10-701 Machine Learning, Spring 2011: Homework 1 Solution

February 1, 2011

Instructions There are 3 questions on this assignment. The last question involves coding. Attach your code to the writeup. Please submit your homework as 3 **separate** sets of pages according to TAs, with your name and userid on each set.

1 Information Gain, KL-divergence and Entropy [Xi Chen, 30 points]

1. When we construct a decision tree, the next attribute to split is the one with maximum mutual information (a.k.a. information gain), which is defined in terms of entropy. In this problem, we will explore its connection to *KL-divergence*. The KL-divergence from a distribution $p(x)$ to a distribution $q(x)$ can be thought of as a distance measure from p to q:

$$KL(p||q) = -\sum_x p(x) \log_2 \frac{q(x)}{p(x)}.$$

If $p(x) = q(x)$, then $KL(p||q) = 0$. Otherwise, $KL(p||q) > 0$.[1]

We can define mutual information as the KL-divergence from the observed joint distribution of X and Y to the product of their marginals:

$$I(X, Y) \equiv KL(p(x, y)||p(x)p(y))$$

(a) Show that this definition of mutual information is equivalent to the one given in class,. That is, show that $I(X, Y) = H(X) - H(X|Y)$ and $I(X, Y) = H(Y) - H(Y|X)$ from the definition in terms of KL-divergence. From this definition, we can easily see that mutual information is symmetric, i.e. $I(X, Y) = I(Y, X)$. [10pt]

(b) According to this definition, under what conditions do we have that $I(X, Y) = 0$. [5pt]

★ **SOLUTION:**

(a)

$$\begin{aligned} KL(p||q) &= -\sum_x \sum_y p(x,y) \log_2 \left(\frac{p(x)p(y)}{p(x,y)} \right) \\ &= -\sum_x \sum_y p(x,y)(\log_2 p(x) + \log_2 p(y) - \log_2 p(x,y)) \\ &= -\sum_y p(y) \log_2 p(y) + \sum_x p(x) \sum_y p(y|x) \log_2 p(y|x) \\ &= H(Y) - H(Y|X) \end{aligned}$$

Equivalence to $H(X) - H(X|Y)$ can be shown in a similar way.

[1]For more details on KL-divergence, refer to Section 1.6 in Bishop.

(b) When X and Y are statistically independent, i.e. $p(x, y) = p(x)p(y)$, $I(X, Y) = 0$.

2. In the class, we define the entropy based on a discrete random variable X. Now consider the case that X is a continuous random variable with the probability density function $p(x)$. The entropy is defined as:

$$H(X) = -\int p(x) \ln p(x) dx$$

Assume that X follows a Gaussian distribution with the mean μ and variance σ^2, i.e.

$$p(x) = \frac{1}{\sqrt{2\pi}\sigma} \exp -\frac{(x-\mu)^2}{2\sigma^2}$$

(a) Please derive its entropy $H(X)$. [10pt]

(b) Give a careful observation for the entropy you derived and please indicate one property, which holds for the entropy for (any) discrete random variable, but does not hold here. [5pt]

★ **SOLUTION:**

$$\begin{aligned} 0H(X) &= -\int p(x) \ln p(x) dx \\ &= -\int p(x) \left(-\frac{1}{2}\ln(2\pi\sigma^2) - \frac{(x-\mu)^2}{2\sigma^2}\right) dx \\ &= \frac{1}{2}\left(\ln(2\pi\sigma^2) + \frac{1}{\sigma^2}\int p(x)(x-\mu)^2 dx\right) \\ &= \frac{1}{2}\left(\ln(2\pi\sigma^2) + 1\right) \end{aligned}$$

The last inequality is according to the variance of a standard normal distribution:

$$\sigma^2 = \text{var}(X) = \mathbb{E}\left((X-\mu)^2\right) = \int p(x)(x-\mu)^2 dx$$

Note that unlike the entropy for discrete variable which is always non-negative, when $\sigma^2 < \frac{1}{2\pi e}$, $H(x) < 0$.

2 Bayes' Rule and Point Estimation [Xi Chen, 30 points]

1. Assume the probability of a certain disease is 0.01. The probability of testing positive given that a person is infected with the disease is 0.95 and the probability of testing positive given the person is not infected with the disease is 0.05.

(a) Calculate the probability of testing positive. [5pt]

(b) Use Bayes' Rule to calculate the probability of being infected with the disease given that the test is positive. [5pt]

★ **SOLUTION:**

(a) Given the information in the problem, we have $P(D) = 0.01$, $P(T|D) = 0.95$ and $P(T|\overline{D}) = 0.05$.

$$\begin{aligned} P(T) &= P(T \wedge D) + P(T \wedge \overline{D}) \\ &= P(T|D)P(D) + P(T|\overline{D})P(\overline{D}) \\ &= 0.95 \times 0.01 + 0.05 \times 0.99 \\ &= 0.059 \end{aligned}$$

(b)

$$P(D|T) = \frac{P(T|D)P(D)}{P(T)} = \frac{0.95 \times 0.01}{0.059} \approx 0.16$$

2. The Poisson distribution is a useful discrete distribution which can be used to model the number of occurrences of something per unit time. For example, in networking, the number of packets to arrive in a given time window is often assumed to follow a poisson distribution. If X is Poisson distributed, i.e. $X \sim Poisson(\lambda)$, its probability mass function takes the following form:

$$P(X|\lambda) = \frac{\lambda^X e^{-\lambda}}{X!},$$

It can be shown that if $\mathbb{E}(X) = \lambda$. Assume now we have n i.i.d. data points from $Poisson(\lambda)$: $\mathcal{D} = \{X_1, \ldots, X_n\}$.

(For the purpose of this problem, you can only use the knowledge about the Poisson and Gamma distributions provided in this problem.)

(a) Show that the sample mean $\hat{\lambda} = \frac{1}{n}\sum_{i=1}^n X_i$ is the maximum likelihood estimate (MLE) of λ and it is unbiased ($\mathbb{E}(\hat{\lambda}) = \lambda$). [8pt]

(b) Now let's be Bayesian and put a prior distribution over λ. Assuming that λ follows a Gamma distribution with the parameters (α, β), its probability density function:

$$p(\lambda|\alpha,\beta) = \frac{\beta^\alpha}{\Gamma(\alpha)}\lambda^{\alpha-1}e^{-\beta\lambda},$$

where $\Gamma(\alpha) = (\alpha-1)!$ (here we assume α is a positive integer). Compute the posterior distribution over λ. [6pt]

(c) Derive an analytic expression for the maximum a posterior (MAP) of λ under $Gamma(\alpha, \beta)$ prior. [6pt]

★ **SOLUTION:**

(a) Write down the log-likelihood

$$\ln P(\mathcal{D}|\lambda) = \ln e^{-n\lambda}\prod_{i=1}^n \frac{\lambda^{X_i}}{X_i!} = -n\lambda + \sum_{i=1}^n \{X_i \ln\lambda - \ln(X_i!)\}.$$

The MLE $\hat{\lambda} = \arg\max_\lambda P(\mathcal{D}|\lambda) = \arg\max_\lambda \ln P(\mathcal{D}|\lambda)$, which can be obtained by setting the gradient of $\ln P(\mathcal{D}|\lambda)$ with respect to λ to 0. More specifically:

$$\frac{d}{d\lambda}\ln P(\mathcal{D}|\lambda) = -n + \frac{1}{\lambda}\sum_{i=1}^n X_i = 0 \Longrightarrow \hat{\lambda} = \frac{1}{n}\sum_{i=1}^n X_i$$

Since $X_1, \ldots, X_n$ are i.i.d. from $Poisson(\lambda)$, for any X_i, $\mathbb{E}(X_i) = \lambda$. $\hat{\lambda}$ is unbiased because:

$$\mathbb{E}(\hat{\lambda}) = \mathbb{E}\left(\frac{1}{n}\sum_{i=1}^n X_i\right) = \frac{1}{n}\sum_{i=1}^n \mathbb{E}(X_i) = \frac{1}{n}\sum_{i=1}^n \lambda = \lambda$$

(b)

$$\begin{aligned} p(\lambda|\mathcal{D}) &\propto P(\mathcal{D}|\lambda)p(\lambda) \\ &= e^{-n\lambda}\left(\prod_{i=1}^n \frac{\lambda^{X_i}}{X_i!}\right)\frac{\beta^\alpha}{\Gamma(\alpha)}\lambda^{\alpha-1}e^{-\beta\lambda} \\ &\propto \lambda^{\sum_{i=1}^n X_i + \alpha - 1}e^{-n\lambda-\beta\lambda} \end{aligned}$$

Therefore, the posterior distribution $p(\lambda|\mathcal{D}) \sim Gamma(\sum_{i=1}^n X_i + \alpha, n + \beta)$

(c) The MAP $\lambda^* = \arg\max_\lambda p(\lambda|\mathcal{D}) = \arg\max_\lambda \ln p(\lambda|\mathcal{D})$. Since $p(\lambda|\mathcal{D}) \propto \lambda^{\sum_{i=1}^n X_i+\alpha-1} e^{-n\lambda-\beta\lambda}$,

$$\ln p(\lambda|\mathcal{D}) = \left(\sum_{i=1}^n X_i + \alpha - 1\right) \ln \lambda - (n+\beta)\lambda + C,$$

where C is a constant with respect to λ. Take the gradient of $\ln p(\lambda|\mathcal{D})$ with respect to λ and set it to 0:

$$\frac{d}{d\lambda} \ln p(\lambda|\mathcal{D}) = \frac{\sum_{i=1}^n X_i + \alpha - 1}{\lambda} - (n+\beta) = 0 \Longrightarrow \lambda^* = \frac{\sum_{i=1}^n X_i + \alpha - 1}{n+\beta}$$

3 Decision Tree [Yi Zhang & Carl Doersch, 50 points]

In this question, you will write our decision tree code and perform experiments with it. You will observe and discuss the overfitting and post-pruning of the decision trees. Our data is a binary classification data set with discrete attributes, and we only require the decision tree to be able to process this kind of data. All resources are provided in the file hw1_dt.zip, including the training, validation (i.e., pruning) and testing data sets, a partially implemented decision tree codebase in C (with a few core parts removed), and necessary instructions to compile and run the codebase. **Note:** use of this codebase in **not** required. If you are not comfortable with coding in C, feel free to choose any other language to implement your own decision tree, as long as the tree can perform the experiments we require on the particular data set we provided. While this codebase is not a part of the question, we included it so that, hopefully, many of you will be able to avoid the tedious implementation details and focus instead on the interesting parts of the decision tree algorithm.

Building the decision tree: we build the decision tree as we learned in the class. Given the *training set*, we will start from a single root node with all the training examples assigned to it. Then for each node, if the assigned examples are not pure (i.e., not with the same label), we consider further spliting this node using a "best" attribute. Selecting the best attribute for a given node is the most important part for building decision trees, which is achieved by maximizing the information gain of the split, or equivalently minimizing the weighted average entropy after the splitting (i.e., the conditional entropy given the attribute, shown as $H_S(Y|A)$ in page 11 and 12 of the slides). We will stop spliting a node if: 1) the node is pure; or 2) we cannot find any attribute that leads to a positive information gain.

Checking a specific node for pruning: Pruning a node means removing the subtree beneath it, keeping the node as a leaf. As a result, all training examples assigned to the subtree are assigned to this node. Examples assigned to the node may not all have the same label, and in this case the label attached to this node is the label of the majority class (and examples of minority classes in this node are misclassified, and usually the classification accuracy on the *training set* will decrease). For performance reasons, we use a criterion different from the lecture: given a specific node and the *validation set* (i.e., the pruning set), we prune this node if the classification accuracy of the resulting new tree on the validation set improves at least $EPSILON$. $EPSILON$ is the threshold of minimal improvement for pruning. For now, we set $EPSILON$ as 0.005 (i.e., 0.5%), this default value has already been set in our codebase.

Post-pruning a decision tree: top-down and bottom-up. In order to post-prune the entire decision tree, we basically need to perform a tree traversal, and check all the nodes along the traversal. We consider **depth-first traversal**, which can be easily implemented as one function via *recursive calls.* By placing the recursive calls at different locations of the function, we can make **two choices**: 1) check the current node before invoking the recursive calls on its children; 2) invoke the recursive calls on its children before checking the current node. Note that if a node is checked and actually pruned, we will no longer travel to its children. We initially call the traversal function at the root node, and clearly the two choices we mentioned will lead to different orders of checking the tree nodes. We call the first one **the top-down approach** since it checks (and tries to prune) the parent node before recursively checking children, and we call the second one **the bottom-up approach** since it checks children before checking the parent.

Implementation and the C codebase. The C codebase provides a decision tree implementation (with a few parts removed by the TAs) with much more functionality than what we need in this question. So if you decide to use the C codebase, you only need to make changes on a few files (as detailed later) without really digging into every detail of this codebase. For a quick guide on how to compile and run the codebase, see **quick_start.txt** in the hw1_dt.zip file.

Data files. We use a noisy mushroom data set for this problem. Using this data set, we will train decision trees to classify each mushroom as poisonous or not, using 22 discrete features such as cap shape, cap color and gill size. There are three data files in hw1_dt.zip: noisy10_train.ssv, noisy10_valid.ssv, and noisy10_test.ssv. They are training set, validation set (i.e., pruning set), and testing set. The format of each file is: first three lines are data statistics (number of variables plus label, variable names, properties of each variable), and from the 4th line is the data, where each line is an example and each column is either the label (the first column) or a variable. You don't need to worry about the data format if you use our codebase.

3.1 Complete the implementation [20 points]

To fully implement the decision tree using the C codebase, there are mainly **two places** in the codebase we need to make changes: 1) **entropy.c**: the file implementing and using the entropy function to calculate information gain and choose the best splitting attribute when building the decision tree (search the comment "YOU MUST MODIFY THIS FUNCTION" in this file to find the place to add your code) ; 2) **prune-dt.c**: the file implementing the post-pruning of the tree (search the comment "YOU MUST MODIFY THIS FUNCTION" in this file to find the place to add your code).

Print the code you added in **entropy.c** and **prune-dt.c** and attach to your homework writeup. Note: if you choose to implement your decision tree without using the codebase, just print and attach your code to the writeup.

★ **SOLUTION:** See the function "Entropy" in **entropy.c** and the function "PruneDecisionTree" in **prune-dt.c** from the solution code.

■ **Common mistake 1:** Not checking the boundary condition when calculating the entropy. If a node contains no positive example or no negative examples, we should directly return 0.0 as the entropy instead of attempting to calculate it, i.e., we don't want to compute $\log_2(0)$.

■ **Common mistake 2:** Not converting *int* to *double* before calculating the quotient of two numbers. C is not as smart as Matlab and R: we need to make sure at least either numerator or denominator is a floating point number before computing their division.

3.2 Experiments with different post-pruning strategies [20 points]

We've discussed the top-down and the bottom-up approaches to travel and prune the tree, which you should already implemented in **prune-dt.c**. Run the codebase (or your own implementation) with both approaches, using the training set for building the tree, the validation set for post-pruning, and testing set to finally test the classification accuracy (again, see **quick_start.txt** for compiling and running the codebase).

Report in your homework 1) for the fully grown tree (without post-pruning): the tree size (i.e., the number of nodes and the depth of the tree), the classification accuracy on the training set, and the classification accuracy on the testing set; 2) for the post-pruned tree with top-down approach: the tree size, the classification accuracy on the training set, and the classification accuracy on the testing set; 3) for the post-pruned tree with bottom-up approach: the tree size, the classification accuracy on the training set, and the classification accuracy on the testing set. Note: all the information can be found from the output when we run the codebase. **[10 points]**

Discuss how different pruning approaches affect the size of the tree, training accuracy, and testing accuracy. Also comment on the difference between the training accuracy and the testing accuracy for each different tree (i.e., the full tree and two pruned trees) [**10 points**].

Table 1: Number of nodes as $EPSILON$ changes

	$EPSILON = 0.001$	$EPSILON = 0.005$	$EPSILON = 0.01$	$EPSILON = 0.03$
Top-Down	8	116	704	2040
Bottom-Up	636	681	1303	2040

★ **SOLUTION:** The full-grown tree has 2919 nodes and its depth is 12, with the training accuracy as 99.7% and the testing accuracy as 79.6%. The top-down pruned tree has 116 nodes and its depth is 8, with the training accuracy as 89.6% and the testing accuracy as 89.0%. The bottom-up pruned tree has 681 nodes and its depth is 11, with the training accuracy as 92.2% and the testing accuracy as 88.1%.

The top-down pruning strategy tries to prune higher-level nodes (i.e., those close to the root) before attempting to prune lower-level nodes (i.e., those close to the leaves), so it is a more aggressive pruning strategy and tends to produce smaller post-pruned trees. Since the resulting tree is small, i.e., a less complex model, the training accuracy will generally be lower than that of the full-grown tree (which "overfits" the training samples), but the testing accuracy will usually be higher than that of the full-grown tree as the less complex model generalizes to unseen testing samples better.

The bottom-up pruning strategy tries to prune children nodes before attempting to prune parents, so it is not as aggressive as the top-down strategy and thus will tend to prune less nodes and produce larger post-pruned trees (compared to the top-down pruned trees). As a result, the training accuracy of the resulting tree will generally be higher than the top-down pruned tree (as it is larger and more complex than a top-down pruned tree), but lower than that of a full-grown tree (since the pruned tree is still smaller and thus less complex than the full-grown tree). The testing accuracy of bottom-up pruned tree is usually higher than the full-grown tree (as pruning helps to prevent overfitting). It's difficult to predict which pruned tree will have lower testing accuracy than the other, because both of the following cases could happen: (1) the top-down pruned tree is over-pruned and thus is too simple to get good testing accuracy; (2) the bottom-up pruned tree is not sufficiently pruned and thus still overfit the training samples to certain degree. In our results, the bottom-up pruned tree has slightly lower testing accuracy (88.1%) than that of the top-down pruned tree (89.0%), indicating the case (2) might happen here.

The gap between the training accuracy and the testing accuracy is a good indicator of how much the model overfits the training samples. As we can see, the full grown tree with 2919 nodes has a large gap: 99.7% training accuracy and 79.6% testing accuracy, indicating serious overfitting. The bottom-up pruned tree with 681 nodes has a small gap: 92.2% training accuracy and 88.1% testing accuracy, indicating slight overfitting. The top-down pruned tree with 116 nodes has almost no gap: 89.6% training accuracy and 89.0% testing accuracy, indicating almost no overfitting.

Finally, we want to clarify that, although in this question the smallest tree (i.e., the top-down pruned tree) achieves the best testing accuracy, it is not always the case that the simplest model is the best. Over-simplified model cannot perform well.

3.3 Experiments with different threshold $EPSILON$ [10 points]

When checking each node, we require a minimal improvement of validation accuracy $EPSILON$ for pruning. So far we use the default $EPSILON = 0.005$(i.e, 0.5%). For both top-down and bottom up pruning, change $EPSILON$ and report the number of nodes in the pruned tree for $EPSILON = 0.001, 0.005, 0.01, 0.03$. Briefly explain your results (one or two sentences will suffice).

NOTE: in the codebase, $EPSILON$ is defined in **auxi.h**: search "#define EPSILON 0.005" to find the location of $EPSILON$.

★ **SOLUTION:** See the Table 1 for detailed results. Generally speaking, larger $EPSILON$ will require more improvement of validation accuracy to approve a pruning, so increasing $EPSILON$ will tend to prune less nodes and produce larger post-pruned trees.