num_neg / total) * LogBase2( num_neg / total );

    return entropy;
}
```

**A2. The command is: `./dt 0.5 0 0.5 tennis.ssv`**
**The results are different, because the code randomly splits the data, and each time a different training set is used.**
**The training accuracy is always 100%. The testing accuracy should be around 82%.**

**A3. The data set now has 13 negatives and 20 positives. So the overall entropy is: 0.967.**
**Using "outlook", the information gain is: 0.218.**
**Using "temperature", the information gain is: 0.047.**
**Using "humidity", the information gain is: 0.221.**
**Using "wind", the information gain is: 0.025.**
**Therefore, "humidity" should be selected.**

**A4.**

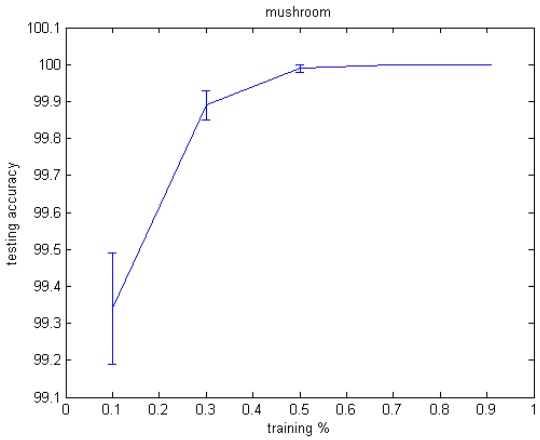
**a. Examples given in question A.3 are actually duplicates that changed the tree.**

**b. The idea is to let "temperature" determine "Play-Tennis". For example, we can add the following:**

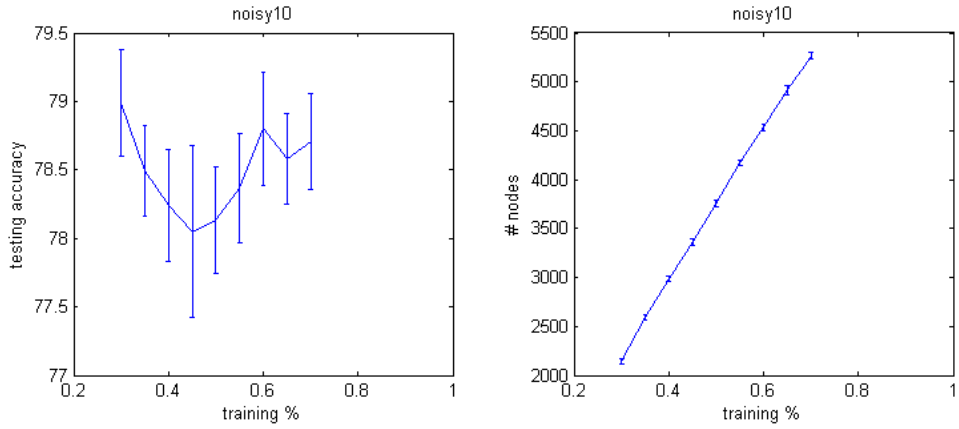
| | | | | |
|---|---|---|---|---|
| 1 | sunny | cool | normal | weak |
| 1 | overcast | cool | normal | weak |
| 1 | rain | cool | normal | weak |
| 1 | sunny | cool | normal | weak |
| 0 | rain | hot | normal | strong |
| 0 | sunny | hot | high | weak |
| 0 | sunny | hot | high | weak |
| 0 | rain | hot | normal | strong |

**A5. Generally, noisy data sets produce bigger trees. However the rules implied by these trees are quite stable. Some trees may have the same top structure as the true structure. These overall similarities to the true structure give some intuition for why *pruning* helps; pruning can cut away the extra subtrees which model small effects which might be from *noise*.**

**B1. I ran each size 20 times, and got a graph like this:**




**B2.**



**This decrease in testing accuracy with the larger training may be caused by a form of overfitting; that is, the algorithm tries to perfectly match the data in the training set, including the noise, and as a result the complexity of the learned tree increases very rapidly as the number of training examples increases.**

**Note that this is not the usual sense of overfitting, since typically overfitting is more of a problem when the number of training examples is small. However, here we also have the problem that the complexity of the hypothesis space is an increasing function of the number of training examples. See how the number of nodes grows.**

**There are also "dips" in the accuracy on the test set, a point where the accuracy decreased before increasing again. This is because of more complex concepts; there are always two competing forces here: the information content of the training data, which increases with the number of training examples and pushes toward higher accuracies, and the complexity of the hypothesis space, which gets worse as the number of training examples increases.**

**You may also notice that the training accuracy slightly decreases as the size of the training set grows. This seems to be purely due to the noisy labels, which makes it impossible to construct a consistent tree, and the more pairs of examples you have in the training set that have contradicting labels, the worse will be the training error.**

**B3.**

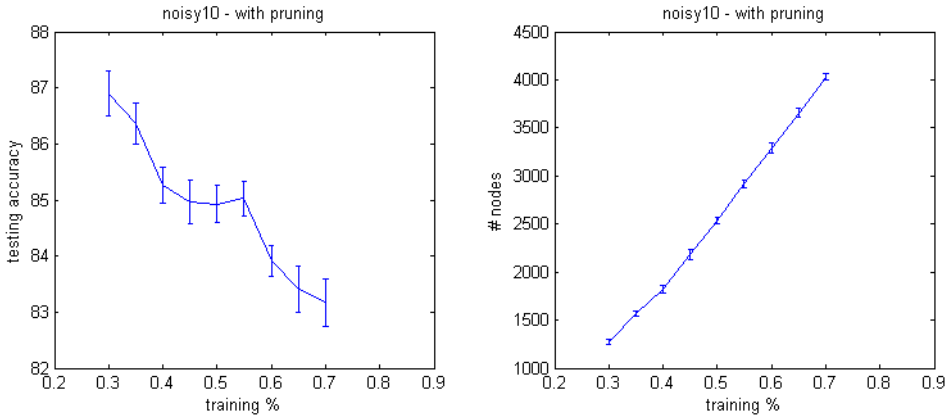
**For bottom-up pruning, add to the beginning of the function:**

```
/*****************************************************************
   You could insert the recursive call BEFORE you check the node
*****************************************************************/
for (i = 0 ; i < node->num_children ; i++)
   PruneDecisionTree(root, node->children[i], data, num_data, pruning_set, num_prune, ssvinfo);
```

**For top-down pruning, add to the end of the function:**

```
/*****************************************************************
   Or you could do the recursive call AFTER you check the node
   (given that you decided to keep it)
*****************************************************************/
for (i = 0 ; i < node->num_children ; i++)
   PruneDecisionTree(root, node->children[i], data, num_data, pruning_set, num_prune, ssvinfo);
```

**By running each size 20 times I got a graph like this:**




**B4.**

**a. Bottom-up. Bottom-up pruning examines all the nodes. Top-down pruning may eliminate a subtree without examining the nodes in the subtree, leading to fewer calls than bottom-up.**

**b. Bottom-up. By the property of the algorithm, bottom-up pruning returns the tree with the LOWEST POSSIBLE ERROR over the pruning set. Since top-down can aggressively eliminate subtree's without considering each of the nodes in the subtree, it could return a non-optimal tree (over the pruning set that is). Keep in mind that the function used to decide whether a node should be removed or not is the same for both BU and TD and only the search strategy differs.**

**c. Data depedent. If the test set is very different from the training set, a shorter tree yielded by top-down pruning may perform better, because of its potentially better generalization power.**

**11.** **(ID3 with continuous attributes: experiment with a Matlab implementation on the Breast Cancer dataset)**

• ○ ***CMU, 2011 fall, T. Mitchell, A. Singh, HW1, pr. 2***

**One very interesting application area of machine learning is in making medical diagnoses. In this problem you will train and test a binary decision tree to detect breast cancer using real world data. *You may use any programming language you like.***

**The Dataset**

**We will use the Wisconsin Diagnostic Breast Cancer (WDBC) dataset[14] The dataset consists of 569 samples of biopsied tissue. The tissue for each sample is imaged and 10 characteristics of the nuclei of cells present in each image are characterized. These characteristics are**

**(a) Radius**
**(b) Texture**
**(c) Perimeter**
**(d) Area**
**(e) Smoothness**
**(f) Compactness**
**(g) Concavity**
**(h) Number of concave portions of contour**
**(i) Symmetry**
**(j) Fractal dimension**

**Each of the 569 samples used in the dataset consists of a feature vector of length 30. The first 10 entries in this feature vector are the *mean* of the characteristics listed above for each image. The second 10 are the *standard deviation* and last 10 are the largest value of each of these characteristics present in each image.**

**Each sample is also associated with a label. A label of value 1 indicates the sample was for malignant (cancerous) tissue. A label of value 0 indicates the sample was for benign tissue.**

**This dataset has already been broken up into training, validation and test sets for you and is available in the compressed archive for this problem on the class website. The names of the files are `trainX.csv`, `trainY.csv`, `validationX.csv`, `validationY.csv`, `testX.csv` and `testY.csv`. The file names ending in `X.csv` contain feature vectors and those ending in `Y.csv` contain labels. Each file is in *comma separated value* format where each row represents a sample.**

**A. Programming**

**A1. Learning a binary decision tree**

**As discussed in class and the reading material, to learn a binary decision tree we must determine which feature attribute to select as well as the threshold**

[14] Original dataset available at http://archive.ics.uci.edu/ml/datasets/Breast+Cancer+Wisconsin+(Diagnostic).

**value to use in the split criterion for each non-leaf node in the tree. This can be done in a recursive manner, where we first find the optimal split for the root node using all of the training data available to us. We then split the training data according to the criterion selected for the root node, which will leave us with two subsets of the original training data. We then find the optimal split for each of these subsets of data, which gives the criterion for splitting on the second level children nodes. We recursively continue this process until the subsets of training data we are left with at a set of children nodes are pure (i.e., they contain only training examples of one class) or the feature vectors associated with a node are all identical (in which case we can not split them) but their labels are different.**

**In this problem, you will implement an algorithm to learn the structure of a tree. The optimal splits at each node should be found using the information gain criterion discussed in class.**

**While you are free to write your algorithm in any language you choose, if you use the provided Matlab code included in the compressed archive for this problem on the class website, you only need to complete one function, `computeOptimalSplit.m`. This function is currently empty and only contains comments describing how it should work. Please complete this function so that given any set of training data it finds the optimal split according to the information gain criterion.**

**Include a printout of your completed `computeOptimalSplit.m` along with any other functions you needed to write with your homework submission. If you choose to not use the provided Matlab code, please include a printout of all the code you wrote to train a binary decision tree according to the description given above.**

***Note*: While there are multiple ways to design a decision tree, in this problem we constrain ourselves to those which simply pick *one* feature attribute to split on. Further, we restrict ourselves to performing only binary splits. In other words, each split should simply determine if the value of a particular attribute in the feature vector of a sample is less than or equal to a threshold value or greater than the threshold value.**

***Note*: Please note that the feature attributes in the provided dataset are continuously valued. There are two things to keep in mind with this.**

**First, this is slightly different than working with feature values which are discrete because it is no longer possible to try splitting at every possible feature value (since there are an infinite number of possible feature values). One way of dealing with this is by recognizing that given a set of training data of $N$ points, there are only $N-1$ places we could place splits for the data (if we constrain ourselves to binary splits). Thus, the approach you should take in this function is to sort the training data by feature value and then test split values that are the mean of ordered training points. For *example*, if the points to split between were $1, 2, 3$, you would test two split values: $1.5$ and $2.5$.**

**Second, when working working with feature values that can only take on one of two values, once we split using one feature attribute, there is no point in trying to split on that feature attribute later. (Can you think of why this would be?) However, when working with continuously valued data, this is no longer the case, so your splitting algorithm should consider splitting on all feature attributes at every split.**

**A2. Pruning a binary decision tree**

**The method of learning the structure and splitting criterion for a binary decision tree described above terminates when the training examples associated with a node are all of the same class or there are no more possible splits. In general, this will lead to *overfitting*. As discussed in class, *pruning* is one method of using validation data to avoid overfitting.**

**You will implement an algorithm to use *validation data* to greedily prune a binary decision tree in an iterative manner. Specifically, the algorithm that we will implement will start with a binary decision tree and perform an exhaustive search for the single node for which removing it (and its children) produces the largest increase (or smallest decrease) in classification accuracy as measured using validation data. Once this node is identified, it and its children are removed from the tree, producing a new tree. This process is repeated, where we iteratively prune one node at a time until we are left with a tree which consists only of the root node.[15]**

**Implement a function which starts with a tree and selects the single best node to remove in order to produce the greatest increase (or smallest decrease) in classification accuracy as measured with validation data. If you are using Matlab, this means you only need to complete the empty function `pruneSingleGreedyNode.m`. Please see the comments in that function for details on what you should implement. We suggest that you make use of the provided Matlab functiona `pruneAllNodes.m` which will return a listing of all possible trees that can be formed by removing a single node from a base tree and `batchClassifyWithDT.m` which will classify a set of samples given a decision tree.**

**Please include your version of `pruneSingleGreedyNode.m` along with any other functions you needed to write with your homework. If not using Matlab, please attach the code for a function which performs the same function described for `pruneSingleGreedyNode.m`.**

**B. Data analysis**

**B1. Training a binary decision tree**

**In this section, we will make use of the code that we have written above. We will start by training a basic decision tree. Please use the training data provided to train a decision tree. (In Matlab, assuming you have completed the `computeOptimalSplit.m` function, the function `trainDT.m` can do this training for you.)**

**Please specify the total number of nodes and the total number of leaf nodes in the tree. (In Matlab, the function `gatherTreeStats.m` will be useful.) Also, please report the classification accuracy (percent correct) of the learned decision tree on the provided training and testing data. (In Matlab, the function `batchClassifyWithDT.m` will be useful.)**

**B2. Pruning a binary decision tree**

[15] In practice, you can often simply continue the pruning process until the validation error fails to increase by a predefined amount. However, for illustration purposes, we will continue until there is only one node left in the tree.

**Now we will make use of the pruning code we have written. Please start with the tree that was just trained in the previous part of the problem and make use of the validation data to iteratively remove nodes in the greedy manner described in the section above. Please continue iterations until a degenerate tree with only a single root node remains. For each tree that is produced, please calculate the classification accuracy for that tree on the training, validation and testing datasets.**

**After collecting this data, please plot a line graph relating classification accuracy on the test set to the number of leaf nodes in each tree (so number of leaf nodes should be on the X-axis and classification accuracy should be on the Y-Axis). Please add to this same figure, similar plots for percent accuracy on training and validation data. The number of leaf nodes should range from 1 (for the degenerate tree) to the the number present in the unpruned tree. The Y-axis should be scaled between 0 and 1.**

**Please comment on what you notice and how this illustrates overfitting. Include the produced figure and any code you needed to write to produce the figure and calculate intermediate results with your homework submission.**

**B3. Drawing a binary decision tree**

**One of the benefits of decision trees is the classification scheme they encode is easily understood by humans. Please select the binary decision tree from the pruning analysis above that produced the highest accuracy on the validation dataset and diagram it. (In the event that two trees have the same accuracy on validation data, select the tree with the smaller number of leaf nodes.) When stating the feature attributes that are used in splits, please use the attribute names (instead of index) listed in the dataset section of this problem. (If using the provided Matlab code, the function `trainDT` has a section of comments which describes how you can interpret the structure used to represent a decision tree in the code.)**

***Hint*: The best decision tree as measured on validation data for this problem should not be too complicated, so if drawing this tree seems like a lot of work, then something may be wrong.**

**B4. An alternative splitting method**

**While information gain is one criterion to use when estimating the optimal split, it is by no means the only one. Consider instead using a criterion where we try to minimize the *weighted misclassification rate*.**

**Formally, assume a set of $D$ data samples $\{< \vec{x}^{(i)}, y^{(i)} >\}_{i=1}^{D}$, where $y^{(i)}$ is the label of sample $i$, and $\vec{x}^{(i)}$ is the feature vector for sample $i$. Let $x(j)^{(i)}$ refer to the value of the $j^{th}$ attribute of the feature vector for data point $i$.**

**Now, to pick a split criterion, we pick a feature attribute, $a$, and a threshold value, $t$, to use in the split. Let:**

$$p_{below}(a,t) = \frac{1}{D}\sum_{i=1}^{D} \mathbb{I}\left(x(a)^{(i)} \leq t\right)$$

$$p_{above}(a,t) = \frac{1}{D}\sum_{i=1}^{D} \mathbb{I}\left(x(a)^{(i)} > t\right)$$

**and let:**

$$l_{below}(a,t) = Mode\left(\{y_i\}_{i:x(a)^{(i)} \leq t}\right)$$
$$l_{above}(a,t) = Mode\left(\{y_i\}_{i:x(a)^{(i)} > t}\right)$$

**The split that minimizes the weighted misclassification rate is then the one which minimizes:**

$$O(a,t) = p_{below}(a,t) \sum_{i:x(a)^{(i)} \leq t} \mathbb{I}\left(y^{(i)} \neq l_{below}(a,t)\right) + p_{above}(a,t) \sum_{i:x(a)^{(i)} > t} \mathbb{I}\left(y^{(i)} \neq l_{above}(a,t)\right)$$

**Please modify the code for your `computeOptimalSplit.m` (or equivalent function if not using Matlab) to perform splits according to this criterion. Attach the code of your modified function when submitting your homework.**

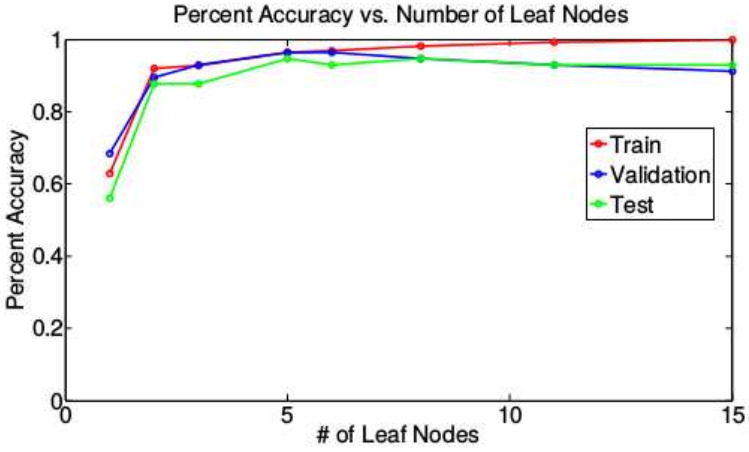
**After modifying `computeOptimalSplit.m`, please retrain a decision tree (without doing any pruning). In your homework submission, please indicate the total number of nodes and total number of leaf nodes in this tree. How does this compare with the tree that was trained using the information gain criterion?**

**Erratum: It is important to note there is an *error* in question B.4, the alternative splitting method. The question stated that if you minimized the equation for $O(a,t)$ with respect to $a$ and $t$, you would find the optimal split for the misclassification rate criteria. However, this function was missing something important. The terms summing the number of samples misclassified above and below the split point should have been *normalized*. Specifically, the term summing the number of samples misclassified above the split should have been divided by the total number of samples above the split and the term summing the number of samples misclassified below the split should have been divided by the total number of samples below the split.**

**Solution:**

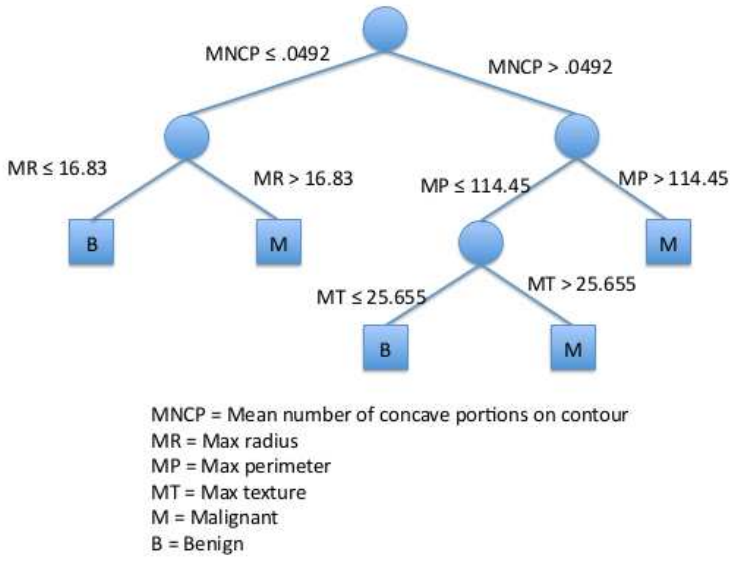
**B1. There are 29 total nodes and 15 leafs in the unpruned tree. The training accuracy is 100% and test accuracy is 92.98%.**

**B2. The correct plot is shown below.**




**Overfitting is evident: as the number of leafs in the decision tree grows, performance on the training set of data increases. However, after a certain point, adding more leaf nodes (after 5 in this case) detrimentally affects performance on test data as the more complicated decision boundaries that are formed essentially reflect noise in the training data.**

**B3. The correct diagram is shown below.**




**B4. The new tree has 16 leafs and 31 nodes. The new tree has 1 more leaf and 2 more nodes than the original tree.**

**12.** **(AdaBoost: application on Bupa Liver Disorder dataset)**

- ***CMU, 2007 spring, Carlos Guestrin, HW2, pr. 2.3***

**Implement the AdaBoost algorithm using a decision stump as the weak classifier.**

**AdaBoost trains a sequence of classifiers. Each classifier is trained on the same set of training data $(\mathbf{x}_i, y_i)$, $i = 1, \ldots, m$, but with the significance $D_t(i)$ of each example $\{\mathbf{x}_i, y_i\}$ weighted differently. At each iteration, a classifier, $h_t(\mathbf{x}) \rightarrow \{-1, 1\}$, is trained to minimize the weighted classification error, $\sum_{i=1}^{m} D_t(i) \cdot I(h_t(\mathbf{x}_i) \neq y_i)$, where $I$ is the indicator function ($0$ if the predicted and actual labels match, and $1$ otherwise). The overall prediction of the AdaBoost algorithm is a linear combination of these classifiers, $H_T(\mathbf{x}) = sign(\sum_{t=1}^{T} \alpha_t h_t(\mathbf{x}))$.**

**A decision stump is a decision tree with a single node. It corresponds to a single threshold in one of the features, and predicts the class for examples falling above and below the threshold respectively, $h_t(\mathbf{x}) = C_1 I(x^j \geq c) + C_2 I(x^j < c)$, where $x^j$ is the $j^{\text{th}}$ component of the feature vector x. For this algorithm split the data based on the weighted classification accuracy described above, and find the class assignments $C_1, C_2 \in \{-1, 1\}$, threshold $c$, and feature choice $j$ that maximizes this accuracy.**

**a. Evaluate your AdaBoost implementation on the Bupa Liver Disorder dataset that is available for download from the ... website. The classification problem is to predict whether an individual has a liver disorder (indicated by the selector feature) based on the results of a number of blood tests and levels of alcohol consumption. Use $90\%$ of the dataset for training and $10\%$ for testing. Average your results over 50 random splits of the data into training sets and test sets. Limit the number of boosting iterations to 100. In a single plot show:**

- **average training error after each boosting iteration**
- **average test error after each boosting iteration**

**b. Using all of the data for training, display the selected feature component $j$, threshold $c$, and class label $C_1$ of the decision stump $h_t(\mathbf{x})$ used in each of the first 10 boosting iterations $(t = 1, 2, ..., 10)$.**

**c. Using all of the data for training, in a single plot, show the empirical cumulative distribution functions of the margins $y_i f_T(\mathbf{x}_i)$ after 10, 50 and 100 iterations respectively, where $f_T(\mathbf{x}) = \sum_{t=1}^{T} \alpha_t h_t(\mathbf{x})$. Notice that in this problem, before calculating $f_T(\mathbf{x})$, you should normalize the $\alpha_t$s so that $\sum_{t=1}^{T} \alpha_t = 1$. This is to ensure that the margins are between $-1$ and $1$.**

***Hint*: The empirical cumulative distribution function of a random variable $X$ at $x$ is the proportion of times $X \leq x$.**

**13.** **(AdaBoost (basic, and randomized decision stumps versions): application to high energy physics)**

• ○ ***Stanford, 2016 fall, A. Ng, J. Duchi, HW2, pr. 6.d***

**In this problem, we apply [two versions of the] AdaBoost algorithm to detect particle emissions in a high-energy particle accelerator. In high energy physics, such as at the Large Hadron Collider (LHC), one accelerates small particles to relativistic speeds and smashes them into one another, tracking the emitted particles. The goal in these problems is to detect the emission of certain interesting particles based on other observed particles and energies.[16] In this problem, we will applied to 18 . All data for the problem is available at ....**

**You will implement AdaBoost using decision stumps and run it on data developed from a physics-based simulation of a high-energy particle accelerator.**

**We provide two datasets, `boosting-train.csv` and `boosting-test.csv`, which consist of training data and test data for a binary classification problem. The files are comma-separated files, the first column of which consists of binary $\pm 1$-labels $y^{(i)}$, the remaining 18 columns are the raw attribtues (low- and high-level physics-based features).**

**The MatLab file `load_data.m`, which we provide, loads the datasets into memory, storing training data and labels in appropriate vectors and matrices, and then performs boosting using your implemented code, and plots the results.**

**a. Implement a method that finds the optimal thresholded decision stump for a training set $\{x^{(i)}, y^{(i)}\}_{i=1}^m$ and distribution $p \in \mathbb{R}_+^m$ on the training set. In particular, fill out the code in the method `find_best_threshold.m`.**

**b. Implement boosted decision stumps by filling out the code in the method `stump_booster.m`. Your code should implement the weight updating at each iteration $t = 1, 2, \ldots$ to find the optimal value $\theta_t$ given the feature index and threshold.**

**c. Implement *random boosting*, where at each step the choice of decision stump is made completely randomly. In particular, at iteration $t$ random boosting chooses a random index $j \in \{1, 2, \ldots, n\}$, then chooses a random threshold $s$ from among the data values $\{x_j^{(i)}\}_{i=1}^m$, and then chooses the $t$-th weight $\theta_t$ optimally for this (random) classifier $\phi_{s,+}(x) = \text{sign}(x_j - s)$. Implement this by filling out the code in `random_booster.m`.**

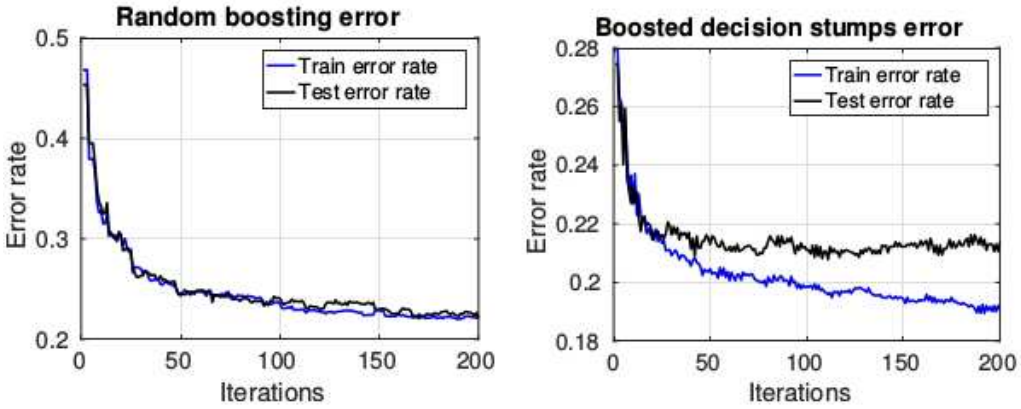
**d. Run the method load data.m with your implemented boosting methods. Include the plots this method displays, which show the training and test error for boosting at each iteration $t = 1, 2, \ldots$. Which method is better?**

**[A few notes: we do not expect boosting to get classification accuracy better than approximately 80% for this problem.]**

**Solution:**

[16] For more information, see the following paper: Baldi, Sadowski, Whiteson. Searching for Exotic Particles in High-Energy Physics with Deep Learning. Nature Communications 5, Article 4308. http://arxiv.org/abs/1402.4735.

**Random decision stumps require about 200 iterations to get to error .22 or so, while regular boosting (with greedy decision stumps) requires about 15 iterations to get this error. See figure below.**




**[Caption:] Boosting error for random selection of decision stumps and the greedy selection made by boosting.**

**14.** **(AdaBoost with logistic loss, applied on a breast cancer dataset)**

● ○ ***MIT, 2003 fall, Tommi Jaakkola, HW4, pr. 2.4-5***

**a. We have provided you with most of [MatLab code for] the boosting algorithm with the logistic loss and decision stumps. The available components are `build_stump.m`, `eval_boost.m`, `eval_stump.m`, and the skeleton of `boost_logistic.m`. The sceleton includes a bi-section search of the optimizing $\alpha$ but is missing the piece of code that updates the weights. Please fill in the appropriate weight update.**

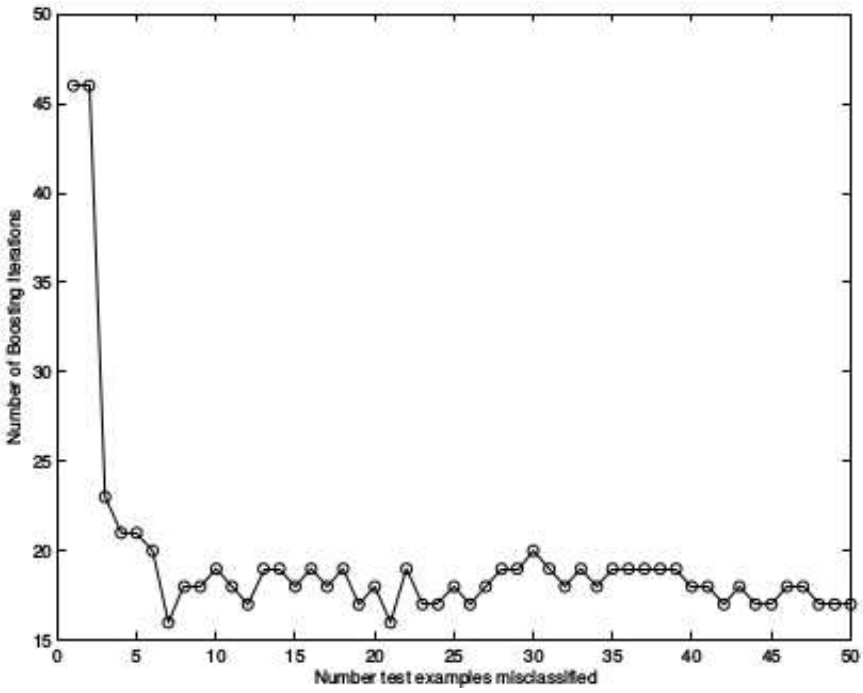
**`model = boost logistic(X,y,10);` returns a cell array of 10 stumps. The routine `eval_boost(model,X)` evaluates the combined discriminant function corresponding to any such array.**

**b. We have provided a dataset pertaining to cancer classification (see `cancer.txt` for details). You can get the data by `data = loaddata`; which gives you training examples `data.xtrain` and labels `data.ytrain`. The test examples are in `data.xtest` and `data.ytest`. Run the boosting algorithm with the logistic loss for 50 iterations and plot the training and test errors as a function of the number of iterations. Interpret the resulting plot.**

**Note that since the boosting algorithm returns a cell array of component stumps, stored for example in `model`, you can easily evaluate the predictions based on any smaller number of iterations by selecting a part of this array as in `model{1:10}`.**

<u>Solution:</u>

**Plot of number of misclassified test cases (out of 483 cases) vs. number of boosting iterations.**



**15.** **(AdaBoost with confidence rated classifiers application to handwritten digit recognition analysis of the evolution of voting margins)**

• ○ ***MIT, 2001 fall, Tommi Jaakkola, HW3, pr. 2.4***

**Let's explore a how AdaBoost behaves in practice. We have provided you with MatLab code that finds and evaluates (confidence rated) decision stumps.[17] These are the hypothesis that our boosting algorithm assumes we can generate. The relevant Matlab files are `boost_digit.m`, `boost.m`, `eval_boost.m`, `find_stump.m`, `eval_stump.m`. You'll only have to make minor modifications to `boost.m` and, a bit later, to `eval_boost.m` and `boost digit.m` to make these work.**

**a. Complete the weight update in `boost.m` and run `boost_digit` to plot the training and test errors for the combined classifier as well as the corresponding training error of the decision stump, as a function of the number of iterations. Are the errors what you would expect them to be? Why or why not?**

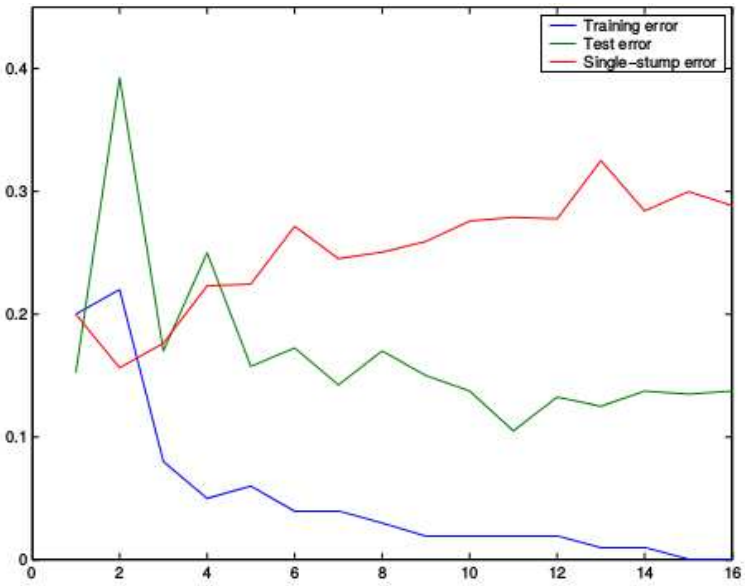
**We will now investigate the classification margins of training examples. Recall that the classification margin of a training point in the boosting context reflects the "confidence" in which the point was classified correctly. You can view the margin of a training example as the difference between the weighted fraction of votes assigned to the correct label and those assigned to the incorrect one. Note that this is not a geometric notion of "margin" but one based on votes. The margin will be positive for correctly classified training points and negative for others.**

**b. Modify `eval boost.m` so that it returns normalized predictions (normalized by the total number of votes). The resulting predictions should be in the range $[-1, 1]$. Fill in the missing computation of the training set margins in `boost digit.m` (that is, the classification margins for each of the training points). You should also uncomment the plotting script for cumulative margin distributions (what is plotted is, for each $-1 < r < 1$ on the horizontal axis, what fraction of the training points have a margin of at least $r$). Explain the differences between the cumulative distributions after 4 and 16 boosting iterations.**

**Solution:**

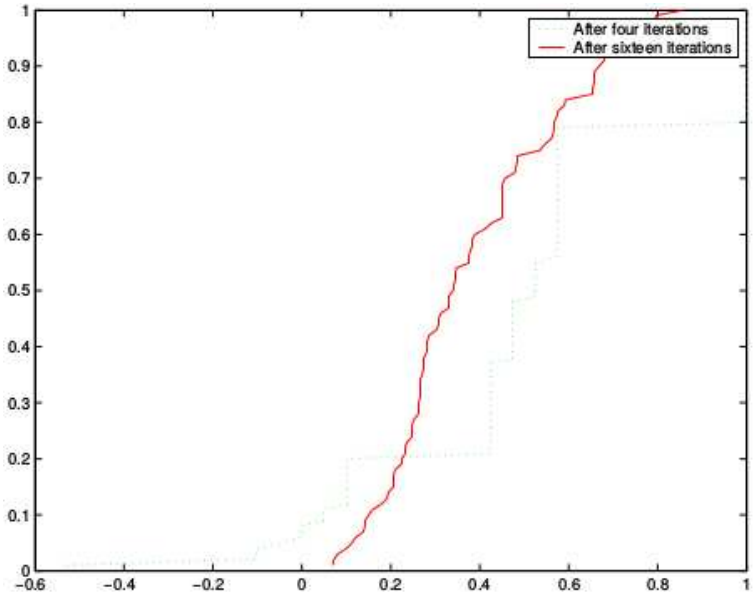
[17] LC: For some teoretical properties of confidence rated [weak] classifiers [when used] in connection with AdaBoost, see MIT, 2001 fall, Tommi Jaakkola, HW3, pr. 2.1-3.

**a.**




**As can be seen in the figure, the training and test errors decrease as we perform more boosting iterations. Eventually the training error reaches zero, but we do not overfit, and the test error remains low (though higher than the training error). However, no single stump can predict the training set well, and especially since we continue to emphasize "difficult" parts of the training set, the error of each particular stump remains high, and does not drop bellow about** $1/3$.

**b.**




**The key difference between the cumulative distributions after 4 and 16 boosting iterations is that the additional iterations seem to push the left (low end) tail of the cumulative distribution to the right. To understand the effect,**

**note that the examples that are difficult to classify have poor or negative classification margins and therefore define the low end tail of the cumulative distribution. Additional boosting iterations concentrate on the difficult examples and ensure that their margins will improve. As the margins improve, the left tail of the cumulative distribution moves to the right, as we see in the figure.**

**16.** **(AdaBoost with logistic loss: studying the evolution of voting margins as a function of boosting iterations)**

• ○ ***MIT, 2009 fall, Tommi Jaakkola, HW3, pr. 2.4***

**We have provided you with MatLab code that you can run to test how AdaBoost works.**
`mod = boost(X,y,ncomp)` **generates an ensemble (a cell array of** `ncomp` **base learners) based on training examples $X$ and labels $y$.**
`load data.mat` **gives you $X$ and $y$ for a simple classification task. You can then generate the ensemble with any number of components (e.g., 50). The cell array** `mod` **simply lists the base learners in the order in which they were found. You can therefore plot the ensemble corresponding to the first** `i` **base learners by** `plot_decision(mod(1:i),X,y)`**, or individual base learners via** `plot_decision({mod{i}},X,y)`**.**

`plot_voting_margin(mod,X,y,th)` **helps you study how the voting margins change as a function of boosting iterations. For example, the plot with** `th = 0` **gives the fraction of correctly classified training points (voting margin $> 0$) as a function of boosting iterations. You can also plot the curves for multiple thresholds at once as in** `plot_voting_margin(mod,X,y,[0,0.05,0.1,0.5])`**. Explain why some of these tend to increase while others decrease as a function of boosting iterations. Why does the curve corresponding to** `th = 0.05` **continue to increase even after all the points are correctly classified?**

<u>Solution:</u>

**Let $h_m(x) = \sum_{i=1}^m \alpha_i h_i(x)$ denote the ensemble classifier after m boosting iterations, and let $\hat{h}_m(x) = \frac{\sum_{i=1}^m \alpha_i h_i(x)}{\sum_{i=1}^m \alpha_i}$ be its normalized version. Let $f(\tau, m)$ denote the fraction of training examples $(x_t, y_t)$ with voting margin $y_t \hat{h}_m(x_t) = \frac{y_t h_m(x_t)}{\sum_{i=1}^m \alpha_i} > \tau$. From our plot, we notice that $f(\tau, m)$ is increasing with $m$ (quite roughly and not at all monotonically) for small values of $\tau$, like $\tau = 0, 0.05, 0.1$, but decreasing for large values of $\tau$, like $\tau = 0.5$. (The threshold at which the transition occurs seems to be somewhere in the interval $0.115 > \tau > 0.105$.)**

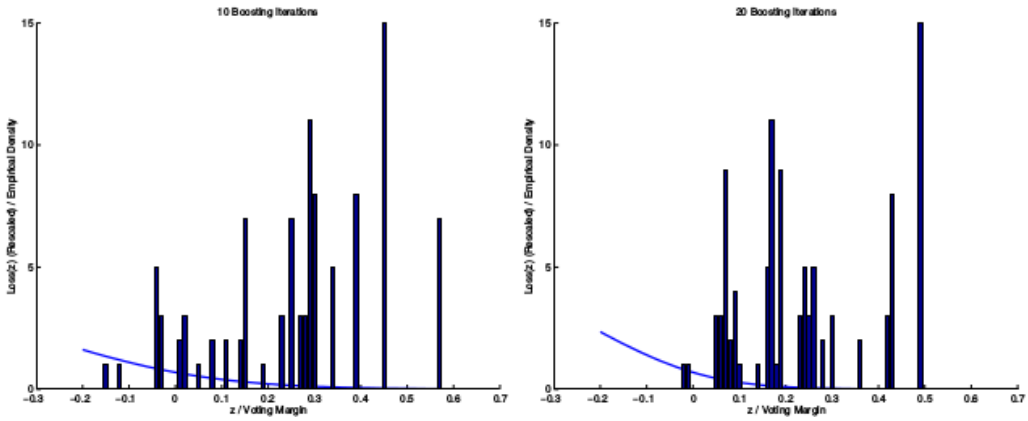
**To explain this, consider the boosting loss function, $J_m = \sum_{t=1}^n L(y_t\, h_m(x_t))$, which is decreasing in the voting margins $y_t \hat{h}_m(x_t)$. To minimize $J_m$, we will try to make all the voting margins $y \hat{h}_m(x_t)$ as positive as possible. As $m$ increases, $\sum_{i=1}^m \alpha_i$ only grows, so a negative voting margin $y_t \hat{h}_m(x_t) < 0$ only becomes more costly. So, after a sufficient number of iterations, we know that boosting will be able to classify all points correctly, and all points will have positive voting margin. So, $f(0, m)$ roughly increases from 0.5 to 1, and stays at 1 once $m$ is sufficiently large.**

**As $m$ increases even more, we should expect that the minimum voting margin $\min_t y_t \hat{h}(x_t)$ continues to increase. This is because there is little incentive to make the larger $y_t \hat{h}_m(x_t)$ any more positive; it is more effective to make the smaller $y_t \hat{h}_m(x_t)$ more positive. Using an argument similar to the one from part $a$ of this problem (MIT, 2009 fall, T. Jaakkola, HW3, pr. 2; LC: or, better CMU, 2016 spring, W. Cohen, N. Balcan, HW4, pr. 3.3.3), we can show that**

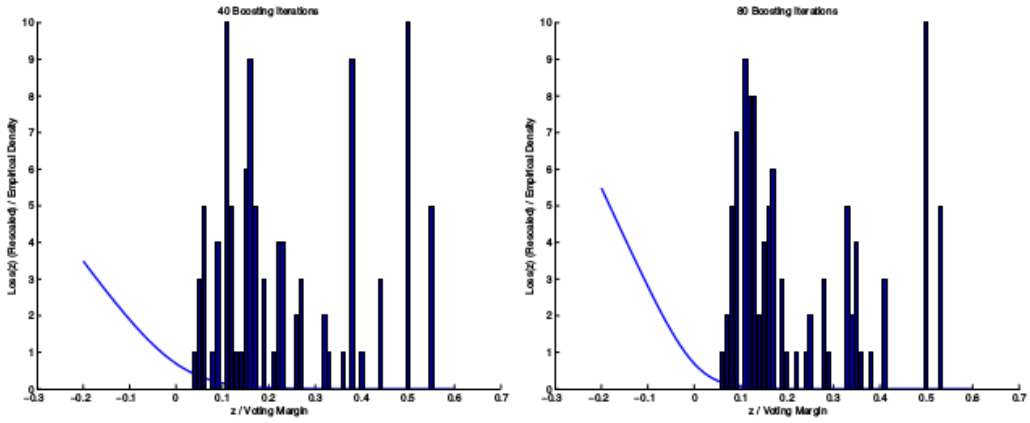
**the examples which are barely correct have larger weight than the examples which are clearly correct, since $dL(z)$ is larger near $0$.**

**However, our decision stumps are fairly weak classifiers. If we want to perform better on some subset of points (namely, the ones with smaller margin), we must compromise on the rest (namely, the ones with larger margin). Thus, what we get is that the minimum voting margin (which costs more) will become larger at the expense of the maximum voting margin (which costs less). Similarly, $f(\tau, n)$ for a small threshold $\tau$ will increase at the expense of $f(\tau, n)$ for a large $\tau$.**

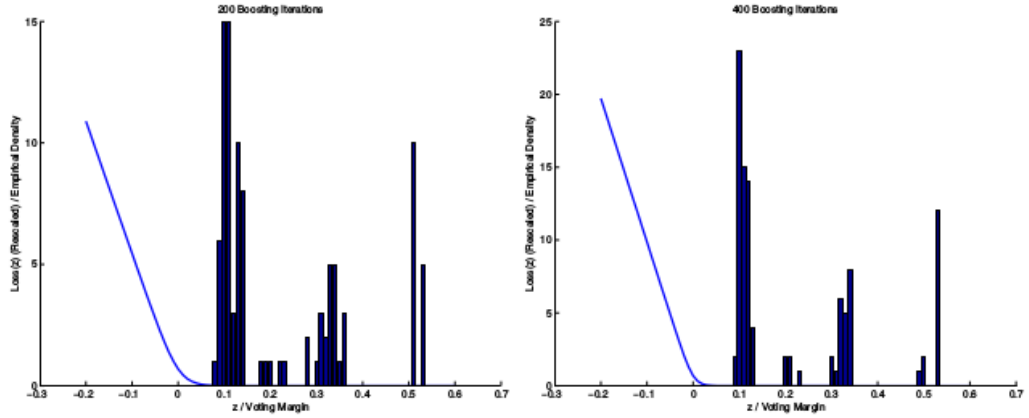
**A visual way to see this is to consider a graph of the rescaled loss function $L((\sum_{i=1}^{m} \alpha_i)\tau)$ vs. the voting margin $\tau = y_t \hat{h}_m(x_t)$. As the number of boosting iterations increases, the graph is compressed along the horizontal axis (although increasingly slowly). So to make $J_m$ smaller, we must basically shift the entire distribution of voting margins to the right as much as possible (though we can only do so increasingly slowly). In doing this, we are forced to compromise some of the points farthest to the right, moving them inward. Thus, with more iterations, the distribution of margins narrows. Here, $f(\tau, n)$ can be related to the cumulative density on the empirical distribution of voting margins. So, $f(\tau, n) = P(\text{margin} > \tau)$ for a small $\tau$ will increase, while $1 - f(\tau, n) = P(\text{margin} < \tau)$ for large $\tau$ will also increase (or at least be non-decreasing).**




**[Caption:] Voting Margins, 10-20 iterations.**




**[Caption:] Voting Margins, 40-80 iterations.**




**[Caption:] Voting Margins, 200-400 iterations.**

**[Caption:] Empirical Distributions of Voting Margins.**

**17.** **(Liniear regression and AdaBoost; cross-validation; application on the *Wine* dataset)**

• ○ ***CMU, 2009 spring, Ziv Bar-Joseph, HW3, pr. 1***

**In this problem, you are going to compare the performance of a basic linear classifier and its boosted version on the *Wine* dataset (available on our website). The dataset, given in the file `wine.mat`, contains the results of a chemical analysis of wines grown in the same region in Italy but derived from three different cultivars. The analysis determined the quantities of 13 constituents found in each of the three types of wines. *Note* that when you are doing cross-validation, you want to ensure that across all the folds the proportion examples from each class is roughly the same.**

**a. Implement a basic linear classifier using linear regression. All data points are equally weighted.**

**A *linear classifier* is defined as:**

$$f(x;\beta) = sign(\beta^\top \cdot x) = \begin{cases} -1 & \text{if } \beta^\top \cdot x < 0; \\ 1 & \text{if } \beta^\top \cdot x \geq 0. \end{cases}$$

**Your algorithm should minimize the classification error defined as:**

$$err(f) = \sum_{i=1}^{n} \frac{(y_i - f(x_i))^2}{4n}$$

***Note*: The first step for data preprocessing is to augment the data. In MatLab, this can be done as:**

```
X_new = [ones(size(X,1), 1) X];
```

***Hint*: You may want to use the MatLab function `fminsearch` to get the optimal solution for $\beta$.**

***Handin*: Please turn in a MatLab source file `linear_learn.m` which takes in two inputs data matrix `x` and label `y`, and returns a linear model. You may have additional functions/files if you want.**

**b. Do 10-fold cross validation for the linear classifier. Report the average training and test errors for all the folds.**

***Handin*: Please turn in a MatLab source file `cv.m`.**

**c. Modify your algorithm in `linear_learn.m` to accommodate weighted samples. Given the weight $w$ for sample data $X$, what is the classification error? You may want to refer to part $a$. Please implement the *weighted version* of the learning algorithm for the *linear classifier*.**

**Note originally the unweighted version could be viewed as one with equal weights $1/n$.**

***Handin*: Please turn in a MatLab source file `linear_learn.m` which takes in three inputs data matrix `x`, label `y` and weights `w`, and returns a linear model. You may have additional functions/files if you want. *Note* tat your code should have *backward compatibility* – it behaves like unweighted version if $w$ is not given.**

**d. Implement AdaBoost for the linear classifer using the re-weighting and re-training idea. Refer to the lecture slides or the ML exercise book for the AdaBoost algorithm (problem DT-62-prim in the *Decision Trees* chapter).**

***Handin*: Please turn in a MatLab source file `adaBoost.m`.**

**e. Do 10-fold cross-validation on the Wine dataset using AdaBoost with the linear classifier as the weak learner, for 1 to 100 iterations. Plot the average training and test errors for all the folds as a function of the number of boosting iterations. Also, draw horizontal lines corresponding to the training and test errors for the linear classifier that you obtain in part $b$. Discuss your results.**

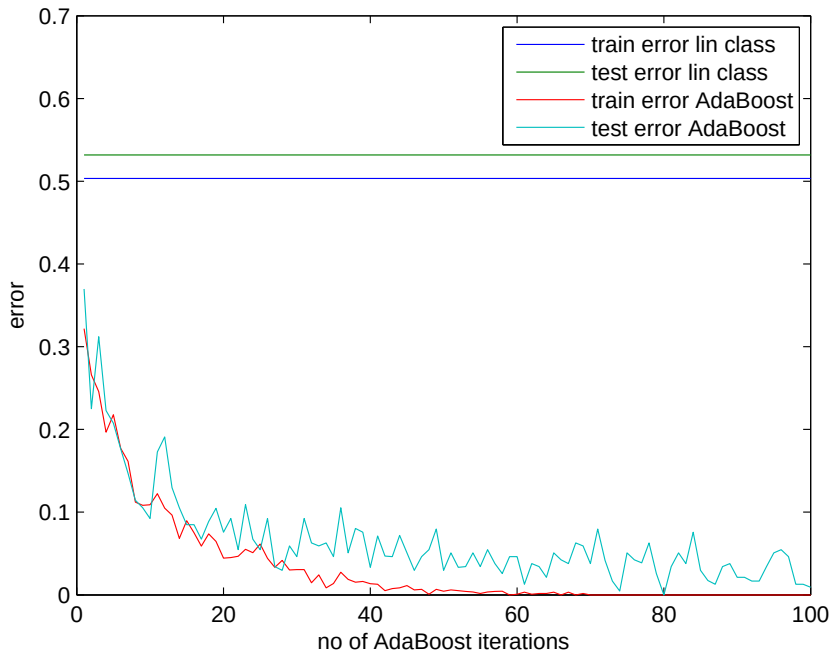
***Handin*: Please turn in a MatLab source file cv `ab.m`. You may reuse functions in part $b$.**

**Solution:**

**c.**

$$err_w(f) = \sum_{i=1}^{n} w_i \frac{(y_i - f(x_i))^2}{4}.$$

**e. Sample plot:**



# 3 Bayesian Classification

**18.** **(Naive Bayes: spam filtering)**

• ○ ***Stanford, 2012 spring, Andrew Ng, pr. 6***
***Stanford, 2015 fall, Andrew Ng, HW2, pr. 3.a-c***
***Stanford, 2009 fall, Andrew Ng, HW2, pr. 3.a-c***

**In this exercise, you will use Naive Bayes to classify email messages into spam and nonspam groups. Your dataset is a preprocessed subset of the *Ling-Spam Dataset*,[18] provided by Ion Androutsopoulos. It is based on 960 real email messages from a linguistics mailing list.**

**There are two ways to complete this exercise. The first option is to use the Matlab/Octave-formatted features we have generated for you. This requires using Matlab/Octave to read prepared data and then writing an implementation of Naive Bayes. To choose this option, download the data pack `ex6DataPrepared.zip`.**

**The second option is to generate the features yourself from the emails and then implement Naive Bayes on top of those features. You may want this option if you want more practice with features and a more open-ended exercise. To choose this option, download the data pack `ex6DataEmails.zip`.**

**Data Description:**

**The dataset you will be working with is split into two subsets: a 700-email subset for training and a 260-email subset for testing. Each of the training and testing subsets contain 50% spam messages and 50% nonspam messages. Additionally, the emails have been preprocessed in the following ways:**
**1. Stop word removal: Certain words like "and," "the," and "of," are very common in all English sentences and are not very meaningful in deciding spam/nonspam status, so these words have been removed from the emails.**
**2. Lemmatization: Words that have the same meaning but different endings have been adjusted so that they all have the same form. For example, "include," "includes," and "included," would all be represented as "include." All words in the email body have also been converted to lower case.**
**3. Removal of non-words: Numbers and punctuation have both been removed. All white spaces (tabs, newlines, spaces) have all been trimmed to a single space character.**

**As an *example*, here are some messages before and after preprocessing:**

**Nonspam message "5-1361msg1" *before* preprocessing:**
**Subject: Re: 5.1344 Native speaker intuitions**

```
The discussion on native speaker intuitions has been extremely
interesting, but I worry that my brief intervention may have
muddied the waters.  I take it that there are a number of
separable issues.  The first is the extent to which a native
speaker is likely to judge a lexical string as grammatical
or ungrammatical per se.  The second is concerned with the
```

[18] http://csmining.org/index.php/ling-spam-datasets.html, accessed on 21st Spetember 2016.

```
relationships between syntax and interpretation (although even
here the distinction may not be entirely clear cut).
```

**Nonspam message "5-1361msg1" *after* preprocessing:**

```
re native speaker intuition discussion native speaker intuition
extremely interest worry brief intervention muddy waters number
separable issue first extent native speaker likely judge lexical
string grammatical ungrammatical per se second concern relationship
between syntax interpretation although even here distinction entirely clear
cut
```

**For comparison, here is a preprocessed *spam* message:**

**Spam message "spmsgc19" after preprocessing:**

```
financial freedom follow financial freedom work ethic
extraordinary desire earn least per month work home special skills
experience required train personal support need ensure success
legitimate homebased income opportunity put back control finance
life ve try opportunity past fail live promise
```

**As you can discover from browsing these messages, preprocessing has left occasional word fragments and nonwords. In the end, though, these details do not matter so much in our implementation (you will see this for yourself).**

**Categorical Naive Bayes**

**To classify our email messages, we will use a Categorical Naive Bayes model. The parameters of our model are as follows:**

$$\phi_{k|y=1} \overset{not.}{=} p(x_j = k|y = 1) = \frac{\left(\sum_{i=1}^{m}\sum_{j=1}^{n_i} 1_{\{x_j^{(i)}=k \text{ and } y^{(i)}=1\}}\right) + 1}{\left(\sum_{i=1}^{m} 1_{\{y^{(i)}=1\}} n_i\right) |V|}$$

$$\phi_{k|y=0} \overset{not.}{=} p(x_j = k|y = 0) = \frac{\left(\sum_{i=1}^{m}\sum_{j=1}^{n_i} 1_{\{x_j^{(i)}=k \text{ and } y^{(i)}=0\}}\right) + 1}{\left(\sum_{i=1}^{m} 1_{\{y^{(i)}=0\}} n_i\right) |V|}$$

$$\phi_y \overset{not.}{=} p(y = 1) = \frac{\sum_{i=1}^{m} 1_{\{y^{(i)}=1\}}}{m},$$

**where**

$\phi_{k|y=1}$ **estimates the probability that a particular word in a spam email will be the $k$-th word in the dictionary,**

$\phi_{k|y=0}$ **estimates the probability that a particular word in a nonspam email will be the $k$-th word in the dictionary,**

$\phi_y$ **estimates the probability that any particular email will be a spam email.**

**Here are some other *notation conventions*:**

$m$ **is the number of emails in our training set,**

**the $i$-th email contains $n_i$ words,**

**the entire dictionary contains $|V|$ words.**

**You will calculate the parameters $\phi_{k|y=1}$, $\phi_{k|y=0}$ and $\phi_y$ from the training data. Then, to make a prediction on an unlabeled email, you will use the parameters to compare $p(x|y=1)p(y=1)$ and $p(x|y=0)p(y=0)$ [A. Ng: as described in the *lecture videos*]. In this exercise, instead of comparing the probabilities directly, it is better to work with their logs. That is, you will classify an email as spam if you find**

$$\log p(x|y=1) + \log p(y=1) > \log p(x|y=0) + \log p(y=0).$$

**A1. Implementing Naive Bayes using prepared features**

**If you want to complete this exercise using the formatted features we provided, follow the instructions in this section.**

**In the data pack for this exercise, you will find a text file named `train-features.txt`, that contains the features of emails to be used in training. The lines of this document have the following form:**

```
2 977 2
2 1481 1
2 1549 1
```

**The first number in a line denotes a document number, the second number indicates the ID of a dictionary word, and the third number is the number of occurrences of the word in the document. So in the snippet above, the first line says that Document 2 has two occurrences of word 977. To look up what word 977 is, use the `feature-tokens.txt` file, which lists each word in the dictionary alongside an ID number.**

**Load the features**

**Now load the training set features into Matlab/Octave in the following way:**

```
numTrainDocs = 700;
numTokens = 2500;
M = dlmread('train-features.txt', ' ');
spmatrix = sparse(M(:,1), M(:,2), M(:,3), numTrainDocs, numTokens);
train_matrix = full(spmatrix);
```

**This loads the data in our `train-features.txt` into a sparse matrix (a matrix that only stores information for non-zero entries). The sparse matrix is then converted into a full matrix, where each row of the full matrix represents one document in our training set, and each column represents a dictionary word. The individual elements represent the number of occurrences of a particular word in a document.**

**For *example*, if the element in the $i$-th row and the $j$-th column of `train_matrix` contains a "4", then the $j$-th word in the dictionary appears 4 times in the $i$-th document of our training set. Most entries in `train_matrix` will be zero, because one email includes only a small subset of the dictionary words.**

**Next, we'll load the labels for our training set.**

```
train_labels = dlmread('train-labels.txt');
```

**This puts the $y$-labels for each of the $m$ the documents into an $m \times 1$ vector. The ordering of the labels is the same as the ordering of the documents in the features matrix, i.e., the $i$-th label corresponds to the $i$-th row in `train_matrix`.**

**A note on the features**

**In a Categorical Naive Bayes model, the formal definition of a feature vector $\vec{x}$ for a document says that $x_j = k$ if the $j$-th word in this document is the $k$-th word in the dictionary. This does not exactly match our Matlab/Octave matrix layout, where the $j$-th term in a row (corresponding to a document) is the number of occurrences of the $j$-th dictionary word in that document.**

**Representing the features in the way we have allows us to have uniform rows whose lengths equal the size of the dictionary. On the other hand, in the formal Categorical Naive Bayes definition, the feature $\vec{x}$ has a length that depends on the number of words in the email. We've taken the uniform-row approach because it makes the features easier to work with in Matlab/Octave.**

**Though our representation does not contain contain any information about the position within an email that a certain word occupies, we do not lose anything relevant for our model. This is because our model assumes that each $\phi_{k|y}$ is the same for all positions of the email, so it's possible to calculate all the probablities we need without knowing about these positions.**

**Training**

**You now have all the training data loaded into your program and are ready to begin training your data. Here are the recommended steps for proceeding:**

**1. Calculate $\phi_y$.**

**2. Calculate $\phi_{k|y=1}$ for each dictionary word and store the all results in a vector.**

**3. Calculate $\phi_{k|y=0}$ fore each dictionary word store the all results in a vector.**

**Testing**

**Now that you have calculated all the parameters of the model, you can use your model to make predictions on test data. If you are putting your program into a script for Matlab/Octave, you may find it helpful to have separate scripts for training and testing. That way, after you've trained your model, you can run the testing independently as long as you don't clear the variables storing your model parameters.**

**Load the test data in `test-features.txt` in the same way you loaded the training data. You should now have a test matrix of the same format as the training matrix you worked with earlier. The columns of the matrix still correspond to the same dictionary words. The only difference is that now the number of documents are different.**

**Using the model parameters you obtained from training, classify each test document as spam or non-spam. Here are some general steps you can take:**

**1. For each document in your test set, calculate $\log p(\vec{x}|y = 1) + \log p(y = 1)$.**

**2. Similarly, calculate $\log p(\vec{x}|y = 0) + \log p(y = 0)$.**

**3. Compare the two quantities from (1) and (2) above and make a decision about whether this email is spam. In Matlab/Octave, you should store your predictions in a vector whose i-th entry indicates the spam/nonspam status of the i-th test document.**

**Once you have made your predictions, answer the questions in the Questions section.**

**Note**

**Be sure you work with log probabilities in the way described in the earlier instructions [A. Ng: and in the *lecture videos*]. The numbers in this exercise are small enough that Matlab/Octave will be susceptible to numerical underflow if you attempt to multiply the probabilities. By taking the log, you will be doing additions instead of multiplications, avoiding the underflow problem.**

**A2. Implementing Naive Bayes without prepared features**

**Here are some guidelines that will help you if you choose to generate your own features. After reading this, you may find it helpful to read the previous section, which tells you how to work with the features.**

**Data contents**

**The data pack you downloaded contains 4 folders:**
**a. The folders `nonspam-train` and `spam-train` contain the preprocessed emails you will use for training. They each have 350 emails.**
**b. The folders `nonspam-train` and `nonspam-test` constitute the test set containing 130 spam and 130 nonspam emails. These are the documents you will make predictions on. Notice that even though the separate folders tell you the correct labeling, you should make your predictions on all the test documents without this knowledge. After you make your predictions, you can use the correct labeling to check whether your classifications were correct.**

**Dictionary**

**You will need to generate a dictionary for your model. There is more than one way to do this, but an easy method is to count the occurrences of all words that appear in the emails and choose your dictionary to be the most frequent words. If you want your results to match ours exactly, you should pick the dictionary to be the 2500 most frequent words.**

**To check that you have done this correctly, here are the 5 most common words you will find, along with their counts.**

1. `email 2172`
2. `address 1650`
3. `order 1649`
4. `language 1543`
5. `report 1384`

**Remember to take the counts over *all* of the emails: spam, nonspam, training set, testing set.**

**Feature generation**

**Once you have the dictionary, you will need to represent your documents as feature vectors over the space of the dictionary words. Again, there are several ways to do this, but here are the *steps* you should take if you want to match the prepared features we described in the previous section.**
**1. For each document, keep track of the dictionary words that appear, along with the count of the number of occurrences.**
**2. Produce a feature file where each line of the file is a triplet of (`docID`, `wordID`, `count`). In the triplet, `docID` is an integer referring to the email, `wordID`**

is an integer referring to a word in the dictionary, and `count` is the number of occurrences of that word. For *example*, here are the first five entries of a training feature file we produced (the lines are sorted by `docID`, then by `wordID`):

```
1 19 2
1 45 1
1 50 1
1 75 1
1 85 1
```

In this snippet, Document 1 refers to the first document in the `nonspam-train` folder, `3-380msg4.txt`. Our dictionary is ordered by the popularity of the words across all documents, so a `wordID` of 19 refers to the 19th most common word.

This format makes it easy for Matlab/Octave to load your features as an array. Notice that this way of representing the emails does not contain any information about the position within an email that a certain word occupies. This is not a problem in our model, since we're assuming each $\phi_{k|y}$ is the same for all positions.

### Training and testing

Finally, you will need to train your model on the training set and predict the spam/nonspam classification on the test set. For some ideas on how to do this, refer to the instructions in the previous section about working with already-generated features.

When you are finished, answer the questions in the following Questions section.

## B. Questions

### Classification error

Load the correct labeling for the test documents into your program. If you used the pre-generated features, you can just read `test-labels.txt` into your program. If you generated your own features, you will need to write your own labeling based on which documents were in the spam folder and which were in the nonspam folder.

Compare your Naive-Bayes predictions on the test set to the correct labeling. How many documents did you misclassify? What percentage of the test set was this?

### Smaller training sets

Let's see how the classification error changes when you train on smaller training sets, but test on the same test set as before. So far you have been working with a 960-document training set. You will now modify your program to train on 50, 100, and 400 documents (the spam to nonspam ratio will still be one-to-one).

If you are using our prepared features for Matlab/Octave, you will see text documents in the data pack named `train-features-#.txt` and `train-labels-#.txt`, where the "#" tells you how many documents make up these training sets. For each of the training set sizes, load the corresponding training data into

**your program and train your model. Then record the test error after testing on the same test set as before.**

**If you are generating your own features from the emails, you will need to select email subsets of 50, 100, and 400, keeping each subset 50% spam and 50% nonspam. For each of these subsets, generate the training features as you did before and train your model. Then, test your model on the 260-document test set and record your classification error.**

**Solution:**

**An `m`-file implementation of Naive Bayes training for Matlab/Octave can be found *here* [...], and another `m`-file for testing is *here* [...]. In order for `test.m` to work, you must first run `train.m` without clearing the variables in the workspace after training.**

**Classification error**

**After training on the full training set (700 documents), you should find that your algorithm misclassifies 5 documents. This amounts to 1.9% of your test set.**

**If your test error was different, you will need to debug your program. Make sure that you are working with log probabilities, and that you are taking logs on the correct expressions. Also, check that you understand the dimensions of your features matrix and what each dimension means.**

**Smaller training sets**

**Here are the errors on the smaller training sets. Your answers may differ slightly if you generated your own features and did not use the same document subsets we used.**
**1. 50 training documents: 7 misclassified, 2.7%.**
**2. 100 training documents: 6 misclassified, 2.3%.**
**3. 400 training documents: 6 misclassified, 2.3%.**

**19.** **(Naive Bayes: weather prediction feature selection based on CVLOO)**

• ○ *CMU, 2010 fall, Ziv Bar-Joseph, HW1, pr. 4*
*CMU, 2009 fall, Ziv Bar-Joseph, HW1, pr. 3*

**You need to decide whether to carry an umbrella to school in Pittsburgh as the local weather channel has been giving inconsistent predictions recently. You are given several input features (observations). These observations are discrete, and you are expected to use a Naive Bayes classification scheme to decide whether or not you will take your umbrella to school. The domain of each of the features is as follows:**

```
season = (w, sp, su, f)
yesterday = (dry, rainy)
daybeforeyesterday = (dry, rainy)
cloud = (sunny, cloudy)
```

**and the possible classes of the output being:** `umbrella = (y, n)`.
**See** `data1.txt` **(posted on website with problem set) for data based on the above scenario with space separated fields conforming to:**

```
season yesterday daybeforeyesterday cloud umbrella
```

**a. Write code in MATLAB to estimate the conditional probabilities of each the features given the outcome. Generate a space separated file with the estimated parameters from the entire dataset by writing out all the conditional probabilities.**

**b. Write code in MATLAB to perform inference by predicting the maximum likelihood class based on training data using a *leave one out cross validation* scheme. Generate a [space separated] file with the maximum likelihood classes in order.**

**c. Are the features** `yesterday` **and** `daybeforeyesterday` **independent of each other?**

**d. Does the Naive Bayes assumption hold on this pair of input features? Why or why not?**

**e. Find a subset of 3 features from this set of 4 features where your algorithm improves its predictive ability based on a leave one out cross validation scheme. Report your improvement.**

Solution:

**a. Without pseudocounts, the conditional probabilities are:**

```
cloud cloudy y 0.8
cloud sunny y 0.2
cloud sunny n 0.6
cloud cloudy n 0.4
daybeforeyesterday dry y
daybeforeyesterday rainy
daybeforeyesterday dry n
daybeforeyesterday rainy
season w n 0.3
```

```
season sp n 0
season su n 0.3
season f n 0.4
season sp y 0.2
season su y 0.1
season f y 0.4
season w y 0.3
yesterday dry y 0.4
yesterday rainy y 0.6
yesterday rainy n 0.4
yesterday dry n 0.6
```

**b. After adding *pseudocounts*, the ML classes for the data are:**

`yyynynnnnnnynnynyyynnynynnnnnyyynnnyynnn`

**Results may vary slightly because of the way pseudocounts are implemented.**

**c. Let `yesterday` be denoted by Y and `daybeforeyesterday` by D, and `umbrella` by U.**

| Y | D | Empirical joint prob. | Empirical prob. Y | Empirical prob. D | Product of the two |
|---|---|---|---|---|---|
| **dry** | **dry** | **0.3** | **0.5** | **0.5** | **0.25** |
| **dry** | **rain** | **0.2** | **0.5** | **0.5** | **0.25** |
| **rain** | **dry** | **0.2** | **0.5** | **0.5** | **0.25** |
| **rain** | **rain** | **0.3** | **0.5** | **0.5** | **0.25** |

**Thus, they do not look independent from the data, but a stricter way to test is to subject it to a Chi-Square Test of independence. Since the number of data samples are low, we do not obtain a statistically significant result either way.**

**d. To check if the naive bayes assumption holds, we need to check for the conditional independence of the two given the umbrella label, so we partition the data based on the umbrella label and check for independence.**

| U | Y | D | Empirical joint prob. | Empirical prob. Y | Empirical prob. D | Product of the two |
|---|---|---|---|---|---|---|
| **y** | **dry** | **dry** | **0.2** | **0.4** | **0.4** | **0.16** |
| **y** | **dry** | **rain** | **0.2** | **0.4** | **0.6** | **0.24** |
| **y** | **rain** | **dry** | **0.2** | **0.6** | **0.4** | **0.24** |
| **y** | **rain** | **rain** | **0.4** | **0.6** | **0.6** | **0.36** |
| **n** | **dry** | **dry** | **0.4** | **0.6** | **0.6** | **0.36** |
| **n** | **dry** | **rain** | **0.2** | **0.6** | **0.4** | **0.24** |
| **n** | **rain** | **dry** | **0.2** | **0.4** | **0.6** | **0.24** |
| **n** | **rain** | **rain** | **0.2** | **0.4** | **0.4** | **0.16** |

**Again, they do not look independent, but the empirical joint probabilities and the product of the individual empirical probabilities look slightly closer. Again, a stricter way to test is to subject it to a Chi-Square Test of independence. Again, since the number of data samples are low, we do not obtain a statistically significant result either way.**

**e. Leaving out `yesterday`, using a LOOCV scheme, the percentage of correctly predicted instances jumps from 55% to 75%.**

**20.** **(Naive Bayes: application to document [$n$-ary] classification)**

• ○ ***CMU, 2011 spring, Tom Mitchell, HW2, pr. 3***

**In this exercise, you will implement the Naive Bayes document classifier and apply it to the classic 20 newsgroups dataset.[19] In this dataset, each document is a posting that was made to one of 20 different usenet newsgroups. Our *goal* is to write a program which can predict which newsgroup a given document was posted to.[20]**

**Model**

**Let's say we have a document $D$ containing $n$ words; call the words $\{X_1, \ldots, X_n\}$. The value of random variable $X$ i is the word found in position $i$ in the document. We wish to predict the label $Y$ of the document, which can be one of $m$ categories. We could use the model:**

$$P(Y|X_1, \ldots, X_n) \propto P(X_1, \ldots, X_n|Y) \cdot P(Y) = P(Y) \prod_i P(X_i|Y)$$

**That is, each $X$ i is sampled from some distribution that depends on its position $X_i$ and the document category $Y$. As usual with discrete data, we assume that $P(X_i|Y)$ is a multinomial distribution over some vocabulary $V$; that is, each $X_i$ can take one of $|V|$ possible values corresponding to the words in the vocabulary. Therefore, in this model, we are assuming (roughly) that for any pair of document positions $i$ and $j$, $P(X_i|Y)$ may be completely different from $P(X_j|Y)$.**

**a. Explain in a sentence or two why it would be difficult to accurately estimate the parameters of this model on a reasonable set of documents (e.g. 1000 documents, each 1000 words long, where each word comes from a 50,000 word vocabulary).**

**To improve the model, we will make the additional assumption that:**

$$\forall i, j \; P(X_i|Y) = p(X_j|Y)$$

**Thus, in addition to estimating $P(Y)$, you must estimate the parameters for the single distribution $P(X|Y)$, which we define to be equal to $P(X_i|Y)$ for all $X_i$. Each word in a document is assumed to be an i.i.d. drawn from this distribution.**

**Data**

**The data file (available on the website) contains six files:**
**1. `vocabulary.txt` is a list of the words that may appear in documents. The line number is word's id in other files. That is, the first word (`archive`) has `wordId` 1, the second word (`name`) has `wordId` 2, etc.**
**2. `newsgrouplabels.txt` is a list of newsgroups from which a document may have come. Again, the line number corresponds to the label's id, which is used in the `.label` files. The first line (`alt.atheism`) has id 1, etc.**
**3. `train.label`: Each line corresponds to the label for one document from the**

[19] http://qwone.com/~jason/20Newsgroups/, accessed on 22nd September 2016.
[20] For this question, you may write your code and solution in teams of at most 2 students.

**training set. Again, the document's id (`docId`) is the line number.**
**4. `test.label`: The same as `train.label`, except that the labels are for the test documents.**
**5. `train.data` specifies the counts for each of the words used in each of the documents. Each line is of the form `docId wordId count`, where `count` specifies the number of times the word with id `wordId` appears in the training document with id `docId`. All word/document pairs that do not appear in the file have count 0.**
**6. `test.data`: Same as train.data, except that it specified counts for test documents. If you are using Matlab, the functions `textread` and `sparse` will be useful in reading these files.**

**Implementation**

**Your first task is to implement the Naive Bayes classifier specified above. You should estimate $P(Y)$ using the MLE, and estimate $P(X|Y)$ using a MAP estimate with the prior distribution $Dirichlet(1+\alpha,\ldots,1+\alpha)$, where $\alpha = 1/|V|$ and $V$ is the vocabulary.**

**b. Report the overall *testing accuracy* (the number of correctly classified documents in the test set over the total number of test documents), and print out the *confusion matrix* (the matrix $C$, where $c_{ij}$ is the number of times a document with ground truth category $j$ was classified as category $i$).**

**c. Are there any newsgroups that the algorithm confuses more often than others? Why do you think this is?**

**In your initial implementation, you used a prior $Dirichlet(1+\alpha,\ldots,1+\alpha)$ to estimate $P(X|Y)$, and we told you set $\alpha = 1/|V|$. Hopefully you wondered where this value came from. In practice, the choice of prior is a difficult question in Bayesian learning: either we must use domain knowledge, or we must look at the performance of different values on some validation set. Here we will use the performance on the testing set to gauge the effect of $\alpha$.[21]**

**d. Re-train your Naive Bayes classifier for values of $\alpha$ between .00001 and 1 and report the accuracy over the test set for each value of $\alpha$. Create a plot with values of $\alpha$ on the $x$-axis and accuracy on the $y$-axis. Use a logarithmic scale for the $x$-axis (in Matlab, the `semilogx` command). Explain in a few sentences why accuracy drops for both small and large values of $\alpha$.**

**Identifying Important Features**

**One useful property of Naive Bayes is that its simplicity makes it easy to understand why the classifier behaves the way it does. This can be useful both while debugging your algorithm and for understanding your dataset in general. For example, it is possible to identify which words are strong indicators of the category labels we're interested in.**

**e. Propose a method for ranking the words in the dataset based on how much the classifier 'relies on' them when performing its classification (hint:**

[21] It is tempting to choose $\alpha$ to be the one with the best performance on the testing set. However, if we do this, then we can no longer assume that the classifier's performance on the test set is an unbiased estimate of the classifier's performance in general. The act of choosing $\alpha$ based on the test set is equivalent to training on the test set; like any training procedure, this choice is subject to *overfitting*.

**information theory will help). Your metric should use only the classifier's estimates of $P(Y)$ and $P(X|Y)$. It should give high scores to those words that appear frequently in one or a few of the newsgroups but not in other ones. Words that are used frequently in general English ('the', 'of', etc.) should have lower scores, as well as words that only appear appear extremely rarely throughout the whole dataset. Finally, your method this should be an overall ranking for the words, not a per-category ranking.**[22]

**f. Implement your method, set $\alpha$ back to $1/|V|$, and print out the 100 words with the highest measure.**

**g. If the points in the training dataset were not sampled independently at random from the same distribution of data we plan to classify in the future, we might call that training set biased. Dataset bias is a problem because the performance of a classifier on a biased dataset will not accurately reflect its future performance in the real world. Look again at the words your classifier is 'relying on'. Do you see any signs of dataset bias?**

**Solution:**

**a. In this model, each *position* in a given document is assumed to have its own probability distribution. Each document has only one word at each position, so if there are $M$ documents then we must estimate the parameters of roughly 50,000-dimensional distributions using only $M$ samples from that distribution. In only a thousand documents, there will not be enough samples.**

**To see it another way, the fact that a word $w$ appeared at the $i$'th position of the document gives us information about the distribution at another position $j$. Namely, in English, it is possible to rearrange the words in a document without significantly altering the document's meaning, and therefore the fact that $w$ appeared at position $i$ means that it is likely that $w$ could appear at position $j$. Thus, it would be statistically inefficient to not to make use of the information in estimating the parameters of the distribution of $X_j$.**

**b. The final accuracy of this classifier is 78.52%, with the following confusion**

[22] Some students might not like the open-endedness of this problem. I [Carl Doersch, TA at CMU] hate to say it, but nebulous problems like this are common in machine learning–this problem was actually inspired by something I worked on last summer in industry. The goal was to design a metric for finding documents similar to some query document, and part of the procedure involved classifying words in the query document into one of 100 categories, based on the word itself and the word's context. The algorithm initially didn't work as well as I thought it should have, and the only path to improving its performance was to understand what these classifiers were 'relying on' in order to do their classification–some way of understanding the classifiers' internal workings, and even I wasn't sure what I was looking for. In the end I designed a metric based on information theory and, after looking at hundreds of word lists printed from these classifiers, I eventually found a way to fix the problem. I felt this experience was valuable enough that I should pass it on to all of you.

**matrix:**

| | | Predicted Class | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | alt.atheism | comp.graphics | comp.os.ms-windows.misc | comp.sys.ibm.pc.hardware | comp.sys.mac.hardware | comp.windows.x | misc.forsale | rec.autos | rec.motorcycles | rec.sport.baseball | rec.sport.hockey | sci.crypt | sci.electronics | sci.med | sci.space | soc.religion.christian | talk.politics.guns | talk.politics.mideast | talk.politics.misc | talk.religion.misc |
| True Class | alt.atheism | 249 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 2 | 0 | 3 | 3 | 24 | 2 | 3 | 4 | 26 |
| | comp.graphics | 0 | 286 | 13 | 14 | 9 | 22 | 4 | 1 | 1 | 0 | 1 | 11 | 8 | 6 | 10 | 1 | 2 | 0 | 0 | 0 |
| | comp.os.ms-windows.misc | 1 | 33 | 204 | 57 | 19 | 21 | 4 | 2 | 3 | 0 | 0 | 12 | 5 | 10 | 8 | 3 | 1 | 0 | 5 | 3 |
| | comp.sys.ibm.pc.hardware | 0 | 11 | 30 | 277 | 20 | 1 | 10 | 2 | 1 | 0 | 1 | 4 | 32 | 1 | 2 | 0 | 0 | 0 | 0 | 0 |
| | comp.sys.mac.hardware | 0 | 17 | 13 | 30 | 269 | 0 | 12 | 2 | 2 | 0 | 0 | 3 | 21 | 8 | 4 | 0 | 1 | 0 | 1 | 0 |
| | comp.windows.x | 0 | 54 | 16 | 6 | 3 | 285 | 1 | 1 | 3 | 0 | 0 | 5 | 3 | 6 | 4 | 0 | 1 | 1 | 1 | 0 |
| | misc.forsale | 0 | 7 | 5 | 32 | 16 | 1 | 270 | 17 | 8 | 1 | 2 | 0 | 7 | 4 | 6 | 0 | 2 | 1 | 2 | 1 |
| | rec.autos | 0 | 3 | 1 | 2 | 0 | 0 | 14 | 331 | 17 | 0 | 0 | 1 | 13 | 0 | 4 | 2 | 0 | 0 | 6 | 1 |
| | rec.motorcycles | 0 | 1 | 0 | 1 | 0 | 0 | 2 | 27 | 360 | 0 | 0 | 0 | 3 | 1 | 0 | 0 | 1 | 1 | 0 | 0 |
| | rec.sport.baseball | 0 | 0 | 0 | 1 | 1 | 0 | 2 | 1 | 2 | 352 | 17 | 0 | 1 | 3 | 3 | 5 | 2 | 1 | 5 | 1 |
| | rec.sport.hockey | 2 | 0 | 1 | 0 | 0 | 0 | 2 | 1 | 2 | 4 | 383 | 0 | 0 | 0 | 0 | 1 | 2 | 0 | 1 | 0 |
| | sci.crypt | 0 | 3 | 0 | 3 | 4 | 1 | 0 | 0 | 0 | 1 | 1 | 362 | 2 | 2 | 2 | 0 | 9 | 0 | 5 | 0 |
| | sci.electronics | 3 | 20 | 4 | 25 | 7 | 4 | 8 | 11 | 6 | 0 | 0 | 21 | 264 | 9 | 7 | 1 | 3 | 0 | 0 | 0 |
| | sci.med | 5 | 7 | 0 | 3 | 0 | 0 | 3 | 5 | 4 | 1 | 0 | 1 | 8 | 320 | 8 | 7 | 6 | 5 | 8 | 2 |
| | sci.space | 0 | 8 | 0 | 1 | 0 | 3 | 1 | 0 | 1 | 0 | 1 | 4 | 6 | 5 | 343 | 3 | 2 | 1 | 12 | 1 |
| | soc.religion.christian | 11 | 2 | 0 | 0 | 0 | 2 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 2 | 0 | 362 | 0 | 1 | 2 | 15 |
| | talk.politics.guns | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 2 | 1 | 1 | 0 | 4 | 0 | 5 | 2 | 1 | 303 | 5 | 23 | 13 |
| | talk.politics.mideast | 12 | 1 | 0 | 1 | 0 | 0 | 1 | 2 | 0 | 2 | 0 | 2 | 1 | 0 | 0 | 6 | 3 | 326 | 18 | 1 |
| | talk.politics.misc | 6 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 5 | 0 | 10 | 6 | 2 | 63 | 6 | 196 | 13 |
| | talk.religion.misc | 39 | 3 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 2 | 6 | 27 | 10 | 3 | 7 | 151 |

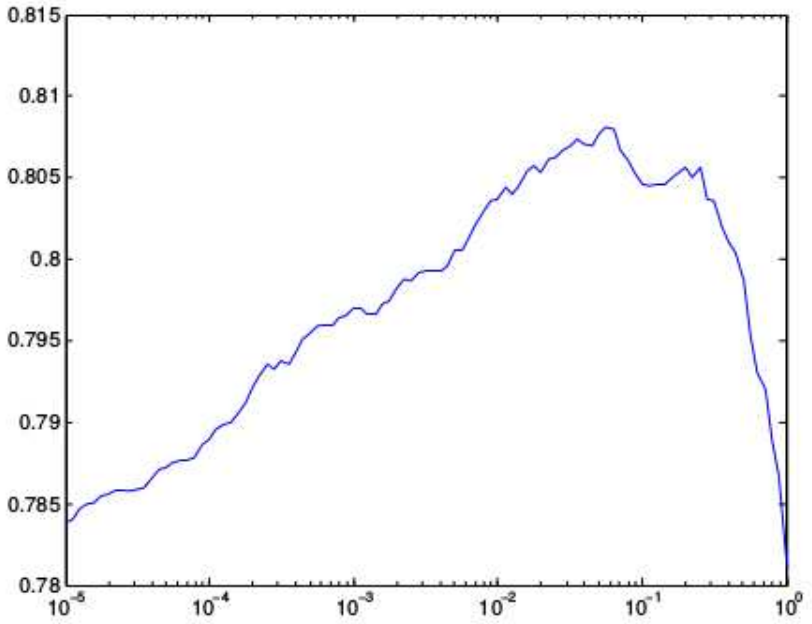
**c. From the confusion matrix, it is clear that newsgroups with a similar topics are confused frequently. Notably, those related to computers (e.g., `comp.os.ms-windows.misc` and `comp.sys.ibm.pc.hardware`), those related to politics (e.g., `talk.politics.guns` and `talk.politics.misc`), and those related to religion (`alt.atheism` and `talk.religon.misc`). Newsgroups with similar topics have similar words that identify them. For example, we would expect the computer-related groups to all use computer terms frequently.**

**d. For very small values of $\alpha$, we have that the probability of rare words not seen during training for a given class tends to zero. There are many testing documents that contain words seen only in one or two training documents, and often these training documents are of a different class than the test document. As $\alpha$ tends to zero, the probabilities of these rare words tends to dominate.[23]**

**For large values of $\alpha$, we see a classic *underfitting* behavior: the final parameter estimates tend toward the prior as $\alpha$ increases, and the prior is just something we made up. In particular, the classifier tends to underestimate the importance of rare words: for example, if $\alpha$ is 1 and we see only one occurrence of the word $w$ in the category $C$ (and we see the same number of words in each category), then the final parameter estimates are $2/21$ for category $C$ and $19/21$ that it would be something else. Furthermore, the most informative words tend to be relatively uncommon, and so we would like to**

[23] One may attribute the poor performance at small values of $\alpha$ to overfitting. While this is strictly speaking correct (the classifier estimates $P(X|Y)$ to be smaller than is realistic simply because that was the case in the data), simply attributing this to overfitting is not a sophisticated answer. Different classifiers overfit for different reasons, and understanding the differences is an important goal for you as students.

**rely on these rare words more.**




**e. There were many acceptable solutions to this question. First we will look at $H(Y|X_i = True)$, the entropy of the label given a document with a single word $w_i$. Intuitively, this value will be low if a word appears most of the time in a single class, because the distribution $P(Y|X_i = True)$ will be highly peaked. More concretely (and abbreviating $True$ as $T$),**

$$\begin{aligned}
H(Y|X_i = T) &= -\sum_k P(Y = y_k|X_i = T)\log(P(Y = y_k|X_i = T)) \\
&= -E_{P(Y=y_k|X_i=T)}\log(P(Y = y_k|X_i = T)) \\
&= -E_{P(Y=y_k|X_i=T)}\log\frac{P(X_i = T|Y = y_k)P(Y = y_k)}{P(X_i = T)} \\
&= -E_{P(Y=y_k|X_i=T)}\log\frac{P(X_i = T|Y = y_k)}{P(X_i = T)} - E_{P(Y=y_k|X_i=T)}\log(P(Y = y_k))
\end{aligned}$$

**Note that**

$$\log\frac{P(X_i = T|Y = y_k)}{P(X_i = T)}$$

**is exactly what gets added to Naive Bayes' internal estimate of the posterior probability $\log(P(Y))$ at each step of the algorithm (although in implementations we usually ignore the constant $P(X_i = T)$). Furthermore, the expectation is over the posterior distribution of the class labels given the appearance of word $w_i$. Thus, the first term of this measure can be interpreted as the expected change in the classifier's estimate of the log-probability of the 'correct' class given the appearance of word $w_i$. The second term tends to be very small relative to the first term since $P(Y)$ is close to uniform.**[24] [25]

[24] I found that the word list is the same with or without it.

[25] Another measure indicated by many students was $I(X_i, Y)$. Prof. Mitchell said that this was quite useful

**f. For the metric $H(Y|X_i = True)$:**

"nhl", "stephanopoulos", "leafs", "alomar", "wolverine", "crypto", "lemieux", "oname", "rsa", "athos", "ripem", "rbi", "firearm", "powerbook", "pitcher", "bruins", "dyer", "lindros", "lciii", "ahl", "fprintf", "candida", "azerbaijan", "baerga", "args", "iisi", "gilmour", "clh", "gfci", "pitchers", "gainey", "clemens", "dodgers", "jagr", "sabretooth", "liefeld", "hawks", "hobgoblin", "rlk", "adb", "crypt", "anonymity", "aspi", "countersteering", "xfree", "punisher", "recchi", "cipher", "oilers", "soderstrom", "azerbaijani", "obp", "goalie", "libxmu", "inning", "xmu", "sdpa", "argic", "serdar", "sumgait", "denning", "ioccc", "obfuscated", "umu", "nsmca", "dineen", "ranck", "xdm", "rayshade", "gaza", "stderr", "dpy", "cardinals", "potvin", "orbiter", "sandberg", "imake", "plaintext", "whalers", "moncton", "jaeger", "uccxkvb", "mydisplay", "wip", "hicnet", "homicides", "bontchev", "canadiens", "messier", "bure", "bikers", "cryptographic", "ssto", "motorcycling", "infante", "karabakh", "baku", "mutants", "keown", "cousineau"

**For the metric $I(X_i, Y)$:**

"windows", "god", "he", "scsi", "car", "drive", "space", "team", "dos", "bike", "file", "of", "that", "mb", "game", "key", "mac", "jesus", "window", "dod", "hockey", "the", "graphics", "card", "image", "his", "gun", "encryption", "sale", "apple", "government", "season", "we", "games", "israel", "disk", "files", "ide", "controller", "players", "shipping", "chip", "program", "was", "cars", "nasa", "win", "year", "were", "they", "turkish", "motif", "people", "armenian", "play", "drives", "bible", "use", "widget", "pc", "clipper", "offer","jpeg", "baseball", "bus", "my", "nhl", "software", "is", "db", "server", "jews","os", "israeli", "output", "data", "system", "who", "league", "armenians", "for", "christian", "christians", "entry", "mhz", "ftp", "price", "christ", "guns","thanks", "church", "color", "teams", "privacy", "condition", "launch", "him", "com", "monitor", "ram"

**Note the presence of the words "car", "of", "that", etc.**

**g. It is certain that the dataset was collected over some finite time period in the past. That means our classifier will tend to rely on some words that are specific to this time period. For the first word list, "stephanopolous" refers to a politician who may not be around in the future, and "whalers" refers to the Connecticut hockey team that was actually being desolved at the same time as this dataset was being collected. For the second list, "ghz" has almost certainly replaced "mhz" in modern computer discussions, and the controversy regarding Turkey and Armenia is far less newsworthy today. As a result, we should expect the classification accuracy on the 20-newsgroups testing set to**

in *functional Magnetic Resonance Imaging* (fMRI) data. Intuitively, this measures the amount of information we learn by observing $X_i$. A *issue* with this measure is that Naive Bayes only really learns from $X_i$ in the event that $X_i = True$, and essentially ignores this variable when $X_i = False$ (thus, the issue was introduced because we're computing our measure on $X_i$ rather than on $X$). Note that this is not the case in fMRI data (i.e., you compute the mutual information directly on the features used for classification), which explains why mutual information works better in that domain. Note that $X_i = False$ most of the time for informative words, so in the formula:

$$I(X_i, Y) = H(X_i) - H(X_i|Y) =$$

$$- \sum_{x_i \in \{T,F\}} P(X_i = x_i) \left[ \log P(X_i = x_i) - \sum_k P(Y = y_k | X = x_i) \log P(Y = y_k | X = x_i) \right]$$

we see that the term for $x_i = F$ tends to dominate even though it is essentially meaningless. Another disadvantage of this metric is that it's more difficult to implement.

**significantly overestimate the classification accuracy our algorithm would have on a testing sample from the same newsgroups taken today.**[26]

[26]Sadly, there is a lot of *bad* machine learning research that has resulted from biased datasets. Researchers will train an algorithm on some dataset and find that the performance is excellent, but then apply it in the real world and find that the performance is terrible. This is especially common in *computer vision* datasets, where there is a tendency to always photograph a given object in the same environment or in the same pose. In your own research, make sure your datasets are realistic!

# 4 Instance-Based Learning

**21.** **($k$-NN vs Gaussian Naive Bayes: application on a [given] dataset of points from $\mathbb{R}^2$)**

● ○ ***CMU, ? spring, ML course 10-701, HW1, pr. 5***

**In this problem, you are asked to implement and compare the $k$-nearest neighbor and Naive Bayes classifiers in Matlab. You are only permitted to use existing tools for simple linear algebra such as matrix multiplication. Do NOT use any toolkit that performs machine learning functions. The provided data (`traindata.txt` for training, `testdata.txt` for testing) has two real features $X_1$, $X_2$ and the variable $Y$ representing a class. Each line in the data files represents a data point $(X_1, X_2, Y)$.**

**a. How many parameters does Gaussian Naive Bayes classifier need to estimate? How many parameters for $k$-NN (for a fixed $k$)? Write down the equation for each parameter estimation.**

**b. Implement $k$-NN in MATLAB and test each point in `testdata.txt` using `traindata.txt` as the set of possible neighbors using $k = 1, \ldots, 20$. Plot the test error vs. $k$. Which value of k is optimal for your test dataset?**

**c. Implement Gaussian Naive Bayes in MATLAB and report the estimated parameters, train error, and test error.**

**d. Plot the learning curves of $k$-NN (using $k$ selected in part $b$) and Naive Bayes: this is a graph where the $x$-axis is the number of training examples and the $y$-axis is the accuracy on the test set (i.e., the estimated future accuracy as a function of the amount of training data). To create this graph, randomize the order of your training examples (you only need to do this once). Create a model using the first 10% of training examples, measure the resulting accuracy on the test set, then repeat using the first 20%, 30%, ..., 100% training examples. Compare the performance of two classifiers and summarize your findings.**

Solution:

**a. For Gaussian Naive Bayes classifier with $n$ features for $X$ and $k$ classes for $Y$, we have to estimate the mean $\mu_{ij}$ and variance $\sigma_{ij}^2$ of each feature $i$ conditioned on each class $j$. So we have to estimate $2nk$ parameters. In addition, we need the prior probabilities for $Y$, so there are $k$ such probabilities of $\pi_j = P(Y = j)$, where the last one ($\pi_k$) can be determined from the first $k - 1$ values by $P(Y = k) = 1 - \sum j = 1^{k-1} P(Y = j)$. Therefore, we have $2nk + k - 1$ parameters in total.**

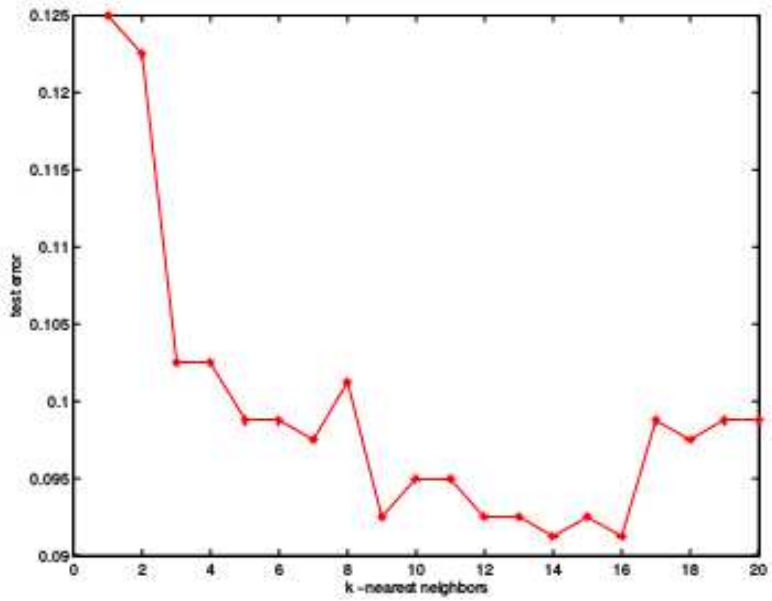
$$\hat{\mu}_{ij} = \frac{\sum_l X_i^{(l)} \, \delta(Y^{(l)} = j)}{\sum_l \delta(Y^{(l)} = j)}$$

$$\hat{\sigma}_{i,j}^2 = \frac{\sum_l (X_i^{(l)} - \mu_{ij})^2 \, \delta(Y^{(l)} = j)}{\sum_l \delta(Y^{(l)} = j)}$$

$$\hat{\pi}_j = \frac{\sum_l \delta(Y^{(l)} = j)}{N}$$

**In the given example where we consider two features with binary labels, we have 8+1=9 parameters. $k$-NN is nonparametric method, and there is no parameter to estimate.**

**b.**

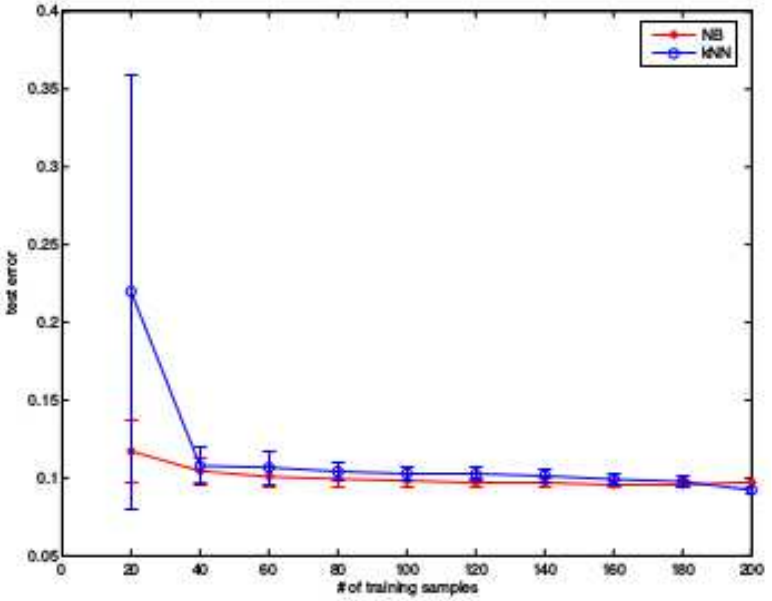



**c.**

$$\hat{\mu}_{ij}^2 = \begin{array}{|c|c|}\hline -0.7438 & 0.9717 \\ \hline -0.9848 & 0.9769 \\ \hline \end{array} \quad \hat{\sigma}_{ij}^2 = \begin{array}{|c|c|}\hline 1.0468 & 0.8861 \\ \hline 0.8889 & 1.1822 \\ \hline \end{array} \quad \hat{\pi}_j = \begin{array}{|c|c|}\hline 0.5100 & 0.4900 \\ \hline \end{array}$$

**The training error was 0.0700, and the test error was 0.0975.**

**d.**



**22.** **($k$-NN applied on hand-written digits from postal zip codes; explore different methods to choose $k$)**

• ○ ***CMU, 2004 fall, Carlos Guestrin, HW4, pr. 3.2-8***

**You will implement a classifier in Matlab and test it on a real data set. The data was generated from handwritten digits, automatically scanned from envelopes by the U.S. Postal Service. Please download the `knn.data` file from the course web page. It contains 364 points. In each row, the first attribute is the class label (0 or 1), the remaining 256 attributes are features (all columns are continuous values). You could use Matlab function `load('knn.data)` to load this data into Matlab.**

**a. Now you will implement a $k$-nearest-neighbor ($k$-NN) classifier using Matlab. For each unknown example, the $k$-NN classifier collects its $k$ nearest neighbors training points, and then takes the most common category among these $k$ neighbors as the predicted label of the test point.**

**We assume our classification is a binary classification task. The class label would be either 0 or 1. The classifier uses the Euclidean distance metric. (But you should keep in mind that normal $k$-NN classifier supports multi-class classification.) Here is the prototype of the Matlab function you need to implement:**

```
function[Y_test] = knn(k, X_train, Y_train, X_test);
```

**`X_train` contains the features of the training points, where each row is a 256-dimensional vector. `Y_train` contains the known labels of the training points, where each row is an 1-dimensional integer either 0 or 1. `X_test` contains the features of the testing points, where each row is a 256-dimensional vector. $k$ is the number of nearest-neighbors we would consider in the classification process.**

**b. For $k = 2, 4, 6, \ldots$, you may encounter ties in the classification. Describe how you handle this situation in your above implementation.**

**c. The choice of $k$ is essential in building the $k$-NN model. In fact, $k$ can be regarded as one of the most important factors of the model that can strongly influence the quality of predictions.**

**One simple way to find $k$ is to use the *train-test* style. Randomly choose 30% of your data to be a test set. The remainder is a training set. Build the classification model on the training set and estimate the future performance with the test set. Try different values of $k$ to find which works best for the testing set.**

**Here we use the *error rate* to measure the performance of a classifier. It equals to the percentage of incorrectly classified cases on a test set.**

**Please implement a Matlab function to implement the above *train-test* way for finding a good $k$ for the kNN classifier. Here is the prototype of the matlab function you need to implement:**

```
function[TestsetErrorRate, TrainsetErrorRate] =
  knn_train_test(kArrayToTry, XData, YData);
```

**`XData` contains the features of the data points, where each row is a 256-dimensional vector. `YData` contains the known labels of the points, where each row is an 1-dimensional integer either 0 or 1. `kArrayToTry` is a $k \times 1$ column vector, containing the $k$ possible values of $k$ you want to try. `TestsetErrorRate` is a $k \times 1$ column vector containing the testing error rate for each possible $k$. `TrainsetErrorRate` is a $k \times 1$ column vector containing the training error rate for each possible $k$.**

**Then test your function knn train test on data set `knn.data`.**

**Report the plot of *train error rate* vs. $k$ and the plot of *test error rate* vs. $k$ for this data. (Make these two curves together in one figure. You could use `hold on` function in Matlab to help you.) What is the best $k$ you would choose according to these two plots?**

**d. Instead of the above *train-test* style, we could also do *$n$-folds Cross-validation* to find the best $k$. $n$-folds Cross-validation is a well established technique that can be used to obtain estimates of model parameters that are unknown. The general idea of this method is to divide the data sample into a number of $n$ folds (randomly drawn, disjointed sub-samples or segments). For a fixed value of $k$, we apply the $k$-NN model to make predictions on the $i$-th segment (i.e., use the $n-1$ segments as the train examples) and evaluate the error. This process is then successively applied to all possible choices of $i$ ($i \in \{1, \ldots, v\}$). At the end of the $n$ folds (cycles), the computed errors are averaged to yield a measure of the stability of the model (how well the model predicts query points). The above steps are then repeated for various $k$ and the value achieving the lowest error rate is then selected as the optimal value for $k$ (optimal in a cross-validation sense).[27]**

**Then please implement a cross-validation function to choose $k$. Here is the prototype of the Matlab function you need to implement:**

```
function[cvErrorRate] = knn_cv(kArrayToTry, XData, YData, numCVFolds);
```

**All the dimensionality of input parameters are the same as in part *c*. `cvErrorRate` is a $k \times 1$ column vector containing the cross validation error rate for each possible $k$.**

**Apply this function on the data set `knn.data` using 10-cross-folds. Report a performance curve of *cross validation error rate* vs. $k$. What is the best $k$ you would choose according to this curve?**

**e. Besides the train-test style and $n$-folds cross validation, we could also use *leave-one-out Cross-validation* (LOOCV) to find the best $k$. LOOCV means omitting each training case in turn and train the classifier model on the remaining $R-1$ datapoints, test on this omitted training case. When you've done all points, report the mean error rate. Implement a LOOCV function to choose $k$ for our $k$-NN classifier. Here is the prototype of the matlab function you need to implement:**

```
function[LoocvErrorRate] = knn_loocv(kArrayToTry, XData, YData);
```

[27] If you want to understand more about cross validation, please look at Andrew Moore's Cross-Validation slides online: http://www-2.cs.cmu.edu/ awm/tutorials/overfit.html.

**All the dimensionality of input parameters are the same as in part $c$. `LoocvErrorRate` is a $k \times 1$ column vector containing LOOCV error rate for each possible $k$.**

**Apply this function on the data set `knn.data` and report the performance curve of LOOCV error rate vs. $k$. What is the best $k$ you would choose according to this curve?**

**f. Compare the four performance curves (from parts $c$, $d$ and $e$). Make the four curves together in one figure here. Can you get some conclusion about the difference between train-test, $n$-folds cross-validation and leave-one-out cross validation?**

***Note*: We provide a Matlab file `TestKnnMain.m` to help you test the above functions. You could download it from the course web site.**

**Solution:**

**b. There are many possible ways to handle this tie case. For example, *i.* choose one of the class; *ii.* use $k-1$ neighbor to decide; *iii.* weighted $k$-NN, etc.**

**c-f. We would get four curves roughly having the similar trend. The best error rate is around 0.02. If you run the program several times, you would find that LOOCV curve would be the same among multiple runs, because it does not have randomness involved. CV curves varies roughly around the LOOCV curve. The *train-test* test curve varies a lot among different runs. But anyway, roughly, as $k$ increases, the error rate increases. From the curve, we can actually choose a small range of $k$ $(1-5)$ as our model selection result.**

**23.** **($k$-NN and SVM: application on a facial attractiveness task)**

- ***CMU, 2007 fall, Carlos Guestrin, HW3, pr. 3***
  ***CMU, 2009 fall, Carlos Guestrin, HW3, pr. 3***

**In this question, you will explore how cross-validation can be used to fit "magic parameters." More specifically, you'll fit the constant $k$ in the $k$-Nearest Neighbor algorithm, and the slack penalty $C$ in the case of Support Vector Machines.**

**Dataset**

**Download the file `hw3_matlab.zip` and unpack it. The file `faces.mat` contains the Matlab variables `traindata` (training data), `trainlabels` (training labels), `testdata` (test data), `testlabels` (test labels) and `evaldata` (evaluation data, needed later).**

**This is a *facial attractiveness* classification task: given a picture of a face, you need to predict whether the average rating of the face is *hot* or *not*. So, each row corresponds to a data point (a picture). Each column is a feature, a pixel. The value of the feature is the value of the pixel in a grayscale image.[28] For fun, try `showface(evaldata(1,:))`, `showface(evaldata(2,:))`, ....**

**`cosineDistance.m` implements the *cosine distance*, a simple distance function. It takes two feature vectors $x$ and $y$, and computes a nonnegative, symmetric distance between $x$ and $y$. To check your data, compute the distance between the first training example from each class. (It should be 0.2617.)**

**A. $k$-NN**

**a. Implement the $k$-Nearest Neighbor ($k$-NN) algorithm in Matlab. *Hint*: You might want to precompute the distances between all pairs of points, to speed up the cross-validation later.**

**b. Implement $n$-fold cross validation for $k$-NN. Your implementation should partition the training data and labels into $n$ parts of approximately equal size.**

**c. For $k = 1, 2, \ldots, 100$, compute and plot the 10-fold (i.e., $n = 10$) cross-validation error for the training data, the training error, and the test error.**

**How do you interpret these plots? Does the value of $k$ which minimizes the cross-validation error also minimize the test set error? Does it minimize the training set error? Either way, can you explain why? Also, what does this tell us about using the training error to pick the value of $k$?**

**B. SVM**

**d. Now download *libsvm* using the link from the course website and unpack it to your working directory. It has a Matlab interface which includes binaries for Windows. It can be used on OS X or Unix but has to be compiled (requires `g++` and `make`) – see the `README` file from the *libsvm* zip package.**

**`hw3_matlab.zip`, which you downloaded earlier, contains files `testSVM.m` (an example demonstration script), `trainSVM.m` (for training) and `classifySVM.m`**

[28] This is an "easier" version of the dataset presented in Ryan White, Ashley Eden, Michael Maire *Automatic Prediction of Human Attractiveness*, CS 280 class report, December 2003 on the project website.

**(for classification), which will show you how to use *libsvm* for training and classifying using an SVM. Run `testSVM`. This should report a test error of 0.4333.**

**In order to train an SVM with slack penalty $C$ on training set `data` with labels `labels`, call**

```
svmModel = trainSVM(data, labels, C)
```

**In order to classify examples `test`, call**

```
testLabels = classifySVM(svmModel, test)
```

**Train an SVM on the training data with $C = 500$, and report the error on the test set.**

**e. Now implement $n$-fold cross-validation for SVMs.**

**f. For $C = 10, 10^2, 10^3, 10^4, 5 \cdot 10^4, 10^5, 5 \cdot 10^5, 10^6$, compute and plot the 10-fold (i.e., $n = 10$) cross-validation error for the training data, the training error, and the test error, with the axis for $C$ in log-scale (try `semilogx`).**

**How do you interpret these plots? Does the value of $C$ which minimizes the cross-validation error also minimize the test set error? Does it minimize the training set error? Either way, can you explain why? Also, what does this tell us about using the training error to pick the value of $C$?**

# 5 Clustering

**24.** **(Hierarchical clustering and $K$-means: application on the *yeast gene expression* dataset)**

● ○ ***CMU, 2010 fall, Carlos Guestrin, HW4, pr. 3***

**Now that you have been in the Machine Learning class for almost 2 months, you have earned the first rank of number crunching machine learner. Recently, you landed a job at the Nexus lab of Cranberry Melon University. Your first task at the lab is to analyze gene expression data (which measures the levels of genes in cells) for some mysterious yeast cells. You are given two datasets: a set of 12 yeast genes in `yeast1.txt`, and a set of 52 yeast genes in `yeast2.txt`.**

**You are told that these genes are critical to solve the "myster" of these cells. You just learnt clustering so you hope that this technique could help you pinpoint groups of genes that may explain the "mystery". The *format* of the files is as follows: the first column lists an identifier for each gene, the second column lists a common name for the gene and a description of its function and the remaining columns list expression values for the gene under various conditions.**

**Your program should not use the gene descriptions when performing clustering. However, it may be informative to see them in the output. These genes belong to four categories for which the genes in each should exhibit fairly similar expression profiles.**

**a. Implement the agglomerative hierarchical clustering that you learnt in class. You cannot use Matlab's linkage function or any function that computes the linkage/tree. Use the Euclidean distance as the distance between expression vectors of two genes. You only need to implement single linkage clustering.**

**Your *output* should be a tab-delimited text file. Each line of the file describes one internal node of the tree: the first column is the identifier of the first node, the second column is the identifier of the second node, and the third column is the linkage between the two nodes. Note that each agglomeration occurs at a greater distance between clusters than the previous agglomeration. For leaf nodes, use the gene name as the identifier and for internal nodes, use the line number where the node was described.**

**To *test* your method, the output using single linkage on the small set is provided in `single1.txt`:**

```
YKL145W YGL048C 2.391171
YFL018C YGR183C 3.383814
YLR038C YLR395C 3.461156
3 2 4.144297
4 1 4.163976
YOR369C YPL090C 4.328152
5 YDL066W 4.463093
6 YOR182C 4.837613
7 YPR001W 5.246656
9 YGR270W 5.373565
10 8 6.050942
```

**Submit the code and following output file `single2.txt` for single linkage clustering of the big set.**

**b. From the output tree, we can get $K$ clusters by cutting the tree at a certain threshold $d$. That is any internal nodes with the linkage greater than $d$ are discarded. The genes are clustered according to the remaining nodes. Implement a function that output $K$ clusters given the value $K$. Your function should find the threshold $d$ automatically from the constructed tree. The output file lists the genes belong to each cluster. Each line of the file contains two columns: the gene identifier (the first column in the original input file) and the description (the second column.) A blank line is used to separated the clusters. For the tree in `single1.txt`, to get 2 clusters, we use the threshold 6.01 to cut the tree. The file `2single1.txt` is an example output file as shown here:**

```
YPL090C RPS6A PROTEIN SYNTHESIS RIBOSOMAL PROTEIN S6A
YOR182C RPS30B PROTEIN SYNTHESIS RIBOSOMAL PROTEIN S30B
YOR369C RPS12 PROTEIN SYNTHESIS RIBOSOMAL PROTEIN S12

YPR001W CIT3 TCA CYCLE CITRATE SYNTHASE
YLR038C COX12 OXIDATIVE PHOSPHORYLATIO CYTOCHROME-C OXIDASE, SUBUNIT VIB
YGR270W YTA7 PROTEIN DEGRADATION 26S PROTEASOME SUBUNIT; ATPASE
YLR395C COX8 OXIDATIVE PHOSPHORYLATIO CYTOCHROME-C OXIDASE CHAIN VIII
YKL145W RPT1 PROTEIN DEGRADATION, UBI 26S PROTEASOME SUBUNIT
YGL048C RPT6 PROTEIN DEGRADATION 26S PROTEASOME REGULATORY SUBUNIT
YDL066W IDP1 TCA CYCLE ISOCITRATE DEHYDROGENASE (NADP+)
YFL018C LPD1 TCA CYCLE DIHYDROLIPOAMIDE DEHYDROGENASE
YGR183C QCR9 OXIDATIVE PHOSPHORYLATIO UBIQUINOL CYTOCHROME-C REDUCTASE SUBUNIT 9
```

**Submit your code and the following tab-delimited output files: `2single2.txt`, `4single2.txt`, `6single2.txt`: 2, 4 and 6 clusters using single linkage on the big dataset.**

**c. Describe another way to get $K$ clusters from the constructed tree. Try to be as succinct as possible. Implement your method. Submit the code and the 3 output files: `2user2.txt`, `4user2.txt`, `6user2.txt`. of running your method on the big set to get 2, 4, 6 clusters respectively.**

**d. Implement $K$-means to cluster these genes. Make sure you use at least 10 random initializations. Submit the code and the 3 output files: `2kmeans2.txt`, `4kmeans2.txt`, `6kmeans2.txt` of running $K$-means on the big set to get 2, 4, 6 clusters respectively.**

**e. We can quantitatively compare the clustering as follows. For each cluster $k$, we can calculate the mean expression values of all genes which we call $m_k$. The residual sum of squares (RSS) is defined as**

$$\sum_i (x_i - m_{c_i})^\top (x_i - m_{c_i})$$

**where $x_i$ is the gene expression of a gene $i$ and $c_i$ is its cluster number.**

**For $K = 2, 4, 6$ clusters, report the RSS of the method in part $b$, your proposed method and $K$-means on the big dataset. Which method is better with respect to RSS?**

**f. Qualitatively compare the result of getting 6 clusters using the method in part $b$, your proposed method with $K$-means on the big dataset. What do you observe? *Hint*: The gene description may give you some clues on what these genes do in the cells.**

**Solution:**

**a. The code is available online in `hcust.m` and `2single.txt`.**

**b. The code is available online in `cuttree.m`.**

**c. One way is to cut the $K$ longest internal edges in the tree. The length of the internal edge = the increase in linkage when the cluster is combined in the next step. This tells how far apart this cluster from the neighbor cluster. The code is available online in `cuttree2.m`.**

**d. The code is available online in `kmean.m`.**

**e. The code is available in `calcRSS.m`. The RSS is reported in the following table. $K$-means performs best in term of RSS.**

| | RSS | | |
|---|---|---|---|
| | $K$-means | CutTree part $b$ | CutTree user |
| K=2 | 839.2200 | 1.6288e+03 | 1.6583e+03 |
| K=4 | 638.7983 | 845.6614 | 1.6498e+03 |
| K=6 | 526.3841 | 854.7895 | 1.6455e+03 |

**f. By looking at the gene annotation, which includes a short description of the gene function, we see that the method in part $b$ provides clusters with more coherent sets of genes.**

**25.** **($K$-means: application to image compression)**

● ○ ***Stanford, 2012 spring, Andrew Ng, HW9***

**In this exercise, you will use $K$-means to compress an image by reducing the number of colors it contains. To begin, download `ex9Data.zip` and unpack its contents into your Matlab/Octave working directory.**

**Photo credit: The bird photo used in this exercise belongs to Frank Wouters and is used with his permission.**

**Image Representation**

**The data pack for this exercise contains a 538-pixel by 538-pixel TIFF image named `bird_large.tiff`. It looks like the picture below.**

**In a straightforward 24-bit color representation of this image, each pixel is represented as three 8-bit numbers (ranging from 0 to 255) that specify red, green and blue intensity values. Our bird photo contains thousands of colors, but we'd like to reduce that number to 16. By making this reduction, it would be possible to represent the photo in a more efficient way by storing only the RGB values of the 16 colors present in the image.**

**In this exercise, you will use $K$-means to reduce the color count to $K = 16$. That is, you will compute 16 colors as the cluster centroids and replace each pixel in the image with its nearest cluster centroid color.**

**Because computing cluster centroids on a $538 \times 538$ image would be time-consuming on a desktop computer, you will instead run $K$-means on the $128 \times 128$ image `bird_small.tiff`.**

**Once you have computed the cluster centroids on the small image, you will then use the 16 colors to replace the pixels in the large image.**

**$K$-means in Matlab/Octave**

**In Matlab/Octave, load the small image into your program with the following command:**

```
A = double(imread('bird_small.tiff'));
```

**This creates a three-dimensional matrix A whose first two indices identify a pixel position and whose last index represents red, green, or blue. For example, A(50, 33, 3) gives you the blue intensity of the pixel at position y = 50, x = 33. (The y-position is given first, but this does not matter so much in our example because the x and y dimensions have the same size).**

**Your task is to compute 16 cluster centroids from this image, with each centroid being a vector of length three that holds a set of RGB values. Here is the $K$-means algorithm as it applies to this problem:**

**$K$-means algorithm**

**1. For initialization, sample 16 colors randomly from the original small picture. These are your $K$ means $\mu_1, \mu_2, \ldots, \mu_K$.**
**2. Go through each pixel in the small image and calculate its nearest mean.**

$$c^{(i)} = \arg\min_j \left\| x^{(i)} - \mu_j \right\|^2$$

**3. Update the values of the means based on the pixels assigned to them.**

$$\mu_j = \frac{\sum_i^m 1_{\{c^{(i)}=j\}} x^{(i)}}{\sum_i^m 1\{c^{(i)} = j\}}$$

**4. Repeat steps 2 and 3 until convergence. This should take between 30 and 100 iterations. You can either run the loop for a preset maximum number of iterations, or you can decide to terminate the loop when the locations of the means are no longer changing by a significant amount.**

***Note*: In Step 3, you should update a mean only if there are pixels assigned to it. Otherwise, you will see a divide-by-zero error. For example, it's possible that during initialization, two of the means will be initialized to the same color (i.e., black). Depending on your implementation, all of the pixels in the photo that are closest to that color may get assigned to one of the means, leaving the other mean with no assigned pixels.**

**Reassigning colors to the large image**

**After $K$-means has converged, load the large image into your program and replace each of its pixels with the nearest of the centroid colors you found from the small image.**

**When you have recalculated the large image, you can display and save it in the following way:**

```
imshow(uint8(round(large_image)))
imwrite(uint8(round(large_image)), 'bird_kmeans.tiff');
```

**When you are finished, compare your image to the one in the solutions.**

**Solution:**

**Here are the 16 colors appearing in the image:**

**26.** **($K$-means: how to select $K$ and the initial centroids (the $K$-means++ algorithm); the importance of scaling the data across different dimensions)**

• ○ ***CMU, 2012 fall, E. Xing, A. Singh, HW3, pr. 1***

**In $K$-means clustering, we are given points $x_1, \ldots, x_n \in \mathbb{R}^d$ and an integer $K > 1$, and our goal is to minimize the within-cluster sum of squares (also known as the $K$-means objective)**

$$J(C, L) = \sum_{i=1}^{n} ||x_i - C_{l_i}||^2,$$

**where $C = (C_1, \ldots, C_K)$ are the cluster centers ($C_j \in \mathbb{R}^d$), and $L = (1, \ldots, l_n)$ are the cluster assignments ($i \in \{1, \ldots, K\}$).**

**Finding the exact minimum of this function is computationally difficult. The most common algorithm for finding an approximate solution is Lloyd's algorithm, which takes as input the set of points and some initial cluster centers $C$, and proceeds as follows:**

*i.* **Keeping $C$ fixed, find cluster assignments $L$ to minimize $J(C, L)$. This step only involves finding nearest neighbors. Ties can be broken using arbitrary (but consistent) rules.**

*ii.* **Keeping $L$ fixed, find $C$ to minimize $J(C, L)$. This is a simple step that only involves averaging points within a cluster.**

*iii.* **If any of the values in $L$ changed from the previous iteration (or if this was the first iteration), repeat from step *i*.**

*iv.* **Return $C$ and $L$.**

**The initial cluster centers $C$ given as input to the algorithm are often picked randomly from $x_1, \ldots, x_n$. In practice, we often repeat multiple runs of Lloyd's algorithm with different initializations, and pick the best resulting clustering in terms of the $k$-means objective. You're about to see why.**

**a. Briefly explain why Lloyd's algorithm is always guaranteed to converge (i.e., stop) in a finite number of steps.**

**b. Implement Lloyd's algorithm. Run it until convergence 200 times, each time initializing using $K$ cluster centers picked at random from the set $\{x_1, \ldots, x_n\}$, with $K = 5$ clusters, on the 500 two dimensional data points in .... Plot in a single figure the original data (in gray), and all 200 $\times$ 5 cluster centers (in black) given by each run of Lloyd's algorithm. You can play around with the plotting options such as point sizes so that the cluster centers are clearly visible. Also compute the minimum, mean, and standard deviation of the within-cluster sums of squares for the clusterings given by each of the 200 runs.**

**b. $K$-means++ is an initialization algorithm for $K$-means proposed by David Arthur and Sergei Vassilvitskii in 2007:**

*i.* **Pick the first cluster center $C_1$ uniformly at random from the data $x_1, \ldots, x_n$. In other words, we first pick an index i uniformly at random from $\{1, \ldots, n\}$, then set $C_1 = x_i$.**

*ii.* **For $j = 2, \ldots, K$:**

- **For each data point, compute its distance $D_i$ to the nearest cluster center picked in a previous iteration:**

$$D_i = \min_{j'=1,\ldots,j-1} ||x_i - C_{j'}||.$$

- **Pick the cluster center $C_j$ at random from $x_1, \ldots, x_n$ with probabilities proportional to $D_1^2, \ldots, D_n^2$. Precisely, we pick an index $i$ at random from $\{1, \ldots, n\}$ with probabilities equal to $D_1^2/(\sum_{i'=1}^n D_{i'}^2), \ldots, D_n^2/(\sum_{i'=1}^n D_{i'}^2)$, and set $C_j = x_i$.**

*iii.* **Return $C$ as the initial cluster assignments for Lloyd's algorithm.**

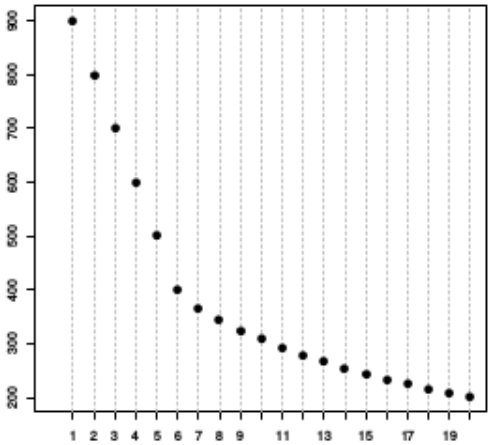
**Replicate the figure and calculations in part $b$ using $K$-means++ as the initialization algorithm, instead of picking $C$ uniformly at random.**[29]

**Picking the number of clusters $K$ is a difficult problem. Now we will see one of the most common heuristics for choosing $K$ in action.**

**d. Explain how the exact minimum of the $K$-means objective behaves on any data set as we increase $K$ from 1 to $n$.**

**A common way to pick $K$ is as follows. For each value of $K$ in some range (e.g., $K = 1, \ldots, n$, or some subset), we find an approximate minimum of the $K$-means objective using our favorite algorithm (e.g., multiple runs of randomly initialized Lloyd's algorithm). Then we plot the resulting values of the $K$-means objective against the values of $K$.**

**Often, if our data set is such that there exists a natural value for $K$, we see a "knee" in this plot, i.e., a value for $K$ where the rate at which the within-cluster sum of squares is decreasing sharply reduces. This suggests we should use the value for $K$ where this knee occurs. In the toy example in th enearby figure, this value would be $K = 6$.**




**e. Produce a plot similar to the one in the above figure for $K = 1, \ldots, 15$ using the data set in part $b$, and show where the "knee" is. For each value of $K$, run $K$-means with at least 200 initializations and pick the best resulting clustering (in terms of the objective) to ensure you get close to the global minimum.**

**f. Repeat part $e$ with the data set in ..... Find 2 knees in the resulting plot (you may need to plot the square root of the within-cluster sum of squares instead, in order to make the second knee obvious). Explain why we get 2 knees for this data set (consider plotting the data to see what's going on).**

[29] Hopefully your results make it clear how sensitive Lloyd's algorithm is to initializations, even in such a simple, two dimensional data set!

**We conclude our exploration of $K$-means clustering with the critical importance of properly scaling the dimensions of your data.**

**g. Load the data in .... Perform $K$-means clustering on this data with $K = 2$ with 500 initializations. Plot the original data (in gray), and overplot the 2 cluster centers (in black).**

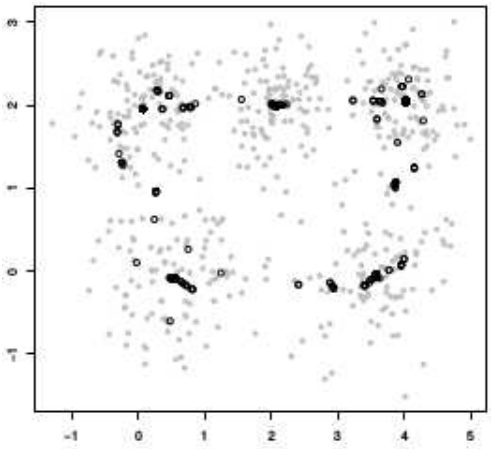
**h. Normalize the features in this data set, i.e., first center the data to be mean 0 in every dimension, then rescale each dimension to have unit variance. Repeat part $g$ with this modified data.**

**As you can see, the results are radically different. You should not take this to mean that data should always be normalized. In some problems, the relative values of the dimensions are meaningful and should be preserved (e.g., the coordinates of earthquake epicenters in a region). But in others, the dimensions are on entirely different scales (e.g., age in years v.s., income in thousands of dollars). Proper pre-processing of data for clustering is often part of the art of machine learning.**

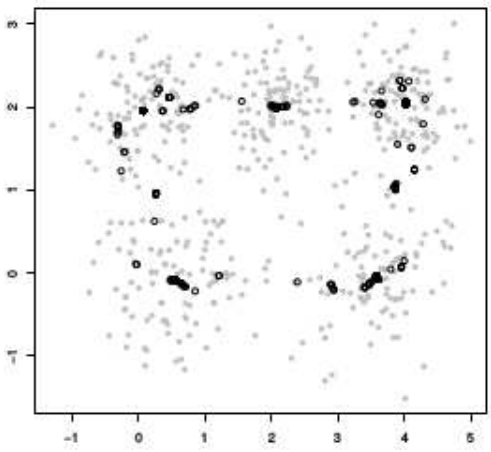
Solution:

**a. The cluster assignments $L$ can take finitely many values ($K^n$, to be precise). The cluster centers $C$ are uniquely determined by the assignments $L$, so after executing step $ii$, the algorithm can be in finitely many possible states. Thus either the algorithm stops in finitely many steps, or at least one value of $L$ is repeated more than once in non-consecutive iterations. However, the latter case is not possible, since after every iteration we have $J(C(t), L(t)) \geq J(C(t+1), L(t+1))$, with equality only when $L(t) = L(t+1)$, which coincides with the termination condition. (Note that this statement depends on the assumption that the tie-breaking rule used in step $i$ is consistent, otherwise infinite loops are possible.)**

**b. Minimum: 222.37, mean: 249.66, standard deviation: 65.64.**
**Plot in the nearby figure.**
**R code: see file `kmeans.r` on the web course page.**

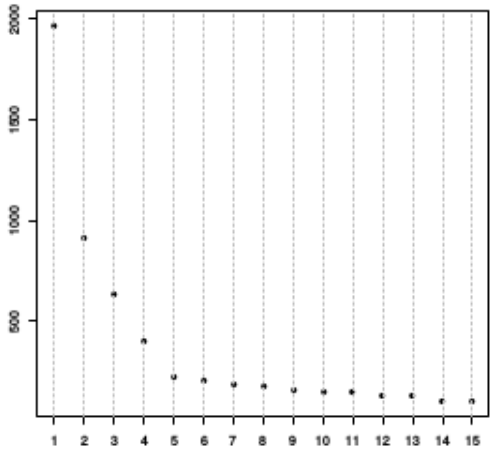


**c. Minimum: 222.37, mean: 248.33, standard deviation: 64.96.**
**Plot in the nearby figure.**
**R code: see file `kmeans++.r` on the web course page.**

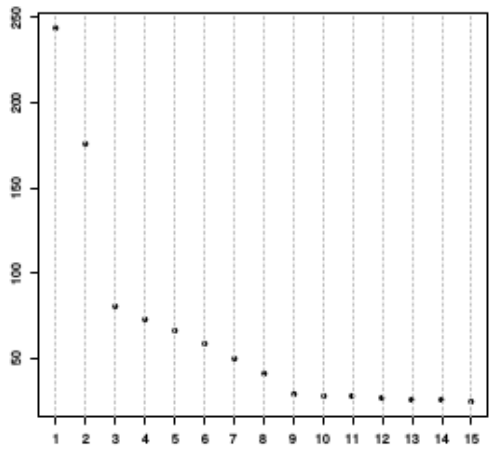


**d. The exact minimum decreases (or stays the same) as $K$ increases, because the set of possible clusterings for K is a subset of the possible clusterings for $K+1$. With $K=n$, the objective of the optimal solution is 0 (every point is in its own cluster, and has 0 distance to the cluster center).**

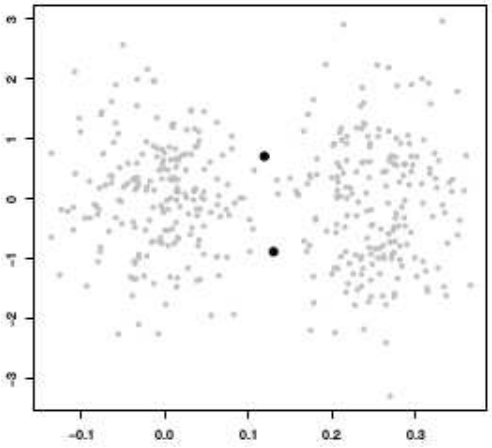
**e. Plot in the nearby figure. The knee is at $K=5$.**



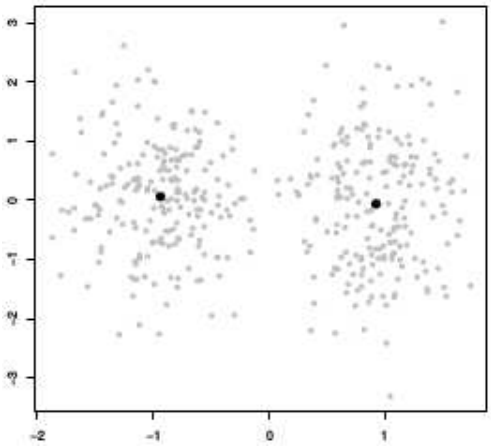
**f. Plot in the nearby figure (square root of objective plotted).**
**The knees are at $K=3$ and $K=9$.**
**These are two values because the data are composed of 3 natural clusters, each of which can further be divided into 3 smaller clusters.**



**g.**




**h.**



**27.** **(EM/GMM: implementation in Matlab and application on data from $\mathbb{R}^1$)**

• · ***CMU, 2010 fall, Aarti Singh, HW4, pr. 2.3-5***

**a. Implement the EM/GMM algorithm using the update equations derived in the exercise ??, the Clustering chapter in Ciortuz et al's book).**

**b. Download the data set from ... Each row of this file is a training instance $x^i$. Run your EM/GMM implementation on this data, using $\mu = [1, 2]$ and $\theta = [.33, .67]$ as your initial parameters. What are the final values of $\mu$ and $\theta$? Plot a histogram of the data and your estimated mixture density $P(X)$. Is the mixture density an accurate model for the data?**

**To plot the density in Matlab, you can use:**

```
density = @(x) (<class 1 prior> * normpdf(x, <class 1 mean>, 1)) + ...
               (<class 2 prior> * normpdf(x, <class 2 mean>, 1));
fplot(density, [-5, 6]);
```

**Recall from class that EM attempts to maximize the marginal data loglikelihood $\ell(\mu, \theta) = \sum_{i=1}^{n} \log P(X = x^i; \mu, \theta)$, but that EM can get stuck in local optima. In this part, we will explore the shape of the loglikelihood function and determine if local optima are a problem. For the remainder of the problem, we will assume that both classes are equally likely, i.e., $\theta_y = \frac{1}{2}$ for $y = 0, 1$. In this case, the data loglikelihood $\ell$ only depends on the mean parameters $\mu$.**

**c. Create a contour plot of the loglikelihood $\ell$ as a function of the two mean parameters, $\mu$. Vary the range of each $\mu_k$ from $-1$ to $4$, evaluating the loglikelihood at intervals of $.25$. You can create a contour plot in Matlab using the `contourf` function. Print out your plot and include in with your solution.**

**Does the loglikelihood have multiple local optima? Is it possible for EM to find a non-globally optimal solution? Why or why not?**

**28.** **($K$-means and EM/GMM: comparison on data from $\mathbb{R}^2$)**

• · ***CMU, 2010 spring, E. Xing, T. Mitchell, A. Singh, HW3, pr. 3***

**Clustering means partitioning your data into "natural" groups, usually because you suspect points in a cluster have something in common. The EM algorithm and $K$-means are two common algorithms (there are many others). This problem will have you implement these algorithms, and explore their limitations.**

**The datasets for you to use are available online, along with a Matlab script for loading them. Ask me if you're having any trouble with it. You can use any language for your implementations, but you may not use libraries which already implement these algorithms (you can, however, use fancy built-in mathematical functions, like Matlab or Mathematica provide).**

**a. In $K$-means clustering, the goal is to pick your clusters such that you minimize the sum, over all points $x$, of $|x - c_x|^2$, where $c_x$ is the mean of the cluster containing $x$; this should remind you of least-squares line fitting. $K$-means clustering is NP-hard, but in practice this algoritm, also called Lloyd's algorithm, works extremely well.**

**Implement Lloyd's algorithm, and apply it to the datasets provided. Plot each dataset, indicating for each point which cluster is was placed in. How well do you think k-means did for each dataset? Explain, intuitively, what (if anything) went badly and why.**

**b. A disadvantage of $K$-means is that the clusters cannot overlap at all. Expectation maximization deals with this by only probabilistically assigning points to clusters.**

**The thing to understand about the EM algorithm is that it's a special case of MLE; you have some data, you assume a parameterized form for the probability distribution (a mixture of Gaussians is, after all, an exotic parameterized probability distribution), and then you pick the parameters to maximize the probability of your data. But the usual MLE approach, solving $\frac{\partial P(X|\theta)}{\partial \theta} = 0$, isn't tractable, so we use the iterative EM algorithm to find $\theta$. The EM algorithm is guaranteed to converge to a local optimum (I'm resisting the temptation to make you prove this :) ).**

**Implement the EM algorithm, and apply it to the datasets provided. Assume that the data is a mixture of two Gaussians; you can assume equal mixing ratios. What parameters do you get for each dataset? Plot each dataset, indicating for each point which cluster it was placed in.**

**c. Modeling dataset 2 as a mixture of gaussians is unrealistic, but the EM algorithm still gives *an* answer. Is there anything "fishy" about your answers which suggests something is wrong?**

**We usually do the EM algorithm with mixed Gaussians, but you can use any distributions; a Gaussian and a Laplacian, three exponentials, etc. Write down the formula for a parameterized probability density suitable for modeling "ring-shaped" clusters in 2D; don't let the density be 0 anywhere. You don't need to work out the EM calculations for this density, but you would if this came up in your research.**

**With high-dimensional data we cannot perform visual checks, and problems can go unnoticed if we assume nice round, filled clusters. Describe in words a clustering algorithm which works even for weirdly-shaped clusters with unknown mixing ration. However, you can assume that the clsuters do not overlap at all, and that you have a LOT of training data. Discuss the weaknesses of your algorithm. Don't work out the details for this problem; just convince me that you know the basic idea and understand its limitations.**

**29.** **(EM for mixtures of Gaussians with independent components (along axis): application to handwritten digit recognition)**

- ***CMU, 2012 spring, Ziv Bar-Joseph, HW4, pr. 3.2***

**In this problem we will be implementing Gaussian mixture models and working with the digits data set. The provided data set is a Matlab file consisting of 5000 10×10 pixel hand written digits between 0 and 9. Each digit is a greyscale image represented as a 100 dimensional row vector (the images have been down sampled from the original 28×28 pixel images. The variable $X$ is a 5000×100 matrix and the vector $Y$ contains the true number for each image. Please submit your code and include in your write-up a copy of the plots that you generated for this problem.**

**a. Implement the Expectation-Maximization (EM) algorithm for the axis aligned Gaussian mixture model. Recall that the axis aligned Gaussian mixture model uses the Gaussian Naive Bayes assumption that, given the class, all features are conditionally independent Gaussians. The specific form of the model is given below:**

$$Z_i \sim Categorical(p_1, \ldots, p_K)$$

$$X_i | Z_i = z \sim \mathcal{N}\left( \begin{bmatrix} \mu_1^z \\ \vdots \\ \mu_d^z \end{bmatrix}, \begin{bmatrix} (\sigma_1^z)^2 & 0 & \ldots & 0 \\ 0 & (\sigma_2^z)^2 & & \vdots \\ \vdots & & \ddots & 0 \\ 0 & \ldots & 0 & (\sigma_d^z)^2 \end{bmatrix} \right)$$

**b. Run EM to fit a Gaussian mixture model with 16 Gaussians on the digits data. Plot each of the means using `subplot(4, 4, i)` to save paper.**

**c. Evaluating clustering performance is difficult. However, because we have information about the ground truth data, we can roughly assess clustering performance. One possible metric is to label each cluster with the majority label for that cluster using the ground truth data. Then, for each point we predict the cluster label and measure the mean 0/1 loss. For the digits data set, report your loss for settings $k = 1$, 10 and 16.**

**30.** **(Aplicarea algoritmului EM [LC; pentru GMM] la clusterizare de documente, folosind sistemul WEKA)**

• ***Edingurgh, Chris Williams and Victor Lavrenko***
***Introductory Applied Machine Learning course, 3 Nov. 2008***

**A. Description of the dataset**

**This assignment is based on the 20 Newsgroups Dataset.[30] This dataset is a collection of approximately 20,000 newsgroup documents, partitioned (nearly) evenly across 20 different newsgroups, each corresponding to a different topic. Some of the newsgroups are very closely related to each other (e.g.** comp.sys.ibm.pc.hardware, comp.sys.mac.hardware**), while others are highly unrelated (e.g.** misc.forsale, soc.religion.christian**).**

**There are three versions of the 20 Newsgroups Dataset. In this assignment we will use the bydate Matlab version in which documents are sorted by date into training (60%) and test (40%) sets, newsgroup-identifying headers are dropped and duplicates are removed. This collection comprises roughly 61,000 different words, which results in a bag-of-words representation with frequency counts. More specifically, each document is represented by a 61,000 dimensional vector that contains the counts for each of the 61,000 different words present in the respective document.**

**To save you time and to make the problem manageable with limited computational resources, we preprocessed the original dataset. We will use documents from only 5 out of the 20 newsgroups, which results in a 5-class problem. More specifically the 5 classes correspond to the following newsgroups 1:**alt.atheism**, 2:**comp.sys.ibm.pc.hardware**, 3:**comp.sys.mac.hardware**, 4:**rec.sport.baseball **and 5:**rec.sport.hockey**. However, note here that classes 2-3 and 4-5 are rather closely related. Additionally, we computed the mutual information of each word with the class attribute and selected the 520 words out of 61,000 that had highest mutual information. Therefore, our dataset is a N $\times$ 520 dimensional matrix, where N is the number of documents.**

**The resulting representation is much more compact and can be used directly to perform our experiments in WEKA. There is, however, a potential caveat: The preprocessed dataset has been prepared by a busy and heavily underpaid teaching assistant who might have been a bit careless when preparing the dataset. You should keep this in mind and be aware of anomalies in the data when answering the questions below.**

**B. Clustering**

**We are interested in clustering the newsgroups documents using the EM algorithm. The most common measure to evaluate the resulting clusters is the log-likelihood of the data. We will additionally use the *Classes to Clusters* evaluation which is straightforward to perform in WEKA, and look at the percentage of correctly clustered instances. Note here that the data likelihood computed during EM is a probability density (NOT a probability mass function) and therefore the log-likelihood can be greater than** 0**. Use the** train_20news_clean_best_tfidf.arff **dataset and the *default seed* (**100**) to train the clusterers.**

[30] http://people.csail.mit.edu/jrennie/20Newsgroups/

**a. First, train and evaluate an EM clusterer with 5 clusters (you need to change the *numClusters* option) using the *Classes to Clusters* evaluation option. Report the log-likelihood and write down the percentage of correctly clustered instances (PC), you will need it in question *b*. Look at the *Classes to Clusters* confusion matrix. Do the clusters correspond to classes? Which classes are more confused with each other? Interpret your results. Keep the result buffer for the clusterer, you will need it in question *c*.**

**HINT: WEKA outputs the percentage of incorrectly classified instances.**

**b. Now, train and evaluate different EM clusterers using 3, 4, 6 and 7 clusters and the *Classes to Clusters* evaluation option. Tabulate the PC as a function of the number of clusters, include the PC for 5 clusters from the previous question. What do you notice? Why do you think we get higher PC for 3 clusters than for 4? Keep the result buffers for all the clusterers, you will need them in question *c*.**

**c. Re-evaluate the five clusterers using the validation set** val_20news_best_tfidf.arff. **Tabulate the log-likelihood on the validation set as a function of the number of clusters. If the dataset was unlabeled, how many clusters would you choose to model it in light of your results? Is it safe to make this decision based on experiments with only one random seed? Why?**

**HINT: To re-evaluate the models, first select the *Supplied test set* option and choose the appropriate dataset. Then right-click on the model and select *Re-evaluate model on current test set*.**

**d. Now consider the model with 5 clusters learned using EM. After EM converges, each cluster is described in terms of the mean and standard deviation for each of the 500 attributes computed from the documents assigned to the respective cluster. Since the attributes are the normalized tf-idf weights for each word in a document, the mean vectors learned by EM correspond to the tf-idf weights for each word in each cluster.**

**For each of the 5 clusters, we selected the 20 attributes with the highest mean values. Open the file** cluster_means.txt. **The 20 attributes for each cluster are displayed columnwise together with their corresponding mean value. By looking at the words with the highest tf-idf weights per cluster, which column (cluster) would you assign to each class (newsgroup topic) and why? Which two clusters are closest to each other? Imagine that we want to assign a new document to one of the clusters and that the document contains only the words "pitching" and "hit". Would this be an easy task for the clusterer? What about a document that contains only the words "drive" and "mac"? Write down three examples of 2-word documents that would be difficult test cases for the clusterer.**

**31.** **(EM for GMM: application on the yeast gene expression dataset)**

● ○ *CMU, 2004 fall, Carlos Guestrin, HW2, pr. 3*

**In this problem you will implement a Gaussian mixture model algorithm and will apply it to the problem of clustering gene expression data. Gene expression measures the levels of messenger RNA (mRNA) in the cell. The data you will be working with is from a *model organism* called *yeast*, and the measurements were taken to study the cell cycle system in that organism. The cell cycle system is one of the most important biological systems playing a major role in development and cancer.**

**All implementation should be done in Matlab. At the end of each sub-problem where you need to implement a new function we specify the prototype of the function.**

**The file `alphaVals.txt` contains 18 time points (every 7 minutes from 0 to 119) measuring the log expression ratios of 745 cycling genes. Each row in this file corresponds to one of the genes. The file `geneNames.txt` contains the names of these genes. For some of the genes, we are missing some of their values due to problems with the *microarray* technology (the tools used to measure gene expression). These cases are represented by values greater than 100.**

**a. Implement (in Matlab) an EM algorithm for learning a mixture of five (18-dimensional) Gaussians. It should learn means, covariance matrices and weights for each of the Gaussian. You can assume, however, independence between the different data points [LC: correct features/attributes], resulting in a *diagonal covariance matrix*. How can you deal with the missing data? Why is this correct?**

**Plot the centers identified for each of the five classes. Each center should be plotted as a time-series of 18 time points.**

**Here is the prototype of the Matlab function you need to implement:**

```
function[mu; s;w] = emcluster(x; k; ploton);
```

**where**

**– `x` is input data, where each row is an 18-dimensional sample. Values above 100 represent missing values;**

**– `k` is the number of desired clusters;**

**– `ploton` is either 1 or 0. If 1, then before returning the function plots log-likelihood of the data after each EM iteration (the function will have to store the log-likelihood of the data after each iteration, and then plot these values as a function of iteration number at the end). If 0, the function does not plot anything;**

**– `s` is a `k` by 18 matrix, with each row being diagonal elements of the corresponding covariance matrix;**

**– `w` is a column vector of size `k`, where `w(i)` is a weight for `i`-th cluster.**

**The function outputs `mu`, a matrix with `k` rows and 18 columns (each row is a center of a cluster).**

**b. How many more parameters would you have had to assign if we remove the independence assumption above? Explain.**

**c. Suggest and implement a method for determining the number of Gaussians (or classes) that are the most appropriate for this data. Please confine the set of choices to values in between 2 and 7. (*Hint*: The method can use an empirical evaluation of clustering results for each possible number of classes). Explain the method.**

**Here is the prototype of the Matlab function you need to implement:**

```
function[k, mu, s, w] = clust(x);
```

**where**

**– `x` is input data, where each row is an 18-dimensional sample. Once again values above 100 represent missing values;**

**– `k` is the number of classes selected by the function;**

**– `mu`, `s` and `w` are defined as in part $a$.**

**d. Use the Gaussians determined in part $d$ to perform hard clustering of your data by finding, for each gene $i$ the Gaussian $j$ that maximizes the likelihood: $P(i|j)$. Use the function `printSelectedGenes.m` to write the names of the genes in each of the clusters to a separate file.**

**Here is the prototype of the matlab function you need to implement:**

```
function[c] = hardclust(x; k; mu; s;w);
```

**where**

**– `x` is defined as before;**

**– `k`, `mu`, `s`, `w` are the output variables from the function written in part $c$ and are therefore defined there;**

**– `c` is a column vector of the same length as the number of rows in $x$. For each row, it should indicate the cluster the corresponding gene belongs to.**

**The function should also write out files as specified above. The filenames should be: `clust1`, `clust2`, ..., `clustk`.**

**e. Use `compSigClust.m` to perform the statistical significance test (everything is already implemented here, so just use the function). Hand in a printout with the top three categories for each cluster (this is the output of `compSigClust.m`).**

**Solution:**

**a. We have put a student code online. The implementation is pretty clear in terms of each step of the GMM iteration. The plot of the log-likelihood should be increasing. The plots of the centers of each cluster should look like a sinusoid shape though with different phases (starting at a different point in the time series).**

**b. The number of clusters times the number of covariances, which is**

$$k((d-1)+(d-2)+\ldots+1) = \frac{kd}{2}(d-1),$$

**where $d = 18$ in our case.**

**c. This is essentially a *model selection* question. You could use different model selection ways to solve it: cross validation, train-test, *minimum description length*.**

**d. For each data point, assign the cluster that has the maximum probability for this point.**

**e. Just run the code we provided on the cluster files you got above.**

# 6 EM Algorithm

**32.** **(EM for Bernoulli MM, using the Naive Bayes assumption, and a penalty term; application to handwritten digit recognition)**

● ○ *U. Toronto, Radford Neal, "Statistical Methods for Machine Learning and Data ining" course, 2014 spring, HW 2*

**In this assignment, you will classify handwritten digits with mixture models fitted by maximum penalized likelihood using the EM algorithm. The data you will use consists of 800 training images and 1000 test images of handwritten digits (from US zip codes). We derived these images from the well-known MNIST dataset, by randomly selecting images from the total 60000 training cases provided, reducing the resolution of the images from $28 \times 28$ to $14 \times 14$ by averaging $2 \times 2$ blocks of pixel values, and then thresholding the pixel values to get binary values. A data file with 800 lines each containing 196 pixel values (either 0 or 1) is provided on webpage associated to this book. Another file containing the labels for these 800 digits (0 to 9) is also provided. Similarly, there is a file with 1000 test images, and another file with the labels for these 1000 test images. You should look at the test labels only at the very end, too see how well the methods do.**

**In this assignment, you should try to classify these images of digits using a *generative model*, from which you can derive the probabilities of the 10 possible classes given the observed image of a digit. You should guess that the class for a test digit is the one with highest probability (i.e., we will use a loss function in which all errors are equally bad).**

**The generative model we will use estimates the class probabilities by their frequencies in the training set (which will be close to, but not exactly, uniform over the 10 digits) and estimates the probability distributions of images within each class by mixture models with $K$ components, with each component modeling the 196 pixel values as being independent. It will be convenient to combine all 10 of these mixture models into a single mixture model with $10K$ components, which model both the pixel values and the class label. The probabilities for class labels in the components will be fixed, however, so that $K$ components give probability 1 to digit 0, $K$ components give probability 1 to digit 1, $K$ components give probability 1 to digit 2, etc.**

**The model for the distribution of the label, $y_i$, and pixel values $x_{i,1}, \ldots, x_{i,196}$, for digit $i$ is therefore as follows:**

$$P(y_i, x_i) = \sum_{k=1}^{10K} \pi_k q_{k,y_i} \prod_{j=1}^{196} \theta_{k,j}^{x_{i,j}} (1 - \theta_{k,j})^{1 - x_{i,j}}$$

**The data items, $(y_i, x_i)$, are assumed to be independent for different cases $i$. The parameters of this model are the mixing proportions, $\pi_1, \ldots, \pi_{10K}$, and the probabilities of pixels being 1 for each component, $\theta_{k,j}$ for $k = 1, \ldots, 10K$ and**

$j = 1, \ldots, 196$. **The probabilities of class labels for each component are fixed, as**

$$q_{k,y} = \begin{cases} 1 & \text{if } k \in \{Ky+1, \ldots, Ky+K\} \\ 0 & \text{otherwise} \end{cases}$$

**for** $k = 1, \ldots, 10K$ **and** $y = 0, \ldots, 9$.

**You should write an R function to try to find the parameter values that maximize the log-likelihood from the training data plus a "penalty". (Note that with this "penalty" higher values are better.) The EM algorithm can easily be adapted to find maximum penalized likelihood estimates rather than maximum likelihood estimates – referring to the general version of the algorithm, the E step remains the same, but the M step will now maximize** $E_Q[\log P(x, z|\theta) + G(\theta)]$, **where** $G(\theta)$ **is the "penalty".**

**The "penalty" to use is designed to avoid estimates for pixel probabilities that are zero or close to zero, which could cause problems when classifying test cases (for example, zero pixel probabilities could result in a test case having zero probability for every possible digit that it might be). The penalty to add to the log likelihood should be**

$$G(\theta) = \alpha \sum_{k=1}^{10K} \sum_{j=1}^{196} [\log(\theta_{k,j}) + \log(1 - \theta_{k,j})].$$

**Here,** $\alpha$ **controls the magnitude of the penalty. For this assignment, you should fix** $\alpha$ **to 0.05, though in a real application you would probably need to set it by some method such as cross-validation. The resulting formula for the update in the M step is**

$$\hat{\theta}_{k,j} = \frac{\alpha + \sum_{i=1}^{n} r_{i,k} x_{i,j}}{2\alpha + \sum_{i=1}^{n} r_{i,k}}$$

**where** $r_{i,k}$ **is the probability that case** $i$ **came from component** $k$**, estimated in the E step. You should write a derivation of this formula from the general form of the EM algorithm presented in the lecture slides (modified as above to include a penalty term).**

**Your function implementing the EM algorithm should take as arguments the images in the training set, the labels for these training cases, the number of mixture components for each digit class (**$K$**), the penalty magnitude (**$\alpha$**), and the number of iterations of EM to do. It should return a list with the parameter estimates (**$\pi$ **and** $\theta$**) and responsibilities (**$r$**). You will need to start with some initial values for the responsibilities (and then start with an M step). The responsibility of component** $k$ **for item** $i$ **should be zero if component** $k$ **has** $q_{k,y_i} = 0$**. Otherwise, you should randomly set** $r_{i,k}$ **from the uniform distribution between 1 and 2 and then rescale these values so that for each** $i$**, the sum over** $k$ **of** $r_{i,k}$ **is one.**

**After each iteration, your EM function should print the value of the log-likelihood and the value of the log likelihood plus the penalty function. The latter should never go down – if it does, you have a bug in your EM function. You should use enough iterations that these values have almost stabilized by the last iteration.**

**You will also need to write an R function that takes the fitted parameter values from running EM and uses them to predict the class of a test image. This**

**function should use Bayes' Rule to find the probability that the image came from each of the $10K$ mixture components, and then add up the probabilities for the $K$ components associated with each digit, to obtain the probabilities of the image being of each digit from 0 to 9. It should return these probabilities, which can then be used to guess what the digit is, by finding the digit with the highest probability.**

**You should first run your program EM and prediction functions for $K = 1$, which should produce the same results as the naive Bayes method would. (Note that EM should converge immediately with $K = 1$.) You should then do ten runs with $K = 5$ using different random number seeds, and see what the predictive accuracy is for each run. Finally, for each test case, you should average the class probabilities obtained from each of the ten runs, and then use these averaged probabilities to classify the test cases. You should compare the accuracy of these "ensemble" predictions with the accuracy obtained using the individual runs that were averaged.**

**You should hand in your derivation of the update formula for $\hat{\theta}$ above, a listing of the R functions you wrote for fitting by EM and predicting digit labels, the R scripts you used to apply these functions to the data provided, the output of these scripts, including the classification error rates on the test set you obtained (with $K = 1$, with $K = 5$ for each of ten initializations, and with the ensemble of ten fits with $K = 5$), and a discussion of the results. Your discussion should consider how naive Bayes ($K = 1$) compares to using a mixture (with $K = 5$), and how the ensemble predictions compare with predicting using a single run of EM, or using the best run of EM according to the log likelihood (with or without the penalty).**

**Solution:**

**With $K = 1$, which is equivalent to a naive Bayes model, the classification error rate on test cases was 0.190.**

**With $K = 5$, 80 iterations of EM seemed sufficient for all ten random initializations. The resulting models had the following error rates on the test cases:**

0.157 0.151 0.158 0.156 0.166 0.162 0.163 0.159 0.158 0.153

**These are all better than the naive Bayes result, showing that using more than one mixture component for each digit is beneficial.**

**I used the "show_digit" function to display the theta parameters of the 50 mixture components as pictures (for the run started with the last random seed). It is clear that the five components for each digit have generally captured reasonable variations in writing style, except perhaps for a few with small mixing proportion (given as the number above the plot), such as the second "1" from the top.**

**Using the ensemble predictions (averaging probabilities of digits over the ten runs above), the classification error rate on test cases was 0.139. This is substantially better than the error rate from every one of the individual runs, showing the benefits of using an ensemble when there is substantial random variation in the results.**

**Note that the individual run with highest log likelihood (and also highest log likelihood + penalty) was the sixth run, whose error rate of 0.162 was actually**

**the third worst. So at least in this example, picking a single run based on log likelihood would certainly not do better than using the ensemble.**

**33.** **(EM for a mixture of two exponential distributions)**

• · *U. Toronto, Radford Neal, "Statistical Computation" course, 2000 fall, HW 4*

Suppose that the time from when a machine is manufactured to when it fails is is exponentially distributed (a common, though simplistic, assumption). However, suppose that some machines have a manufacturing defect that causes them to be more likely to fail early than machines that don't have the defect.

Let the probability that a machine has the defect be $p$, the mean time to failure for machines without the defect be $g$, and the mean time to failure for machines with the defect be $d$. The probability density for the time to failure will then be the following mixture density:

$$p \cdot \frac{1}{\mu_d} \cdot \exp\left(-\frac{x}{\mu_d}\right) + (1-p) \cdot \frac{1}{\mu_g} \exp\left(-\frac{x}{\mu_g}\right)$$

Suppose that you have a number of independent observations of times to failure for machines, and that you wish to nd maximum likelihood estimates for $p$, $\mu_g$, and $\mu_d$. Write a program to find these estimates using the EM algorithm, with the unobserved variables being the indicators of whether or not each machine is defective. Note that the model is not identifiable — swapping $\mu_d$ and $\mu_g$ while replacing $p$ with $1-p$ has no effect on the density. This isn't really a problem; you can just interpret whichever mean is smaller as the mean for the defective machines.

You may write your program so that it simply runs for however many iterations you specify (i.e., you don't have to come up with a convergence test). However, your program should have the option of printing the parameter estimates and the log likelihood at each iteration, so that you can manually see whether it has converged. (This will also help debugging.)

You should test your program on two data sets (ass4a.data and ass4b), each with 1000 observations, which are on the web page associated to this book. You can read this data with a command like

```
> x <- scan("ass4a.data")
```

For both data sets, run your algorithm for as long as you need to to be sure that you have obtained close to the correct maximum likelihood estimates. To be sure, we recommend that you run it for hundreds of iterations or more (this shouldn't take long in R). Discuss how rapidly the algorithm converges on the two data sets. You may find it useful to generate your own data sets, for which you know the true parameters values, in order to debug your program. You could start with data sets where $\mu_g$ and $\mu_d$ are very different.

You should hand in your derivation of the formulas neeed for the EM algorithm, your program, the output of your tests, and your discussion of the results.

# 7 Artificial Neural Networks

**34.** **(Reţele neuronale: chestiuni de bază)**

• ***MPI, 2005 spring, Jörg Rahnenfuhrer, Adrian Alexa, HW2, pr. 5***

**a. Simulate data by randomly drawing two-dimensional data points uniformly distributed in $[0,1]^2$. The class $Y$ of a sample $X = (X_1, X_2)$ is 1 if $X_1 + X_2 > 1$ and $-1$ otherwise. You can add some noise to the data by using the rule $X_1 + X_2 > 1 + \varepsilon$ with $\varepsilon \sim \mathcal{N}(0, 0.1)$. Use 100 samples for the training set (you can use training data with or without noise).**

**b. Write a function to train a perceptron for a two class classification problem. A perceptron is a classifier which constructs a linear decision boundary that tries to separate the data into different classes as best as possible. Section 4.5.1 in *The Elements of Statistical Learning* book by Hastie, Tibshirani, Friedman describes how the perceptron learning algorithm works.**

**c. Write a function to predict the class for new data points. Write a function that performs LOOCV (Leave-One-Out Cross-Validation) for your classifier. Use these functions to estimate the train error and the the prediction error. Generate a test set of 1000 samples and compute the test error of your classifier.**

**d. Use the data generator `xor.data()` from the tutorial homepage to generate a new training (ca. 100 samples) and test set (ca. 1000 samples). Train the perceptron on this data. Report the train and test errors. Plot the test samples to see how they are classified by the perceptron.**

**e. Comment your findings. Is the perceptron able to learn the XOR data? What is the main difference between the data generated in part $a$ and the data from part $d$?**

**f. Use the `nnet` R package to train a neural network for the XOR data. The `nnet()` function is fitting a neural network. Use the `predict()` function to asses the train and test errors.**

**h. Vary the number of units in the hidden layer and report the train and test errors. We have seen at part $e$ that a perceptron, a neural network with no unit in the hidden layer can not correctly classify the XOR data. Argue which is the minimal number of units in the hidden layer that a neural network must have to correctly classify the XOR data. Train such a network and report the train, prediction and test errors.**

**35.** **(The Perceptron algorithm: spam identification)**

- ***New York University, 2016 spring, David Sontag, HW1***

**In this problem set you will implement the *Perceptron* algorithm and apply it to the problem of e-mail spam classification.**

***Instructions.*** **You may use the programming language of your choice (we recommend Python, and using `matplotlib` for plotting). However, you are not permitted to use or reference any machine learning code or packages not written by yourself.**

***Data files.*** **We have provided you with two files: `spam train.txt`, and `spam test.txt`. Each row of the data files corresponds to a single email. The first column gives the label (1=spam, 0=not spam).**

***Pre-processing.*** **The dataset included for this exercise is based on a subset of the SpamAssassin Public Corpus. Figure 1 shows a sample email that contains a URL, an email address (at the end), numbers, and dollar amounts. While many emails would contain similar types of entities (e.g., numbers, other URLs, or other email addresses), the specific entities (e.g., the specific URL or specific dollar amount) will be different in almost every email. Therefore, one method often employed in processing emails is to "normalize" these values, so that all URLs are treated the same, all numbers are treated the same, etc. For example, we could replace each URL in the email with the unique string "httpaddr" to indicate that a URL was present. This has the effect of letting the spam classifier make a classification decision based on whether any URL was present, rather than whether a specific URL was present. This typically improves the performance of a spam classifier, since spammers often randomize the URLs, and thus the odds of seeing any particular URL again in a new piece of spam is very small.**

**We have already implemented the following email preprocessing steps: lowercasing; removal of HTML tags; normalization of URLs, e-mail addresses, and numbers. In addition, words are reduced to their stemmed form. For example, "discount", "discounts", "discounted" and "discounting" are all replaced with "discoun. Finally, we removed all non-words and punctuation. The result of these preprocessing steps is shown in Figure 2.**

**Figure 1: Sample e-mail in SpamAssassin corpus before pre-processing.**
**> Anyone knows how much it costs to host a web portal?**
**> Well, it depends on how many visitors youre expecting. This can be anywhere from less than 10 bucks a month to a couple of $100. You should checkout http://www.rackspace.com/ or perhaps Amazon EC2 if youre running something big.**
**To unsubscribe yourself from this mailing list, send an email to: groupname-unsubscribeegroups.com.**

**Figure 2: Pre-processed version of the sample e-mail from Figure 1.**
**anyon know how much it cost to host a web portal well it depend on how mani visitor your expect thi can be anywher from less than number buck a month to a coupl of dollarnumb you should checkout httpaddr or perhap amazon ecnumb if your run someth big to unsubscrib yourself from thi mail list send an email to emailaddr**

**a. This problem set will involve your implementing several *variants of the Perceptron algorithm*. Before you can build these models and measure their performance, split your training data (i.e. `spam_train.txt`) into a training and validation set, putting the last 1000 emails into the validation set. Thus, you will have a new training set with 4000 emails and a validation set with 1000 emails. You will not use `spam_test.txt` until problem $j$. Explain why measuring the performance of your final classifier would be problematic had you not created this validation set.**

**b. Transform all of the data into feature vectors. Build a vocabulary list using only the 4000 e-mail training set by finding all words that occur across the training set. Note that we assume that the data in the validation and test sets is completely unseen when we train our model, and thus we do not use any information contained in them. Ignore all words that appear in fewer than $X = 30$ e-mails of the 4000 e-mail training set – this is both a means of preventing overfitting and of improving scalability. For each email, transform it into a feature vector $\bar{x}$ where the $i$th entry, $x_i$, is 1 if the $i$th word in the vocabulary occurs in the email, and 0 otherwise.**

**c. Implement the functions `perceptron_train(data)` and `perceptron_test(w, data)`.**
**The function `perceptron_train(data)` trains a perceptron classifier using the examples provided to the function, and should return $\bar{w}$, $k$, and $iter$, the final classification vector, the number of updates (mistakes) performed, and the number of passes through the data, respectively. You may assume that the input data provided to your function is linearly separable (so the stopping criterion should be that all points are correctly classified). For the corner case of $w \cdot x = 0$, predict the $+1$ (spam) class.**
**For this exercise, you do not need to add a bias feature to the feature vector (it turns out not to improve classification accuracy, possibly because a frequently occurring word already serves this purpose). Your implementation should cycle through the data points in the order as given in the data files (rather than randomizing), so that results are consistent for grading purposes.**
**The function `perceptron_test(w, data)` should take as input the weight vector $\bar{w}$ (the classification vector to be used) and a set of examples. The function should return the test error, i.e. the fraction of examples that are misclassified by $\bar{w}$.**

**d. Train the linear classifier using your training set. How many mistakes are made before the algorithm terminates? Test your implementation of `perceptron_test` by running it with the learned parameters and the training data, making sure that the training error is zero. Next, classify the emails in your validation set. What is the validation error?**

**e. To better understand how the spam classifier works, we can inspect the parameters to see which words the classifier thinks are the most predictive of spam. Using the vocabulary list together with the parameters learned in the previous question, output the 15 words with the *most positive* weights. What are they? Which 15 words have the *most negative* weights?**

**f. Implement the *averaged* perceptron algorithm, which is the same as your current implementation but which, rather than returning the final weight vector, returns the average of all weight vectors considered during the algorithm**

**(including examples where no mistake was made). Averaging reduces the variance between the different vectors, and is a powerful means of preventing the learning algorithm from overfitting (serving as a type of regularization).**

**g. One should expect that the test error decreases as the amount of training data increases. Using only the first $N$ rows of your training data, run both the perceptron and the averaged perceptron algorithms on this smaller training set and evaluate the corresponding validation error (using all of the validation data). Do this for $N = 100, 200, 400, 800, 2000, 4000$, and create a plot of the validation error of both algorithms as a function of $N$.**

**h. Also for $N = 100, 200, 400, 800, 2000, 4000$, create a plot of the number of perceptron iterations as a function of $N$, where by iteration we mean a complete pass through the training data. As the amount of training data increases, the margin of the training set decreases, which generally leads to an increase in the number of iterations perceptron takes to converge (although it need not be monotonic).**

**i. One consequence of this is that the later iterations typically perform updates on only a small subset of the data points, which can contribute to overfitting. A way to solve this is to control the maximum number of iterations of the perceptron algorithm. Add an argument to both the perceptron and averaged perceptron algorithms that controls the maximum number of passes over the data.**

**j. Congratulations, you now understand various properties of the perceptron algorithm. Try various configurations of the algorithms on your own using all 4000 training points, and find a good configuration having a low error on your validation set. In particular, try changing the choice of perceptron algorithm and the maximum number of iterations. You could additionally change $X$ from question $b$ (this is optional). Report the validation error for several of the configurations that you tried; which configuration works best?**
**You are ready to train on the full training set, and see if it works on completely new data. Combine the training set and the validation set (i.e. use all of `spam_train.txt`) and learn using the best of the configurations previously found. You do not need to rebuild the vocabulary when re-training on the train+validate set.**
**What is the error on the test set (i.e., now you finally use spam test.txt)?**

***Note*: This problem set is based partly on an assignment developed by Andrew Ng of Stanford University and Coursera.**

**36.** **(The Perceptron algorithm: application on the Breast Cancer dataset)**

• ○ ***(CMU, 2009 spring, Ziv Bar-Joseph, HW2, pr. 4.3-5)***

**For this exercise, you will use the Breast Cancer dataset, downloadable from the course web page (...). Given 9 different attributes, such as "uniformity of cell size", the *task* is to predict malignancy.[31] The archive from the course web page contains a Matlab method `loaddata.m`, so you can easily load in the data by typing (from the directory containing `loaddata.m`): `data = loaddata`. The variables in the resulting data structure relevant for you are:**

- **`data.X`: 683 9-dimensional data points, each element in the interval [1, 10].**
- **`data.Y`: the 683 corresponding classes, either 0 (benign), or 1 (malignant).**

**A very simple and popular linear classifier is the perceptron algorithm of Rosenblatt (1962), a single-layer neural network model of the form**

$$y(x) = f(w^\top x),$$

**with the activation function**

$$\begin{cases} f(a) = 1 & \text{if } a \geq 0 \\ f(a) = -1 & \text{otherwise.} \end{cases}$$

**For this classifier, we need our classes to be $-1$ (benign) and $1$ (malignant), which can be achieved with the Matlab command: `data.Y = data.Y * 2 - 1`.**

**Weight training usually proceeds in an online fashion, iterating through the individual data points $x^j$ one or more times. For each $x^j$, we compute the predicted class $\hat{y}^j = f(w^\top x^j)$ for $x^j$ under the current parameters $w$, and update the weight vector as follows:**

$$w \leftarrow w + x^j(y^j - \hat{y}^j)$$

**Note how $w$ only changes if $x_j$ was misclassified under the current model.**

**a. Implement this training algorithm in Matlab. To avoid dealing with intercept explicitly, augment each point in `data.X` with a nonzero constant tenth element. In Matlab this can be done by typing: `data.X(:,10)=1`. Have your algorithm iterate through the whole training data 20 times and report the number of examples that were still mis-classified in the 20th iteration. Does it look like the training data is linearly separable? (*Hint*: The perceptron algorithm is guaranteed to converge if the data is linearly separable.)**

**b. To test your program, use 10-fold cross-validation, splitting [`data.X` `data.Y`] into 10 random approximately equal-sized portions, training on 9 concatenated parts, and testing on the remaining part. For each split, do 20 training iterations to train the weights. Report the mean classification accuracy over the 10 runs and the 95% *confidence interval*.**

**c. If the data is not linearly separable, weights can toggle back and forth from iteration to iteration. Even in the linearly separable case, the learned model is often very dependent on which training data points come first in the training sequence. A simple improvement is the *weighted perceptron*: Training**

[31] For more information on what the individual attributes mean, see ftp://ftp.ics.uci.edu/pub/machine-learning-databases/breast-cancer-wisconsin/breastcancer-wisconsin.names.

**proceeds as before, but the weight vector $w$ is saved after each update. After training, instead of the final $w$, the average of all saved $w$ is taken to be the learned weight vector. Report 10-fold CV accuracy for this variant and compare it to the simple perceptron's.**

**Solution:**

**You should have gotten something like this:**

**a. 30 mis-classifications in 20th iteration (note that using the trained weights *after* the 20th iteration results in only around 24 mis-classifications. When running with 200 iterations, still more than 20 mis-classifications occur, so the data is unlikely to be linearly separable as otherwise the training error would become zero after many enough iterations.**

**b. [Perceptron:] Mean accuracy = 0.956. Confidence Interval: (0.940618, 0.971382).**

**b. [Weighted Perceptron:] Mean accuracy = 0.968. Confidence Interval: (0.954800, 0.981200).**

**37.** **(The backpropagation algorithm: application on the Breast Cancer dataset)**

• ***MPI, 2005 spring, Jörg Rahnenfuhrer, Adrian Alexa, HW2, pr. 6***

**Download the breast cancer data set breastcancer.zip from the tutorial homepage. The data are described in Mike West et al.: *Predicting the clinical status of human breast cancer by using gene expression profiles*, PNAS 98(20):11462-11467, 2001. The file contains expression profiles of 46 patients from two different classes: 23 patients are estrogen receptor positive (ER+) and 23 are estrogen receptor negative (ER-). For every patient, the expression values of 7129 genes were measured. Use the `nnet` R package to train a neural network.**

**a. Load `breastcancer.Rdata` and apply `summary()` to get an overview of this data object: `breastcancer$x` contains the expression data and `breastcancer$y` the class labels. Reformat the data by transposing the gene expression matrix and renaming the classes {ER+, ER−} to $\{+1, -1\}$.**

**b. Train a Neural Network using the `nnet()` function. Check if the inputs are standardized (mean zero and standard deviation one) and if this is not the case, standardize them.**

**c. Apply the function predict() to the training data and calculate the training error. Perform a LOOCV to estimate the prediction error (you must implement by yourself the cross validation procedure).**

**d. Predict the classes of the three new patients (`newpatients`). The true class labels are stored in (`trueclasses`). Are they correctly classified?**

**e. Try different parameters in the nnet() function (the number of units in the hidden layer, the weights, the activation function, the weight decay parameter, etc.) and report the parameters for which you obtained the best result. Comment the way the parameters affect the performance of the network.**

**38.** **(Artificial neural networks: Digit classification competition)**

- ***CMU, 2014 fall, William Cohen, Ziv Bar-Joseph, HW3***

**In this section, you are asked to construct a neural network using a dataset in real world. The training samples and training labels are provided in the handout folder. Each sample is a $28 \times 28$ gray scale image. Each pixel (feature) is a real value between 0 and 1 denoting the pixel intensity. Each label is a integer from 0 to 9 which corresponds to the digit in the image.**

**A. Getting Started**

**Getting Familiar with Data**

**As mentioned above, each sample is an image with 784 pixels. Load the data using the following command:**
`load('digits.mat')`
**Visualize an image using the following command:**
`imshow(vec2mat(XTrain(i,:),28)')`
**where $X \in \mathbb{R}^{n \times 784}$ is the training samples; $i$ is the row index of a training sample.**

**Neural Network Structure**

**In this competition, you are free to use any neural network structure. A simple feed forward neural network with one hidden layer is shown in Figure... . The input layer has a bias neuron and 784 neurons with each corresponding to one pixel in the image. The output layer has 10 neurons with each representing the probability of each digit given the image. You need to decide the size of the hidden layer.**

**Code Structure**

**You should implement your training algorithm (typically the forward propagation and back propagation) in `train_ann.m` and testing algorithm (using trained weights to predict labels) in `test_ann.m`. In your training algorithm, you need to store your initial and final weights into a mat file. In the simple example below, two weight matrices $W_{ih}$ and $W_{ho}$ are stored into `weights.mat`:**
`save('weights.mat','Wih','Who');`
**Be sure your `test ann.m` runs fast enough. It is always good to vectorize your code in Matlab.**

**Separating Data**

**`Digits.mat` contains 3000 instances which you used in previous section. The number of instances are pretty balanced for each digit so you do not need to worry about skewness of the data. However, you need to handle the overfitting problem. Neural networks are very powerful models which are capable to express extremely complicated functions but very prone to overfit.**

**The standard approach for building a model on a dataset can be described as follows:**

- **Divide your data into three sets: a training set, a validation set, a test set. You can use any sizes for three sets as long as they are reasonable (e.g. 60%, 20%, 20%). You can also combine the training set and the validation set and do k-fold cross-validation. Make sure to have balanced numbers of instances for each class in every set.**

- Train your model on the training set and tune your parameters on the validation set. By tuning the parameters (e.g. number of neurons, number of layers, regularization, etc...) to achieve maximum performance on the validation set, the overfitting problems can be somehow alleviated. The following webpage provides some reasonable ranges for parameter selection:
  `http://en.wikibooks.org/wiki/Artificial_Neural_Networks/Neural_Network_Basics`
- If the training accuracy is much higher than validation accuracy, the model is overfitting; if the training accuracy and validation accuracy are both very low, the model is underfitting; if both accuracies are high but test accuracy is low, the model should be discarded.

### B. Bag of Tricks for Training a Neural Network

#### Overfitting vs Underfitting

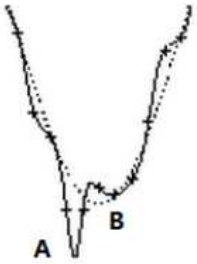
This is related to the model selection problem [that we are going to discuss later in this course]. It is extremely important to determine whether the model is overfitting or underfitting. The table below shows several general approaches to discover and alleviate these problems:

| | Overfit | Underfit |
|---|---|---|
| Performance | Training accuracy much higher than validation accuracy | Both accuracies are low |
| Data | Need more data | If two accuracies are close, no need for extra data |
| Model | Use a simpler model | Use a more complicated model |
| Features | Reduce number of features | Increase number of features |
| Regularization | Increase regularization | Reduce regularization |

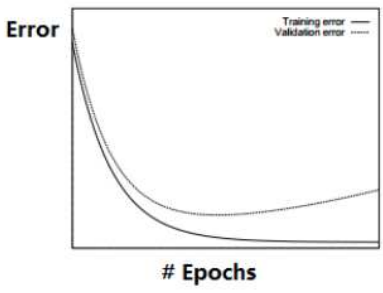
There are other ways to reduce overfitting and underfitting problems particular for neural networks, and we will discuss them in other tricks.

#### Early Stopping

A common reason for overfitting is that the neural net converges to a bad minimum. In the nearby figure, the solid line corresponds to the error surface of a trained neural net while the dash line corresponds to the true model. Point A is very likely to be a bad minimum since the "narrow valley" is very likely to be caused by overfitting to training data. Point B is a better minimum since it is much smoother and more likely be the true minimum.




To alleviate overfitting, we can stop the training process before the network converges. In the nearby figure, if the training procedure stops when network achieves best performance on validation set, the overfitting problem is somehow reduced. However, in reality, the error surface may be very irregular. A *common approach* is to store the weights after each epoch until the network converges. Pick the weights that performs well on the validation set.



### Multiple Initialization

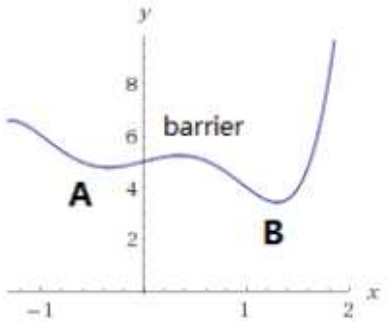
When training a neural net, people typically initialize weights to very small numbers (e.g. a Gaussian random number with 0 mean and 0.005 variance). This process is called *symmetry breaking*. If all the weights are initialized to zero, all the neurons will end up learning the same feature. Since the error surface of neural networks is highly non-convex, different weight initializations will potentially converge to different minima. You should store the initialized weights into `ini weights.mat`.

### Momentum

Another way to escape from a bad minimum is adding a momentum term into weight updates. The momentum term is $\alpha\Delta W(n-1)$ in equation 1, where $n$ denotes the number of epochs. By adding this term to the update rule, the weights will have some chance to escape from minimum. You can set initial momentum to zero.

$$\Delta W(n) = \nabla_W J(W, b) + \alpha\Delta W(n-1) \tag{1}$$

The intuition behind this approach is the same as this term in physics systems. In the nearby figure, assume weights grow positively during training, without the momentum term, the neural net will converge to point A. If we add the momentum term, the weights may jump over the barrier and converge to a better minimum at point B.




### Batch Gradient Descent vs Stochastic Gradient Descent

As we discussed in the lectures, given enough memory space, batch gradient descent usually converges faster than stochastic gradient descent. However, if working on a large dataset (which exceeds the capacity of memory space), stochastic gradient descent is preferred because it uses memory space more efficiently. Mini-batch is a compromise of these two approaches.

### Change Activation Function

As we mentioned in the theoretical questions, there are many other activation functions other than logistic sigmoid activation, such as (but not limited to) rectified linear function, arctangent function, hyperbolic function, Gaussian function, polynomial function and softmax function. Each activation has different expressiveness and computation complexity. The selection of activation function is problem dependent. Make sure to calculate the gradients correctly before implementing them.

### Pre-training

Autoencoder is a unsupervised learning algorithm to automatically learn features from unlabeled data. It has a neural network structure with its input being exactly the same as output. From input layer to hidden layer(s), the features are abstracted to a lower dimensional space. Form hidden layer(s) to output layer, the features are reconstructed. If the activation is linear, the

network performs very similar to Principle Component Analysis (PCA). After training an autoencoder, you should keep the weights from input layer and hidden layers, and build a classifier on top of hidden layer(s). For implementation details, please refer to Andrew Ng's cs294A course handout at Stanford: `http://web.stanford.edu/class/cs294a/sparseAutoencoder_2011new.pdf`

#### More Neurons vs Less Neurons

As mentioned above, we should use more complicated models for underfitting cases, and simpler models for overfitting cases. In terms of neural networks, more neurons mean higher complexity. You should pick the size of hidden layer based on training accuracy and validation accuracy.

#### More Layers?

Adding one or two hidden layers may be useful, since the model expressiveness grows exponentially with extra hidden layers. You can apply the same back propagation technique as training a single hidden layer network. However, if you use even more layers (e.g. 10 layers), you are definitely going to get extremely bad results. Any networks with more than one hidden layer is called a *deep network*. Large deep network encounters the "vanishing gradient" problem using the standard back propagation algorithm (except *convolutional neural nets*). If you are not familiar with convolutional neural nets, or *training stacks* of *Restricted Boltzmann Machines*, you should stick with a few hidden layers.

#### Sparsity

Sparsity on weights (LASSO penalty) forces neurons to learn localized information. Sparsity on activations (KL-divergence penalty) forces neurons to learn complicated features.

#### Other Techniques

All the tricks above can be applied to both shallow networks and deep networks. If you are interested, there are other tricks which can be applied to (usually deep) neural networks:

- Dropout
- Model Averaging

[You can find these information in coursera lectures provided by Geoffrey Hinton.]

**39.** **(Perceptronul Rosenblatt: calcul "mistake bounds"; calcul margini; comparaţie cu SVM)**

• ○ ***MIT, 2006 fall, Tommy Jaakkola, HW1, pr. 1-2***

**Implement a Perceptron classifier in MATLAB. Start by implementing the following functions:**

**– a function `perceptron_train(X, y)` where `X` and `y` are $n \times d$ and $n \times 1$ matrices respectively. This function trains a Perceptron classifier on a training set of $n$ examples, each of which is a $d$-dimensional vector. The labels for the examples are in `y` and are $1$ or $-1$. The function should return [`theta`, `k`], the final classification vector and the number of updates performed, respectively. You may assume that the input data provided to your function is linearly separable. Training the Perceptron should stop when it makes no errors at all on the training data.**

**– a function `perceptron_test(theta, X_test, y_test)` where `theta` is the classification vector to be used. `X_test` and `y_test` are $m \times d$ and $m \times 1$ matrices respectively, corresponding to $m$ test examples and their true labels. The function should return `test err`, the fraction of test examples which were misclassified.**

**For this problem, we have provided you two custom-created datasets. The dimension $d$ of both the datasets is 2, for ease of plotting and visualization.**

**a. Load data using the load p1 a script and train your Perceptron classifier on it. Using the function `perceptron_test`, ensure that your classifier makes no errors on the training data. What is the angle between `theta` and the vector $(1,0)^\top$? What is the number of updates $k_a$ required before the Perceptron algorithm converges?**

**b. Repeat the above steps for data loaded from script `load_p1_b`. What is the angle between `theta` and the vector $(1,0)^\top$ now? What is the number of updates $k_b$ now?**

**c. For parts $a$ and $b$, compute the geometric margins, $\gamma^a_{geom}$ and $\gamma^b_{geom}$, of your classifiers with respect to their corresponding training datasets. Recall that the distance of a point $x_t$ from the $\theta^\top x = 0$ is $|\frac{\theta^\top x_t}{||\theta||}|$.**

**d. For parts $a$ and $b$, compute $R_a$ and $R_b$, respectively. Recall that for any dataset $\chi$, $R = \max\{||x|| | x \in \chi\}$.**

**e. Plot the data (as points in the X-Y plane) from part $a$, along with decision boundary that your Perceptron classifier computed. Create another plot, this time using data from part $b$ and the corresponding decision boundary. Your plots should clearly indicate the class of each point (e.g., by choosing different colors or symbols to mark the points from the two classes). We have a provided a MATLAB function `plot_points_and_classifier` which you may find useful.**

**Implement an SVM classifier in MATLAB, arranged like the [above] Perceptron algorithm, with functions `svm_train(X, y)` and `svm_test(theta, X test, y test)`. Again, include a printout of your code for these functions.**

***Hint*: Use the built-in quadratic program solver `quadprog(H, f, A, b)` which solves the quadratic program: $\min \frac{1}{2}x^\top Hx + f^\top x$ subject to the constraint $Ax \leq b$.**

**f. Try the SVM on the two datasets from parts $a$ and $b$. How different are the values of theta from values the Perceptron achieved? To do this comparison, should you compute the difference between two vectors or something else?**

**g. For the decision boundaries computed by SVM, compute the corresponding geometric margins (as in part $c$). How do the margins achieved using the SVM compare with those achieved by using the Perceptron?**

**40.** **(Kernelized perceptron)**

● ○ ***MIT, 2006 fall, Tommy Jaakkola, HW2, pr. 3***

**Most linear classifiers can be turned into a kernel form. We will focus here on the simple perceptron algorithm and use the resulting kernel version to classify data that are not linearly separable.**

**a. First we need to turn the perceptron algorithm into a form that involves only inner products between the feature vectors. We will focus on hyperplanes through origin in the feature space (any offset component [LC: is assumed to be] provided as part of the feature vectors). The mistake driven parameter updates are:** $\theta \leftarrow \theta + y_t \phi(x_t)$ **if** $y_t \theta^\top \phi(x_t) \leq 0$, **where** $\theta = 0$ **initially. Show that we can rewrite the perceptron updates in terms of simple additive updates on the discriminant function** $f(x) = \theta^\top \phi(x)$**:**

$$f(x) \leftarrow f(x) + y_t K(x_t, x) \text{ if } y_t f(x_t) \leq 0,$$

**where** $K(x_t, x) = \phi(x_t)^\top \phi(x)$ **is any kernel function and** $f(x) = 0$ **initially.**

**b. We can replace** $K(x_t, x)$ **with any kernel function of our choice such as the radial basis kernel where the corresponding feature mapping is infinite dimensional. Show that there always is a separating hyperplane if we use the radial basis kernel.** ***hint*****: Use the answers to the previous exercise in this homework (MIT, 2006 fall, Tommy Jaakkola, HW2, pr. 2).**

**c. With the radial basis kernel we can therefore conclude that the perceptron algorithm will converge (stop updating) after a finite number of steps for any dataset with distinct points. The resulting function can therefore be written as**

$$f(x) = \sum_{i=1}^{n} w_i y_i K(x_i, x)$$

**where** $w_i$ **is the number of times we made a mistake on example** $x_i$**. Most of** $w_i$**'s are exactly zero so our function won't be difficult to handle. The same form holds for any kernel except that we can no longer tell whether the** $w_i$**'s remain bounded (problem is separable with the chosen kernel). Implement the new kernel perceptron algorithm in MATLAB using a radial basis and polynomial kernels. The data and helpful scripts are provided in...**

**Define functions**
`alpha = train_kernel_perceptron(X, y, kernel_type)` **and**
`f = discriminant_function(alpha, X, kernel_type, X_test)`
**to train the pereptron and to evaluate the resulting** $f(x)$ **for test examples, respectively.**

**d. Load the data using the** `load_p3_a` **script. When you use a polynomial kernel to separate the classes, what degree polynomials do you need? Draw the decision boundary (see the provided script** `plot_dec_boundary`**) for the lowest-degree polynomial kernel that separates the data. Repeat the process for the radial basis kernel. Briefly discuss your observations.**

**41.** **(Convolutional nueral networks: implementation and application on the MNIST dataset)**

- ***CMU, 2016 fall, N. Balcan, M. Gormley, HW6***
  ***CMU, 2016 spring, W. Cohen, N. Balcan, HW7***

**In this assignment, we are going to implement a Convolution Neural Network (CNN) to classify hand written digits of MNIST[32] data. Since the breakthrough of CNNs on ImageNet classification (A. Krizhevsky, I. Sutskever, G. E. Hinton, 2012), CNNs have been widely applied and achieved the state the art of results in many areas of computer vision. The recent AI programs that can beat humans in playing Atari game (V. Mnih, K. Kavukcuoglu et al, 2015) and Go (D. Silver, A. Huang et al, 2016) also used CNNs in their models.**

**We are going to implement the earliest CNN model, LeNet (Y. LeCun, L. Bottou, Y. Bengio, P. Haffner, 1998), that was successfully applied to classify hand written digits. You will get familiar with the workflow needed to build a neural network model after this assignment.**

**The Stanford CNN course[33] and UFLDL[34] material are excellent for beginners to read. You are encouraged to read some of them before doing this assignment.**

**We begin by introducing the basic structure and building blocks of CNNs. CNNs are made up of layers that have learnable parameters including weights and bias. Each layer takes the output from previous layer, performs some operations and produces an output. The final layer is typically a softmax function which outputs the probability of the input being in different classes. We optimize an objective function over the parameters of all the layers and then use stochastic gradient descent (SGD) to update the parameters to train a model.**

**Depending on the operation in the layers, we can divide the layers into following types:**

**1. *Inner product layer (fully connected layer)***

**As the name suggests, every output neuron of inner product layer has full connection to the input neurons. The output is the multiplication of the input with a weight matrix plus a bias offset, i.e.:**

$$f(x) = Wx + b. \tag{2}$$

**This is simply a linear transformation of the input. The weight parameter $W$ and bias parameter $b$ are learnable in this layer. The input $x$ is $d$ dimensional column vector, and $W$ is a $d \times n$ matrix and $b$ is $n$ dimensional column vector.**

**2. *Activation layer***

**We add nonlinear activation functions after the inner product layers to model the non-linearity of real data. Here are some of the popular choices for nonlinear activation:**

- **Sigmoid:** $\sigma(x) = \dfrac{1}{(1 + e^{-x})}$;

[32] http://yann.lecun.com/exdb/mnist/
[33] http://cs231n.github.io/
[34] http://ufldl.stanford.edu/tutorial/

- **tanh:** $\tanh(x) = \frac{(e^{2x}-1)}{(e^{2x}+1)}$;
- **ReLU:** $\text{relu}(x) = \max(0, x)$.

**Rectified Linear Unit (ReLU) has been found to work well in vision related problems. There is no learnable parameters in the ReLU layer. In this homework, you will use ReLU, and a recently proposed modification of it called Exponential Linear Unit (ELU).**

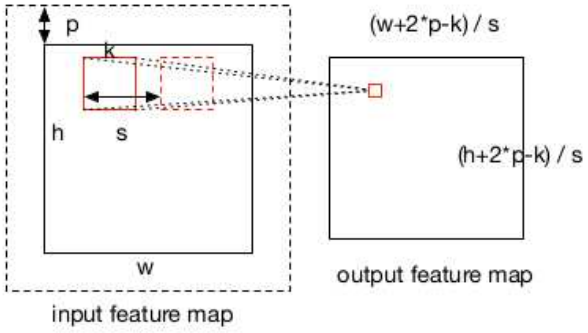
**Note that the activation is usually combined with inner product layer as a single layer, but here we separate them in order to make the code modular.**

**3.** ***Convolution layer***

**The convolution layer is the core building block of CNNs. Unlike the inner product layer, each output neuron of a convolution layer is connected only to some input neurons. As the name suggest, in the convolution layer, we apply convolution operations with filters on input feature maps (or images). In image processing, there are many types of kernels (filters) that can be used to blur, sharpen an image or detect edges in an image. Read the Wikipedia page[35] page if you are not familiar with the convolution operation.**

**In a convolution layer, the filter (or kernel) parameters are learnable and we want to adapt the filters to data. There is also more than one filter at each convolution layer. The input to the convolution layer is a three dimensional tensor (and is often referred to as the *input feature map* in the rest of this document), rather than a vector as in inner product layer, and it is of the shape $h \times w \times c$, where $h$ is the height of each input image, $w$ is the width and $c$ is the number of channels. Note that we represent each channel of the image as a different slice in the input tensor.**

**The nearby figure shows the detailed convolution operation. The input is a feature map, i.e., a three dimensional tensor with size $h \times w \times c$. The convolution operation involves applying filters on this input. Each filter is a sliding window, and the output of the convolution layer is the sequence of outputs produced by each of those filters during the sliding operation.**




**Let us assume each filter has a square window of size $k \times k$ per channel, thus making filter size $k \times k \times c$. We use $n$ filters in a convolution layer, making the number of parameters in this layer $k \times k \times c \times n$. In addition to these parameters, the convolution layer also has two hyper-parameters: the padding size $p$ and stride step $s$. In the sliding window process described above, the output from each filter is a function of a neighborhood of input feature map. Since the edges have fewer neighbors, applying a filter directly is not feasible. To avoid this problem, inputs are typically padded (with zeros) on all sides,**

[35] https://en.wikipedia.org/wiki/Kernel_(image_processing)

effectively making the the height and width of the padded input $h + 2p$ and $w + 2p$ respectively, where $p$ is the size of padding. Stride ($s$) is the step size of convolution operation.

As the above figure shows, the red square on the left is a filter applied locally on the input feature map. We multiply the filter weights (of size $k \times k \times c$) with a local region of the input feature map and then sum the product to get the *output feature map*. Hence, the first two dimensions of output feature map is $[(h + 2p - k)/s + 1] \times [(w + 2p - k)/s + 1]$. Since we have $n$ filters in a convolution layer, the output feature map is of size $[(h+2p-k)/s+1] \times [(w+2p-k)/s+1] \times n$.

For more details about the convolutional layer, see Stanford's course on CNNs for visual recognition.[36]

4. *Pooling layer*

It is common to use pooling layers after convolutional layers to reduce the spatial size of feature maps. Pooling layers are also called down-sample layers, and perform an aggregation operation on the output of a convolution layer. Like the convolution layer, the pooling operation also acts locally on the feature maps. A popular kind of pooling is max-pooling, and it simply involves computing the maximum value within each feature window. This allows us to extract more salient feature maps and reduce the number of parameters of CNNs to reduce over-fitting. Pooling is typically applied independently within each channel of the input feature map.

5. *Loss layer*

For classification task, we use a softmax function to assign probability to each class given the input feature map:

$$p = \textbf{softmax}(Wx + b). \tag{3}$$

In training, we know the label given the input image, hence, we want to minimize the negative log probability of the given label:

$$l = -\log(p_j), \tag{4}$$

where $j$ is the label of the input. This is the objective function we would like optimize.

*LeNet* Having introduced the building components of CNNs, we now introduce the architecture of LeNet.

| Layer Type | Configuration |
|---|---|
| Input | size: $28 \times 28 \times 1$ |
| Convolution | $k = 5, s = 1, p = 0, n = 20$ |
| Pooling | MAX, $k = 2, s = 2, p = 0$ |
| Convolution | $k = 5, s = 1, p = 0, n = 50$ |
| Pooling | MAX, $k = 2, s = 2, p = 0$ |
| IP | $n = 500$ |
| ReLU | |
| Loss | |

[36] http://cs231n.github.io/convolutional-networks/

**The architecture of LeNet is shown in Table. 41. The name of the layer type explains itself. LeNet is composed of interleaving of convolution layers and pooling layers, followed by an inner product layer and finally a loss layer. This is the typical structure of CNNs.**