Exam-Style Problems

1. The customary warnings apply for this section of the assignment:

   • It is in your interest to formulate your answers by hand and without peeking at your notes. You may then want to verify some of them in MATLAB and against your notes.

   • Your answers to exam-style problems are graded just like the rest, but count for a small fraction of the overall homework score (1 to 3 points per question).

(a) Summarize the comparison of linear learners and decision trees in terms of bias that Pedro Domingo made in his article *A few useful things to know about machine learning*. Specifically, briefly state which has lower bias and why.

(b) In his article *A few useful things to know about machine learning*, Pedro Domingo shows a bare-bones decision tree learner. What impurity measure does he use? Just give the name, you need not explain.

(c) Give your best estimate of the Recall-Precision Equal-Error Rate (RPEER) for the diagram below.
You are training a classification tree for a binary labeling problem, using the empirical error as a measure of impurity. Features are real numbers (one-dimensional features), and possible labels are 0 and 1. The training samples that reach node \( n \) in the tree are as follows:

\[ S = \{ (0, 1), (1, 0), (2, 1), (3, 1), (4, 1), (5, 0), (6, 0), (7, 0), (8, 1), (9, 1) \} . \]

These samples are also drawn in the diagram below for your convenience.

1. What is the impurity \( i(S) \) of \( S \)?
2. Give the optimal splitting threshold \( \tau \) for \( S \).
3. Training samples that are below the splitting threshold are placed in set \( L \) and the others in set \( R \). What is the combined impurity

\[ i_c = \frac{|L|}{|S|} i(L) + \frac{|R|}{|S|} i(R) \]

after the split? Show your calculations.
4. Is it possible to improve purity in \( L \) and \( R \) by further splits? If you are sure, just answer yes or no.

(e) When is a training patch labeled as positive in a Hough forest?

(f) You want to find a local minimum of the function

\[ y = f(x) = \frac{1}{4} (x_1^2 - 4x_1 + 5)(x_2^2 + 1) \]

using numerical optimization starting at \( x_0 = [0, 0]^T \). Whenever numerical values are requested below, it is OK to leave the values as simple fractions of mutually prime integers (example: \( 9/16 \)).

1. Write an expression for the gradient \( g(x) = \nabla f(x) \) of \( f \) at \( x \). Also give \( g(x_0) \).
2. Write an expression for the Hessian

\[ H(x) = \frac{\partial^2 f}{\partial x \partial x^T}(x) \]

of \( f \) at \( x \). Also give \( H(x_0) \).
3. Write the numerical values for the point \( x_1 \) and value \( y_1 \) reached by one step of steepest descent from \( x_0 \), without using momentum and with a fixed step size \( \alpha = 0.5 \).
4. Write the numerical values for the point \( x_1 \) and value \( y_1 \) reached by one step of steepest descent with exact line search from \( x_0 \). [One step includes a full line search, and “exact” means that line search reaches the exact minimum along the search direction.]

### Decision Trees

2. Problems in this section ask you to implement decision trees from scratch by writing inefficient but simple code that follows the class notes. Writing your own code ensures that you understand the concepts. You are not allowed to use MATLAB’s own implementation of decision trees.

Some functions will have more arguments than the corresponding functions in the class notes, so that you can try out various parameter selections if you like. MATLAB does not allow question marks in function names, so your \( \text{split?} \) function will be called \( \text{OkToSplit} \) (admittedly not as nice). Your code should check that impurity is positive in \( \text{OkToSplit} \). What the functions below are supposed to do is explained in the class notes.

The file \( \text{data.mat} \) provided with this assignment contains three data structures \( T, V, \) and \( S \) with training, validation, and test data, respectively. Let us call data structures of this type set structures. Each set structure has the following fields:
**X:** An \( N \times 2 \) array of \( N \) training feature vectors \( x_n \). Each vector is two-dimensional, so we can draw features in the problems that follow. Consistently with conventions used in some of the MATLAB toolboxes, feature vectors are the rows of field \( X \), rather than its columns.

**y:** A column vector with \( N \) labels \( y_n \).

**labelMap:** The set of possible label values. In this problem, all labels are consecutive positive integers starting with 1. For instance, a problem with 3 classes has labelMap equal to \([1\ 2\ 3]\).

(a) Write \texttt{trainTree} in recursive form following the class notes and without using functions from the MATLAB Statistics toolbox. Your function should also define functions \texttt{OkToSplit} (replaces \texttt{split?}), \texttt{findSplit}, \texttt{distribution}, \texttt{impurity}, as defined in the notes. Your top function should have header

```matlab
function tau = trainTree(S, depth, random, dMax, sMin)
```

The meaning of the arguments is as follows (the term “optional” below means that the caller need not specify the argument, but your function still needs to handle the corresponding parameter properly):

- \( S \) is a training set in the format of a set structure (described earlier).
- \( depth \) is the depth in the tree of the node currently being built (zero for the root).
- \( random \) (optional) is \texttt{true} if the dimension on which to split at every node is to be chosen at random. Default is \texttt{false}.
- \( dMax \) (optional) is the maximum depth of the tree. Default is \texttt{Inf}.
- \( sMin \) (optional) is the minimum number of training samples in a leaf. Default is 1.

A skeleton for \texttt{trainTree} that also shows how to handle optional arguments is provided with this assignment. If the function calls itself internally, it needs to specify all arguments, so they are passed properly to other invocations. All helper functions (including any of your own) should be included in the same file (see skeleton). It is OK to use \texttt{histcounts} if you like, but, if you do, be careful with its arguments. Keep it simple! In particular, the tree data structure should be as described in the notes.

**Programming Notes:**

- While the pseudo-code in the class notes is correct, two issues of efficiency arise in a practical implementation:
  - In some circumstances, two or more splits may have the same quality (same value of \( \Delta \)). The output tree is correct regardless of how these ties are broken, but some ways to break ties lead to smaller trees than other ways.
  - Finite-precision arithmetic leads to small errors in the results of numerical calculations, and what is ideally zero may turn out to be slightly positive or slightly negative.

These two issues interact with each other in subtle ways, because what is ideally a tie (first issue) may end up not being recognized as such because of numerical inaccuracies (second issue).

Here is a **recipe** for addressing these issues:

- If \( i \) is the name of the output argument in the body of the function \texttt{impurity}, then the last instruction of \texttt{impurity} should be
  ```matlab
  i = clip(i);
  ```
  The function \texttt{clip} is provided, and clips anything that is very close to zero to exact zero.

- Right after computing the value of \( \Delta \) in \texttt{findSplit}, insert the instruction
  ```matlab
  Delta = fix(Delta, S, L, R);
  ```
  This syntax assumes that the names of the variables that represent \( \Delta, S, L, R \) are \texttt{Delta, S, L, R}. If your code uses different names, change this instruction accordingly.

This recipe works for the small datasets in this assignment. For much larger datasets, the solution would be more complex. **You need not understand why this recipe works. If you are curious, an explanation is given in the Appendix at the end of this assignment.**

- If the training set contains two samples \((x_1, y_1)\) and \((x_2, y_2)\) with \( x_1 = x_2 \) but \( y_1 \neq y_2 \), then it is impossible to split the set any further. Logically, you would have to check for this occurrence in \texttt{OkToSplit}, but it is more efficient to keep this function simple. Instead, allow for the possibility that \texttt{findSplit} returns nothing (for instance, let \texttt{findSplit} set \( L \) to the empty array [] to communicate this to the caller), and stop splitting in that case.

- The function \texttt{findSplit} takes a set structure \( S \) as input, and its first two output arguments are set structures as well. When you make these two set structures, make sure they have all the fields that a set structure needs.

**What To Hand In:** Show your code, as well as the two figures resulting from the following calls:
rng(3);
load data
tau1 = trainTree(T, 0, false);
showPartition(tau1, T, 1)
tau2 = trainTree(T, 0, true);
showPartition(tau2, T, 2)

The function showPartition.m is provided with this assignment. The last argument to showPartition is a MATLAB figure number. The call rng(3) ensures that your pseudo-random numbers are the same as ours, and makes our grading (and your debugging!) easier.

(b) Write a function with header

    function y = treeClassify(x, tau)

following the class notes. The function returns the label y that classification tree tau assigns to feature x.

HINT: To check if the current node is a leaf you can simply say

    if isempty(tau.d)

WHAT TO HAND IN: Show your code.

(c) Write a function with header

    function e = err(f, S)

that takes a classifier f and a set S of samples and computes the empirical error rate of f on S. The empirical error rate (or misclassification rate) is a number between 0 and 1 (inclusive).

PROGRAMMING NOTES:

• The argument f is a handle to a MATLAB function. For instance, to compute the error rate of decision tree tau on set T you would write

        err(@(x) treeClassify(x, tau), T)

• It is OK to loop over training samples for this question.

WHAT TO HAND IN: Show your code for err and report the training error rates (error rates on T) for the trees tau1 and tau2 you developed earlier.

(d) Explain briefly why the empirical error rates you found in your previous answer make sense.

(e) Report the test error rates for the trees tau1 and tau2. These error rates are computed on the provided test set S.

(f) Consider trees trained as follows:

    tau = trainTree(T, 0, false, d);

These trees use optimal split dimensions (because the third argument is set to false) and have maximum depth d. Use the provided validation set V to plot the validation error rate for trees tau trained in this way as the maximum depth d is increased from 1 to 20 in increments of 1. Make sure you optimize (an estimate of) the generalization error, not the training error! That is, train on T, validate on V.

WHAT TO HAND IN: Show the code you used to compute the values in your plot. Also show your plot and report the depth(s) for which a minimum value is achieved. Finally, state what that value is.

(g) Report the test error rates (on S, provided) for the best tree you found in the previous problem. In case of ties, use the shallower tree.
Random Forests

3. Problems in this section use your implementation of decision trees from the previous section, as well as the data for it. If you were not able to make your implementation work, use the MATLAB decision trees for problems in this section (not for those in the previous one!), but implement your own forest. If you use the MATLAB version, state clearly that you did so. It is up to you to figure out how the MATLAB software works.

(a) Write a function with header

```matlab
function [phi, used] = trainForest(T, M)
```

that trains a forest phi with M trees on training set T as described in the class notes.

Differently from the class notes, your `trainForest` also returns an \( N \times M \) array `used` of logical values, where \( N \) is the number of samples in the training set T. Entry \((n,m)\) of `used` is true if and only if training sample \( n \) was used to train tree \( m \).

This array will be used later on to compute the out-of-bag error rate for phi.

Train your forest on the provided set T with \( M = 10 \) trees.

**PROGRAMMING NOTES:**

- A vector of \( N \) pseudo-random integers drawn with replacement from \([1, N]\) can be computed as follows:
  
  ```matlab
  r = round(1 + (N-1) * rand(N, 1));
  ```

- The change marked in red in `findSplitR` in the class notes can be implemented simply by setting the third argument of `trainTree` to `true`.

- Make sure that your trees are complete (that is, set `dMax` to `Inf`).

- Use the default value (1) for `sMin`, so there is no limit on set size for splitting.

**WHAT TO HAND IN:** Show your code and state how long training took on your computer (use `tic` and `toc` to find out).

(b) Write a function `forestClassify` as described in the class notes.

**PROGRAMMING NOTE:** The function `forestClassify` needs to know the number \( K \) of classes in order to allocate the vote vector \( v \). Your function should figure out \( K \) from the forest phi. If you are unable to do so, pass \( K \) as an additional parameter to `forestClassify`, for a small grade penalty.

**WHAT TO HAND IN:** Show your code and report the training error rate (on the set \( T \)) and the test error rate (on \( S \)) of the classifier phi you trained in the previous problem.

(c) Write a function with header

```matlab
function e = oobErr(phi, T, used)
```

that computes the out-of-bag error rate of forest phi.

**PROGRAMMING NOTES:** Just as for `err`, it is OK to loop over samples. Use as much of your previous code as possible.

**WHAT TO HAND IN:** Show your code and report the out-of-bag error rate for the forest phi you trained earlier.
Function Optimization

4. The Matlab function `linesearch.m` provided with this assignment has the following header:

```matlab
function x = linesearch(Grad, x0, p)
```

This function takes three arguments:

- The handle `Grad` of a function with format
  ```matlab
  function g = Grad(x)
  ```
  that returns the gradient of some function `f(x)` at `x` (this function is specific to some known `f`).
- A starting point `x0` for line search.
- A search direction `p` that need not be normalized to unit length.

The idea is that if you want to minimize a function `f(x)` with some method that requires knowing the gradient `g(x)` of `f(x)`, then you just provide a routine to compute `g(x)`. The function `linesearch` is different from what we saw in class: instead of finding a minimum of `f(x)`, it finds a zero of `g(x)^T p`, after checking that `g(x_0)^T p` is indeed negative. This is why the version of line search given above does not take `f` itself as argument. In fact, `f(x)` itself is never needed for this homework. All you need is its gradient.

(a) Write a Matlab function that computes a local minimum of a function `f(x)` by steepest descent. Your function should have the following header:

```matlab
function [x, allx] = steepest(Grad, x, maxits, deltax)
```

where

- `Grad` is the name of a function that computes the gradient of `f(x)`, as discussed above.
- The input argument `x` is a starting point.
- `maxits` is the maximum number of iterations for steepest descent.
- `deltax` is a termination bound: your function will stop either when the gradient of the function at `x_k` is zero, or when the magnitude
  \[ \delta = \|x_k - x_{k-1}\| \]
  of the last step taken becomes shorter than `deltax`, or after `maxits` iterations, whichever comes first.
- The output argument `x` contains the solution point.
- The output `allx` contains all the steps `x_k` that `steepest` took towards the minimum. This is useful in order to see what path your minimization function follows.

**WHAT TO HAND IN:** Show your code.

(b) The function

\[ f(x) = 100(x_2 - x_1^2)^2 + (1 - x_1)^2 \]

is known as Rosenbrock's function. It has a unique minimum at `x^* = (1, 1)^T`. Write an expression for the gradient

\[ g(x) = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \end{bmatrix} \]

of this function.

You may want to verify that `g([1, 1]^T) = [0, 0]^T`, but you need not show your calculations to this end.

(c) Write a Matlab function `g = rosengrad(x)` that computes the gradient `g` of Rosenbrock's function at `x`.

**WHAT TO HAND IN:** Show your code.
(d) Call steepest with rosengrad as follows:

```matlab
x0 = [-1.2 1]';
deltax = 1.0e-4;
[x, allx] = steepest(@rosengrad, x0, 100, deltax);
drawRosenbrock(allx, fig);
```

where `fig` is a figure number. This will plot the first 100 steps taken by your algorithm when it is starts at \( x_0 = (-1.2, 1.0)^T \).

**What To Hand In:** Show the resulting plot.

(e) Does your algorithm converge to the correct solution \( x^* \) within 100 steps? If not, what is `deltax` when it gives up?

**Hint:** `allx` contains all the steps taken.

(f) How far is the solution \( x \) from \( x^* \) upon termination? Measure this distance in Euclidean norm.

(g) Arm yourself with patience (that is, set `maxits` to `Inf` and ignore the warning). How many steps does it take for your steepest descent code to converge within the value of \( \delta_x = 10^{-4} \)? You can find out the number of steps as `size(allx, 2)` once the function is done.

**What To Hand In:** Show the new plot of the optimization path, state how many steps the algorithm takes, and how far the solution \( x \) is from \( x^* \) upon termination.

(h) Now push your luck, and keep going until the step size becomes so small that numerical inaccuracies start to predominate:

```matlab
[x, allx] = steepest(@rosengrad, x0, Inf, eps);
```

Repeat the analysis you did in the previous problem (including the plot).
Appendix: Clipping Zeros and Breaking Ties

This appendix is optional reading, and explains the rationale for the functions \texttt{clip} and \texttt{fix} provided for the problems on decision trees.

Clipping Zeros

Impurity values that are very close to zero are likely to be true zeros perturbed by numerical errors. These errors are caused by the use of finite precision arithmetic. For instance, if you write

\[
\frac{193}{300} - \left(\frac{296}{300} \times \frac{193}{296}\right)
\]

in MATLAB you get exactly zero, as appropriate. However, the mathematically equivalent expression

\[
\frac{193}{300} - \left(\frac{296 \times (193/296)}{300}\right)
\]

yields about \(1.11 \times 10^{-16}\) (try it!).

These errors seem harmless for decision trees: If one split of a decision tree is better than another by \(10^{-16}\), they have essentially the same quality, so picking the “wrong” split has no serious consequence. However, as shown in the next section, in some situations breaking ties appropriately between equally good splits leads to much smaller trees. To take advantage of this insight, one needs to recognize ties in the first place, and numerical errors prevent us from doing so reliably.

These errors could be avoided by using exact rational arithmetic: For instance, represent \(\frac{193}{300}\) by the pair \((193, 300)\) rather than \(0.64333...\), and then carry out all the math with integers only. Solutions like this may be the only option for “industrial grade” software that makes no assumptions about the size of the data sets involved.

A simpler solution is possible when the datasets are small. First, the ties of interest arise when \(\Delta\) is zero, so all we need to do is to correct these values when they are very small. Second, it is not too hard to show that the smallest (exact) nonzero value of \(\Delta\) that can arise when splitting a set with at most \(N\) elements cannot be arbitrarily close to zero. Instead, this value is bounded away from zero by a positive number \(\delta\) that is of the order of magnitude of \(1/N\). In other words, the exact value of \(\Delta\) is either zero or greater than \(\delta\), and therefore any \(\Delta\) that is found (numerically) to be smaller than \(\delta\) can be safely clipped to zero.

It is not immediate to compute a precise estimate of \(\delta\). However, for datasets that only have a few thousand (or even a few hundreds of thousands of) entries, \(\delta\) is definitely less than \(10^{-6}\), and we use the safe value

\[
\delta = \sqrt{\epsilon} \approx 1.49 \times 10^{-8} \quad \text{where} \quad \epsilon \approx 2.22 \times 10^{-16}.
\]

The constant \(\epsilon\) is the smallest nonzero number greater than 1 that a 64-bit floating point number can represent. The value of \(\epsilon\) is a good estimate of the rounding error, and its square root \(\delta\) is much bigger than that (so we can still do useful arithmetic with numbers of that magnitude) and yet still much smaller than \(1/N\) for our experiments.

The function \texttt{clip} provided with this assignment implements this method, and coerces to zero any value that is within \(\delta\) from zero.

Breaking Ties

Consider the following set of features, where a ‘+’ denotes a positive label and ‘−’ a negative one. The feature vector \(x\) is the pair of coordinates of each point on the plane.

\[
\begin{array}{c}
+ \\
- \\
\end{array}
\]

No single vertical or horizontal line can separate the positive sample from the two negative samples, and none of the four possible separation lines changes impurity. For all of them we have

\[
\Delta = i(S) - \frac{|L|}{|S|} i(L) - \frac{|R|}{|S|} i(R) = \left(1 - \frac{2}{3}\right) - \frac{1}{3} \cdot 0 - \frac{2}{3} \cdot \left(1 - \frac{1}{2}\right) = 0
\]

(possibly by renaming \(L\) and \(R\)). This zero result is not a coincidence, and the small example above can be generalized as follows (see also the figure below).
Theorem A.1. Suppose that a set \( S \) is split into two sets \( L \) and \( R \) where, say, \( L \) is pure. Further assume that the only label in \( L \) is the majority label \( y^* \) in \( S \), and that the number \( R_y \) of samples with label \( y^* \) in \( R \) is greater than or equal to \(|R_y|\) for \( y \neq y^* \). In other words, by removing \( L \) from \( S \) to create \( R \), the majority label in \( R \) is still \( y^* \). Then, \( \Delta = 0 \).

Each rectangle with solid boundaries represents a set of samples with the same label, and rectangle sizes represent set sizes. Label \( y^* \) is the majority label in \( S \). Once \( S \) is split into \( L \) and \( R \) (dashed line), there are still enough \( y^* \) labels in \( R \) to make \( y^* \) the majority label in \( R \) as well. Of course, a similar result holds with the roles of \( L \) and \( R \) reversed.

Proof. If \( S_y \) is the subset of samples of \( S \) with label \( y \), then the impurity of \( S \) is

\[
i(S) = 1 - \max_y \frac{|S_y|}{|S|} = 1 - \frac{|S_{y^*}|}{|S|} = \frac{|S| - |S_{y^*}|}{|S|} \quad \text{where} \quad y^* = \arg \max_y |S_y|.
\]

Since \( y^* \) is also the majority label in \( R \), the impurity of \( R \) is

\[
i(R) = 1 - \max_y \frac{|R_y|}{|R|} = 1 - \frac{|R_{y^*}|}{|R|} = \frac{|R| - |R_{y^*}|}{|R|}.
\]

However,

\[|R_{y^*}| = |S_{y^*}| - |L| \quad \text{and} \quad |R| = |S| - |L|\]

so that

\[
i(R) = \frac{|R| - |R_{y^*}|}{|R|} = \frac{|S| - |S_{y^*}|}{|R|}.
\]

Since \( i(L) = 0 \), we have

\[
\Delta = i(S) - \frac{|R|}{|S|} i(R) = \frac{|S| - |S_{y^*}|}{|S|} - \frac{|R|}{|S|} \frac{|S| - |S_{y^*}|}{|R|} = 0
\]

as promised.

Assume then that at some point in the execution of \texttt{trainTree} all possible best splits in the current dimension are of the type above. (We saw at least in a small example that this can happen, and it happens quite often in practice). Then the theorem above says that all these splits are equally indifferent. We still do want to split, because we want to reach complete purity eventually. So, which split do we choose? It is obviously most efficient to make \( L \) as large as possible, because otherwise we end up slicing off majority labels from \( S \) in several splits, where a single split would do. If we keep handling ties optimally in this way, the trees we build are significantly smaller than otherwise.

The function \texttt{fix} implements this strategy by modifying the value of \( \Delta \) so that larger pure sets are favored over smaller ones if a tie occurs. To this end, the function first maps the raw value of \( \Delta \) (which is between 0 and 1/2 if impurity is defined as the error rate) to the interval \([\delta, 1/2]\), where \( \delta \) was defined earlier. This creates a modest amount of room (between 0 and \( \delta \)) to reward larger sets should a tie occur, without interfering with the values of \( \Delta \) for other cases. The function then checks that the conditions of the theorem hold, not by looking at impurities (because these might be tainted by numerical errors), but examining the (integer!) numbers of labels in each set. If the condition holds, the value of \( \Delta \) is changed from zero to the tiny value \( 0.9 \delta |L|/|S| \), which is strictly below \( \delta \). This \( \Delta \) favors larger sets \( L \). Of course, the code is written so that the analogous action is taken when the roles of \( L \) and \( R \) are reversed.