\section{\ell_1-Regularized Linear Regression: LASSO}

The \ell_1-norm is one of the popular regularizers used to enhance the robustness of regression models. Regression with an \ell_1-penalty is referred to as the LASSO regression. It promotes sparsity in the resulting solution. In this problem, we will explore the optimization of the LASSO in a simplified setting.

Assume the training data points are denoted as the rows of a \( n \times d \) matrix \( X \) and their corresponding output value as an \( n \times 1 \) vector \( y \). The generic parameter vector and its optimal value (relative to the LASSO cost function) are represented by \( d \times 1 \) vectors \( w \) and \( \hat{w} \), respectively. For the sake of simplicity, assume columns of data have been standardized to have mean 0 and variance 1, and are also uncorrelated (i.e. \( X^T X = nI \)). (We center the data mean to zero, so that the penalty treats all features similarly. We assume uncorrelated features as a simplified assumption in order to reason about LASSO in this question. In general, this is a major simplification since the power of regularizers is that they often enable us to make reliable inferences even when we do not have as many samples as we have raw features.)

For LASSO regression, the optimal parameter vector is given by:

\[
\hat{w} = \arg \min_w \{ J_\lambda(w) = \frac{1}{2} \| y - Xw \|_2^2 + \lambda \| w \|_1 \},
\]

where \( \lambda > 0 \).

(a) \textbf{Show that for data with uncorrelated features, one can learn the parameter} \( w_i \) \textbf{corresponding to each} \( i \)-th feature independently from the other features, one at a time, and \textbf{get a solution which is equivalent to having learned them all jointly as we normally do.}

\textbf{Hint:} To show this, write \( J_\lambda(w) \) in the following form for appropriate functions \( g \) and \( f \):

\[
J_\lambda(w) = g(y) + \sum_{i=1}^{d} f(X_i, y, w_i, \lambda)
\]

where \( X_i \) is the \( i \)-th column of \( X \). By having no interaction terms that link the different \( w_i \) variables, you know that the joint optimization can be decomposed into individual optimizations.
Solution: Considering $X_i$ as the $i$-th column of $X$:
\[
\frac{1}{2} \| y - Xw \|_2^2 + \lambda \| w \|_1 = \frac{1}{2} y^\top y + \sum_{i=1}^d \left\{ -y^\top X_i w_i + \frac{n}{2} w_i^2 + \lambda |w_i| \right\}
\]

Notice that we achieved the form:
\[
J_\lambda(w) = g(y) + \sum_{i=1}^d f(X_i, y, w_i, \lambda)
\]

Using this with the hint we get the solution.

(b) Assume that $\hat{w}_i > 0$. What is the value of $\hat{w}_i$ in this case?

Solution: If $\hat{w}_i > 0$, then we want to minimize
\[
- y^\top X_i w_i + \frac{n}{2} w_i^2 + \lambda |w_i| = - y^\top X_i w_i + \frac{n}{2} w_i^2 + \lambda w_i
\]

Since the function above is convex, taking the derivative and equating to zero will give us the minimum. So, we get:
\[
\hat{w}_i = \frac{1}{n} (y^\top X_i - \lambda)
\]

This is only valid if $\frac{1}{n} (y^\top X_i - \lambda)$ is indeed greater than zero.

(c) Assume that $\hat{w}_i < 0$. What is the value of $\hat{w}_i$ in this case?

Solution: If $\hat{w}_i < 0$, then we want to minimize
\[
- y^\top X_i w_i + \frac{n}{2} w_i^2 + \lambda |w_i| = - y^\top X_i w_i + \frac{n}{2} w_i^2 - \lambda w_i
\]

Take the derivative and equate to zero, we have:
\[
\hat{w}_i = \frac{1}{n} (y^\top X_i + \lambda)
\]

This is only valid if $\frac{1}{n} (y^\top X_i - \lambda)$ is indeed less than zero.

(d) From the previous two parts, what is the condition for $\hat{w}_i$ to be zero?

Solution: At $w_i = 0$ the derivative is not defined, it is a critical point. The optimum will be there if no zeros exist for the derivative. From the previous parts, we know $\hat{w}_i = 0$ if none of the above conditions hold, that is;
\[
y^\top X_i + \lambda \geq 0, \quad y^\top X_i - \lambda \leq 0
\]

Combining them, we get
\[-\lambda \leq y^\top X_i \leq \lambda\]
(e) Now consider the ridge regression problem where the regularization term is replaced by $\lambda\|w\|^2_2$ where the optimal parameter vector is now given by:

$$\hat{w} = \arg\min_w \{ J_\lambda(w) = \frac{1}{2}\|y - Xw\|^2_2 + \lambda\|w\|^2_2 \},$$

where $\lambda > 0$.

**What is the condition for $\hat{w}_i = 0$? How does it differ from the condition you obtained in the previous part?** Can you see why the $\ell_1$ norm promotes sparsity?

**Solution:** If the LASSO is replaced by $\lambda\|w\|^2_2$, the optimization problem regarding $w_i$ is given by:

$$-y^\top X_i w_i + \frac{n}{2}w_i^2 + \lambda w_i^2$$

take the derivative and equate to zero:

$$\hat{w}_i = \frac{y^\top X_i}{n + 2\lambda}$$

It is equal to zero if $y^\top X_i = 0$ exactly or $\lambda$ goes to infinity. A finite $\lambda$, if $y$ is noisy this will never happen. In contrast, $\hat{w}_i = 0$ when $|y^\top X_i| < \lambda$ in K LASSO regression. This is why the $l_1$-norm regularization encourages sparsity.

(f) Assume that we have a sparse image vectorized in the vector $w$ (so $w$ is a sparse vector). We have a Gaussian matrix $n \times d$ matrix $X$ and an $n \times 1$ noise vector $z$ where $n > 1$. Our measurements take the form $y = Xw + z$. We want to extract the original image $w$ given matrix $X$ knowing that this image is sparse. The fact that $w$ is sparse suggests using $\ell_1$ regularization. Use the provided iPython notebook and apply $\ell_1$ regularization.

(This might remind you of EE16A. That’s as intended.)

**Change the hyperparameter $\lambda$ to extract the best looking image and report it.**

**Solution:** Please refer to the provided code.
3 Variance of Sparse Linear Models Obtained by Thresholding

In this question, we will analyze the variance of learning sparse linear models. In particular, we will analyze two simple procedures (computing the OLS solution and then just keeping the biggest entries or just keeping entries bigger than a threshold) that perform feature selection in linear models, and show quantitatively that feature selection lowers the variance of linear models. This should make sense to you at an intuitive level: enforcing sparsity is equivalent to deliberately constraining model complexity; think about where this puts you on the bias variance trade-off.

However, note that there is a subtle difference between feature selection before training and after training. If we use fewer features to begin with, our results so far imply that we will have low variance because of smaller model complexity. What we learn from working through this problem is that selecting features adaptively (that is based on the training data itself) does not hurt either, if done properly. In other words, although there is a philosophical difference between doing feature selection before or after using the data, post training feature selection still leads to variance reduction under certain assumptions.

First, some setup. Data from a sparse linear model is generated using

\[ y = Xw^* + z, \]

where \( y \in \mathbb{R}^n \) denotes a vector of responses, \( X \in \mathbb{R}^{n \times d} \) is our data matrix, \( w^* \in \mathbb{R}^d \) is an unknown, \( s \)-sparse vector of true parameters (with at most \( s \) non-zero entries), and \( z \sim N(0, \sigma^2 I_n) \) is an \( n \)-dimensional vector of i.i.d. Gaussian noise of variances \( \sigma^2 \).

The solution to first three parts of the problem can be filled out in the provided iPython notebook. Parts (d)-(j) must have a separate, written solution. All logarithms are to the base \( e \).

(a) Let us first do some numerical exploration. **In the provided iPython notebook, you will find code to generate and plot the behavior of the ordinary least squares algorithm.**

**Solution:** See solution notebook.

The error is defined as \( \|w^* - \hat{w}\|_2^2 \). \( n \) is the number of examples, \( d \) is the number of feature dimensions and \( s \) is the number of non-zero entries in the weights \( w \). The 3 figures below plots the error versus \( n, d, s \) respectively.

Above, we plotted the normalized prediction error over:

1. \( n \) (in the range \((100, 2000))\), with \( d = 100 \) and \( s = 5 \).
2. \( d \) (in the range \((10, 1000))\), with \( n = 1000 \) and \( s = 5 \).
3. \( s \) (in the range \((5, 50))\), with \( d = 100 \) and \( s = 5 \).

Clearly, these plots are expected. The first has slope \(-1\), showing that the error has an inverse dependence on \( n \). The second is linear, with slope roughly \( 1/1000 = 1/n \), as expected. The third shows that the least squares error does not depend at all on the sparsity \( s \), as expected.

See Figure 1, 2 and 3 for details.
Figure 1: OLS log error vs log n. Errors in all figures are defined as $\|w^* - \hat{w}\|_2^2$. In this case $d = 100$ and $s = 5$. It has slope -1, showing that the error has an inverse dependence on $n$.

Figure 2: OLS error vs $d$. In this case, $n = 1000$ and $s = 5$. It is linear, with slope roughly $1/1000 = 1/n$.

Figure 3: OLS error vs $s$. In this case, $d = 100$ and $s = 5$. The least squares error does not depend at all on the sparsity $s$, as expected.

(b) In this problem, we will analyze two estimators that explicitly take into account the fact that $w^*$ is sparse, and consequently attain lower error than the vanilla least-squares estimate.
Let us define two operators. Given a vector \( v \in \mathbb{R}^d \), the operation \( \tau_k(v) \) zeroes out all but the top \( k \) entries of \( v \) measured in absolute value. The operator \( T_\lambda(v) \), on the other hand, zeros out all entries that are less than \( \lambda \) in absolute value.

Recall that the least squares estimate was given by \( \hat{w}_{\text{LS}} = X^\dagger y = X^\top y \), where \( X^\dagger \) is the pseudo-inverse of \( X \) (it is equal to the transpose since \( X \) has orthonormal columns). We now define

\[
\hat{w}_{\text{top}}(s) = \tau_s(\hat{w}_{\text{LS}}) \\
\hat{w}_T(\lambda) = T_\lambda(\hat{w}_{\text{LS}}),
\]

which are the two sparsity-inducing estimators that we will consider.

Now implement the two estimators described above and plot their performance as a function of \( n, d \) and \( s \).

**Solution:** See solution notebook.

Same as above, the error is defined as \( ||w^* - \hat{w}||^2 \). \( n \) is the number of examples, \( d \) is the number of feature dimensions and \( s \) is the number of non-zero entries in the weights \( w \). The 3 figures below plots the error versus \( n, d, s \) respectively. Here for each of the plots, we plot 3 methods: least square, the top \( s \) estimator and the threshold estimator.

We see here that the sparsity seeking estimators are way better than the least squares estimator as long as the sparsity is small enough. This advantage is most stark in the case of \( d \)-dependence, where we see that while the least squares estimator has error depending linearly on \( d \), the sparsity-seeking estimators have way lower dependence (logarithmic, as predicted by our theory).

See Figure 4, 5 and 6 for details. We also included the oracle performance, where we are given which entries of the weight \( w \) are non-zero, denoted as \( \text{LS}_{\text{oracle}} \) in the figures. Note that you don’t need to include the oracle in your solution.

![Figure 4: Three estimators’ log error vs log n. LS is the ordinary least square estimator, Tops is the \( \hat{w}_{\text{top}}(s) \) estimator and Thresh is the \( \hat{w}_T(\lambda) \) estimator. In this case \( d=100 \) and \( s=5 \). The sparsity aware estimators are much better than OLS. The slop of Tops and Thresh are the same as that of LS.](image)
Figure 5: Three estimators’ error vs $d$. In this case, $n = 1000$, $s = 500$. We see that while the least squares estimator has error depending linearly on $d$, the sparsity-seeking estimators have way lower dependence (logarithmic, as predicted by our theory).

Figure 6: Three estimators’ error vs $s$. In this case, $n = 1000$, $d = 100$. LS does not have any dependence on the underlying model sparsity, while the sparsity aware estimators’ error increase linearly with the sparsity $s$.

(c) **Now generate data from a non-sparse linear model, and numerically compute the estimators for this data. Explain the behavior you are seeing in these plots in terms of the bias-variance trade-off.**

**Solution:** See solution notebook.

We see here that the top(s) estimator does way worse than expected when it searches for a model with sparsity 5, even when the true sparsity is 25. Clearly, this implies that the bias of the estimator is high, since it is searching within a small class of models that does not contain the truth. When the sparsity being searched over is large enough, the error now drops to something reasonable, since we are now measuring variance as before.

See Figure 7, 8 and 9 for details.
Figure 7: On a non-sparse dataset, three estimators’ log error vs log n. In this case $d = 100$, $s = 5$. Since we assume the wrong number of sparsity entries, Tops performs the worst due to high bias.

Figure 8: On a non-sparse dataset, three estimators’ error vs $d$. In this case, $n = 1000$, $s = 5$. The Thres estimator performs the best, since it has the correct assumption of the noise level. The performance of LS is independent of the true $s$. Tops still perform worst due to high bias.

(d) In the rest of the problem, we will theoretically analyze the variance of the top-k procedure for sparse estimation, and try to explain the curves you saw in the numerical explorations above. We will need to use a handy tool, which is a bound on the maximum of $d$ Gaussian random variables.

**Show that given $d$ Gaussians $\{Z_i\}_{i=1}^d$ (not necessarily independent) with mean 0 and variance $\sigma^2$, we have**

$$
\Pr \left\{ \max_{i \in \{1, 2, \ldots, d\}} |Z_i| \geq 2\sigma \sqrt{\log d} \right\} \leq \frac{1}{d}.
$$

**Hint 1:** You may use without proof the fact that for a Gaussian random variable $Z \sim N(0, \sigma^2)$ and scalar $t > 0$, we have $\Pr\{|Z| \geq t\} \leq e^{-\frac{t^2}{2\sigma^2}}$.

**Hint 2:** For the maximum to be large, one of the Gaussians must be large. Now use the union bound.
Figure 9: On a non-sparse dataset, three estimators’ error vs s. In this case, \( n = 1000, d = 100 \). We see here that the top(s) estimator does way worse than expected when it searches for a model with sparsity 5, even when the true sparsity is 25. Clearly, this implies that the bias of the estimator is high, since it is searching within a small class of models that does not contain the truth. When the sparsity being searched over is large enough, the error now drops to something reasonable.

(If you wanted to, you could prove tighter concentration for the maximum of iid Gaussian random variables by invoking/proving the weak-law-of-large numbers for the maximum of iid random variables and introducing an appropriate tolerance \( \epsilon \), etc. But we don’t ask you to do that here to keep the math lighter.)

**Solution:** As provided in the hint, the maximum is large if and only if at least one of the individual Gaussians is large. In other words, for every \( t > 0 \), we have the inclusion of events

\[
\left\{ \max_{i \in \{1,2,...,d\}} \left| Z_i \right| \geq t \right\} = \bigcup_{i \in \{1,2,...,d\}} \left\{ \left| Z_i \right| \geq t \right\}.
\]

Using the union bound, we therefore have

\[
\Pr \left\{ \max_{i \in \{1,2,...,d\}} \left| Z_i \right| \geq t \right\} \leq \sum_{i=1}^{d} \Pr \left\{ \left| Z_i \right| \geq t \right\} 
\leq \sum_{i=1}^{d} e^{-\frac{t^2}{2\sigma^2}} 
= de^{-\frac{t^2}{2\sigma^2}} 
= e^{-\frac{t^2}{2\sigma^2} + \log d},
\]

where we have used the first hint to bound the probability of each individual Gaussian being large. Finally, substituting \( t = 2\sigma \sqrt{\log d} \), we have

\[
\Pr \left\{ \max_{i \in \{1,2,...,d\}} \left| Z_i \right| \geq 2\sigma \sqrt{\log d} \right\} \leq e^{-\log d} = 1/d,
\]

which completes the proof.
(e) As in the previous problem, for algebraic convenience, we will restrict attention in this entire problem to the special case where the input data matrix \( X \) has orthonormal columns.

Show that \( \hat{w}_{\text{top}}(s) \) returns the top \( s \) entries of the vector \( w^* + z' \) in absolute value, where \( z' \) is i.i.d. Gaussian with variance \( \sigma^2 \).

Solution: In order to show this, it is sufficient to show that \( \hat{w}_{\text{LS}} = w^* + z' \). After all, the definition says \( \hat{w}_{\text{top}}(s) = \tau_s(\hat{w}_{\text{LS}}) \).

This is easy to do. since

\[
\hat{w}_{\text{LS}} = X^\top y = X^\top (Xw^* + z) = w^* + X^\top z = w^* + z',
\]

where we have used the fact that \( X^\top X = I \). To conclude, we must show that \( z' \) is i.i.d. Gaussian. It is clearly Gaussian, since it is a linear combination of i.i.d. Gaussians. Additionally, \( \mathbb{E}[z'z'^\top] = X^\top \mathbb{E}[zz^\top]X = \sigma^2 I \), and a multivariate Gaussian with covariance equal to the scaled identity is i.i.d.

(f) Argue that the (random) error vector \( e = \hat{w}_{\text{top}}(s) - w^* \) is always (at most) \( 2s \)-sparse.

Solution: This is true by definition, since both vectors \( w^* \) and \( \hat{w}_{\text{top}}(s) \) are \( s \)-sparse. The difference has the largest number of non-zeros when both of the sparsity patterns are disjoint, and so \( e \) is at most \( 2s \)-sparse.

(g) Let us now condition on the event \( \mathcal{E} = \{ \max |z'_i| \leq 2\sigma \sqrt{\log d} \} \). Conditioned on this event, show that we have \( |e_i| \leq 4\sigma \sqrt{\log d} \) for each index \( i \).

Solution: Conditioned on the event \( \mathcal{E} \), we may assume that all the noise is actually bounded by \( 2\sigma \sqrt{\log d} \) in absolute value. Now notice that by the thresholding operation, we have that

\[
e_i = \begin{cases} 
0 & \text{if } w^*_i = [\hat{w}_{\text{top}}(s)]_i = 0 \\
\hat{z}_i & \text{if } w^*_i = 0 \text{ and } [\hat{w}_{\text{top}}(s)]_i \neq 0 \\
\hat{w}^*_i & \text{if } w^*_i \neq 0 \text{ and } [\hat{w}_{\text{top}}(s)]_i = 0
\end{cases}
\]

In addition, there are at most \( 2s \) indices such that one of the latter two cases holds.

If \( e_i = \hat{z}_i \), then clearly, we have \( |e_i| \leq 2\sigma \sqrt{\log d} \leq 4\sigma \sqrt{\log d} \). It remains to handle the final case.

Notice that a non-zero entry of \( w^* \) (let us call this index \( k \)) only gets zeroed out by the thresholding operation if it becomes one of the \( d - s \) smallest entries of \( w^* + z' \). This can only happen if some zero entry of \( w^* \) (say entry \( \ell \)) was such that \( |(w^* + z')_{\ell}| > |(w^* + z')_k| \), which implies that \( |z'_k| > |w^*_k + z'_k| \). Clearly, if \( |w^*_k| > 4\sigma \sqrt{\log d} \), this can never occur, since \( |z'_i| \leq 2\sigma \sqrt{\log d} \) for all \( i \).

Consequently, we must have that for any such index \( k \) which is zeroed out, \( |w^*_k| \leq 4\sigma \sqrt{\log d} \), which results in the required result \( |e_i| \leq 4\sigma \sqrt{\log d} \) for each index \( i \).
(h) Conclude that with probability at least $1 - 1/d$, we have
\[
\|\hat{w}_{\text{top}}(s) - w^*\|_2^2 \leq 32\sigma^2 s \log d.
\]

Solution: We know that conditioned on the event $E$ from the previous part, we have $|e_i| \leq 4\sigma \sqrt{\log d}$ for each index $i$. Additionally, there are at most $2s$ indices $i$ for which $e_i$ is non-zero. Consequently, we have
\[
\|\hat{w}_{\text{top}}(s) - w^*\|_2^2 = \sum_{i=1}^{d} e_i^2 \\
\leq (2s) \cdot 16\sigma^2 \log d \\
= 32\sigma^2 s \log d.
\]

By our previous calculations, the event $E$ occurs with probability at least $1 - 1/d$, which completes the proof.

Clearly, we have $32\sigma^2 s \log d \leq \sigma^2 d$ whenever $s \leq \frac{d}{32 \log d}$. In particular, when $d$ grows with $s$ remaining a constant, we see significantly better estimation by exploiting the sparsity in our models than the naive least squares estimator.

(i) Use the above part to show that with probability at least $1 - 1/d$, we have
\[
\frac{1}{n} \|X(\hat{w}_{\text{top}}(s) - w^*)\|_2^2 \leq 32\sigma^2 s \log d/n.
\]

Solution: By unitary invariance of the Euclidean norm (note that $X$ has orthonormal columns), we have from the calculations of the previous part that
\[
\frac{1}{n} \|X(\hat{w}_{\text{top}}(s) - w^*)\|_2^2 = \frac{1}{n} \|\hat{w}_{\text{top}}(s) - w^*\|_2^2 \\
\leq 32\sigma^2 s \log d/n.
\]

(j) Recall that we have already analyzed the performance of the simple least-squares estimator $\hat{w}_{\text{LS}} = (X^\top X)^{-1}X^\top y$ in earlier homework and the earlier practice midterm. In particular, we showed that
\[
\mathbb{E} \left[ \frac{1}{n} \|X(\hat{w}_{\text{LS}} - w^*)\|_2^2 \right] = \sigma^2 \frac{d}{n}, \quad \text{(1)}
\]
\[
\mathbb{E} \left[ \|\hat{w}_{\text{LS}} - w^*\|_2^2 \right] = \sigma^2 \text{trace} \left[ (X^\top X)^{-1} \right], \quad \text{(2)}
\]
respectively. Equations (1) and (2) represent the “prediction” error (or variance) and the mean squared error of the parameters of our OLS model, respectively. Recall that in this problem, we have been assuming that the matrix $X$ has orthonormal columns, and so the bounds become
\[
\mathbb{E} \left[ \frac{1}{n} \|X(\hat{w}_{\text{LS}} - w^*)\|_2^2 \right] = \sigma^2 \frac{d}{n}, \quad \text{and} \quad \text{(3)}
\]
\[ \mathbb{E} \left[ \| \tilde{\mathbf{w}}_{\text{LS}} - \mathbf{w}^* \|_2^2 \right] = \sigma^2 \frac{d}{n}. \] (4)

Compare these to the previous part, assume that the statement proved with probability \(1 - \frac{1}{d}\) can actually be modified with more work to be an analogous statement that holds with high probability, and conclude that by leveraging the sparsity assumption, we have smaller variance as long as \(s \leq \frac{d}{32 \log d}\).

**Solution:** Comparing the variance of the sparse inducing estimator \(32 \sigma^2 \frac{s \log d}{n}\) with the LS variance \(\sigma^2 \frac{d}{n}\), we see gains when \(s \leq \frac{d}{32 \log d}\).

In conclusion, roughly speaking the sparse solution has a mean prediction error upper bound of \(O(\sigma^2 \frac{s \log d}{n})\) with high probability. On the other hand, the least square solution has an expected mean prediction error of \(\sigma^2 \frac{d}{n}\). Thus when \(s \log d < c \times d\), the sparse solution has smaller error than the least square solution, where \(c\) is some constant.

(k) **Now consider the case if we already knew the important \(s\) features to begin with.** What would be the variance of the sparse OLS estimator that just used the \(s\) important features? How does this variance compare to the behavior for the sparse \(\hat{\mathbf{w}}_{\text{top}}(s)\) derived above? What is the price of not knowing which are the important features?

**Solution:** If we knew the important \(s\) features, the variance would be \(\sigma^2 \frac{s}{n}\), comparing to the variance of the sparse estimator, \(32 \sigma^2 \frac{s \log d}{n}\), we pay a price of \(\alpha \log d\), where \(\alpha\) is some constant.

4 Decision Trees and Random Forests

In this problem, you will implement decision trees, random forests, and boosted trees for classification on two datasets:

1. Titanic Dataset: predict Titanic survivors
2. Spam Dataset: predict if a message is spam

In lecture, you were given a basic introduction to decision trees and how such trees are learned from training data. You were also introduced to random forests. Feel free to research different decision-tree training techniques online.

**NOTE:** You should NOT use any software package for decision trees for Part (a).

(a) **Implement the information gain,** i.e., entropy of the parent node minus the weighted sum of entropy of the child nodes and **Gini purification,** i.e., Gini impurity of the parent node minus the weighted sum of Gini impurities of the child nodes splitting rules for greedy decision tree learning.

See decision_tree_starter.py for the recommended starter code. The code sample is a simplified implementation, which combines decision tree and decision node functionalities and splits only on one feature at a time. **Include your code for information gain and Gini purification.**
Note: The sample implementation assumes that all features are continuous. You may convert all your features to be continuous or augment the implementation to handle discrete features.

Solution:

def entropy(y):
    if y.size == 0:
        return 0
    p0 = np.where(y < 0.5)[0].size / y.size
    if np.abs(p0) < 1e-10 or np.abs(1 - p0) < 1e-10:
        return 0
    return -p0 * np.log(p0) - (1 - p0) * np.log(1 - p0)

def information_gain(X, y, thresh):
    base = entropy(y)
    y0 = y[np.where(X < thresh)[0]]
    p0 = y0.size / y.size
    y1 = y[np.where(X >= thresh)[0]]
    p1 = y1.size / y.size
    children = p0 * entropy(y0) + p1 * entropy(y1)
    return base - children

def gini_impurity(X, y, thresh):
    if y.size == 0:
        return 0
    p0 = np.where(y < 0.5)[0].size / y.size
    if np.abs(p0) < 1e-10 or np.abs(1 - p0) < 1e-10:
        return 0
    return 1.0 - p0**2 - (1 - p0)**2

def gini_purification(X, y, thresh):
    base = gini_impurity(y)
    y0 = y[np.where(X < thresh)[0]]
    p0 = y0.size / y.size
    y1 = y[np.where(X >= thresh)[0]]
    p1 = y1.size / y.size
    children = p0 * gini_impurity(y0) + p1 * gini_impurity(y1)
    return base - children

(b) Before applying the decision-tree learning algorithm to the Titanic dataset, we will first preprocess the dataset. In the real-world, pre-processing the data is a very important step since real-life data can be quite imperfect. However, to make this problem easier, we have provided some code to preprocess the data. Look at the code and describe how we deal with the following problems:
• Some data points are missing class labels;
• Some features are not numerical values;
• Some data points are missing some features.

Data Processing for Titanic  Here is a brief overview of the fields in the Titanic dataset.

(a) survived - 1 is survived; 0 is not. This is the class label.
(b) pclass - Measure of socioeconomic status: 1 is upper, 2 is middle, 3 is lower.
(c) sex - Male/Female
(d) age - Fractional if less than 1.
(e) sibsp - Number of siblings/spouses aboard the Titanic
(f) parch - Number of parents/children aboard the Titanic
(g) ticket - Ticket number
(h) fare - Fare.
(i) cabin - Cabin number.
(j) embarked - Port of Embarkation (C = Cherbourg, Q = Queenstown, S = Southampton)

Solution:

• Some data are missing class labels or are blank. For those, we simply remove that data.
• Some features are not numerical values, which is needed for thresholding, such as gender or the Port of Embarkation. For these, we hash the feature value to convert it into one hot vectors. For example, if we are dealing with a feature "make of a car", and the categories are "Toyota", "Honda", "GM", "BMW", etc. We will split it into multiple binary features "is_Toyota", "is_Honda", "is_GM", "is_BMW", etc.
• Some data are missing features. Depending on the dataset, there are multiple ways to do that. In this homework, we fill it using the mode value of that feature. We use the mode instead of the mean or median, as this makes more sense for categorical features such as gender or cabin type, which are not ordered. For such a small dataset, simply removing data with missing features is not an option.

(c) Train a shallow decision tree on the Titanic dataset. (for example, a depth 3 tree, although you may choose any depth that looks good) and visualize your tree. Include for each non-leaf node the feature name and the split rule, and include for leaf nodes the class your decision tree would assign. You may use sklearn in this problem.

We provide you a code snippet to draw sklearn’s tree using pydot and graphviz. If it is hard for you to install these dependencies, you need to draw the diagram by hand.

Solution:
If you need draw the diagram by hand, the following function can be added to the `DecisionTree` class for visualization:

```python
def __repr__(self):
    if self.max_depth == 0:
        return "%s (%s)" % (self.pred, self.labels.size)
    else:
        return "[%s < %s: %s | %s]" % (self.features[self.split_idx], self.thresh, self.left.__repr__(), self.right.__repr__())
```

In the training set, nearly all women in the upper and middle classes survived. For whatever reason, women in the lower class who paid a fare higher than 23.35 largely perished. Men of all socioeconomic classes largely perished.

(d) From this point forward, you are allowed to use `sklearn.tree.*` and the classes we have imported for you below in the starter code snippets. You are NOT allowed to use other functions from `sklearn`. **Implement bagged trees as follows:** for each tree up to $n$, sample with replacement from the original training set until you have as many samples as the training set. Fit a decision tree for each sampling. **Include your bagged trees code.** Below is optional starter code.

```python
import numpy as np
from sklearn.tree import DecisionTreeClassifier
from sklearn.base import BaseEstimator, ClassifierMixin

class BaggedTrees(BaseEstimator, ClassifierMixin):
    def __init__(self, params=None, n=200):
        self.n = n
        self.models = [DecisionTreeClassifier(**params) for _ in range(n)]

    def fit(self, X, y):
        for model in self.models:
            model.fit(X, y)

    def predict(self, X):
        return np.array([model.predict(X) for model in self.models]).mean(axis=0)
```

HW12, ©UCB CS 189, Spring 2018. All Rights Reserved. This may not be publicly shared without explicit permission.
if params is None:
    params = {}
self.params = params
self.n = n
self.decision_trees = [
    DecisionTreeClassifier(random_state=i,
                            **self.params) for i in
    range(self.n)]

def fit(self, X, y):
    # TODO implement function
    pass

def predict(self, X):
    # TODO implement function
    pass

**Solution:**

def fit(self, X, y):
    for i in range(self.n):
        idx = np.random.randint(0, X.shape[0], X.shape[0])
        newX, newy = X[idx, :], y[idx]
        self.decision_trees[i].fit(newX, newy)
    return self

def predict(self, X):
    yhat = [self.decision_trees[i].predict(X)
            for i in range(self.n)]
    return np.array(np.round(np.mean(yhat, axis=0)),
                    dtype=np.bool)

(e) Apply bagged trees to the titanic and spam datasets. Find and state the most common splits made at the root node of the trees. For example:

(a) (“thanks”) < 4 (15 trees)
(b) (“nigeria”) ≥ 1 (5 trees)

**Data format for Spam** The preprocessed spam dataset given to you as part of the homework in spam.data.mat consists of 11,029 email messages, from which 32 features have been extracted as follows:

- 25 features giving the frequency (count) of words in a given message which match the following words: pain, private, bank, money, drug, spam, prescription, creative, height,
featured, differ, width, other, energy, business, message, volumes, revision, path, meter, memo, planning, pleased, record, out.

• 7 features giving the frequency (count) of characters in the email that match the following characters: ;, $, #, !, (, [, &.

The dataset consists of a training set size 5172 and a test set of size 5857.

Solution: Here’s a sample decision tree for spam.

The following are unlikely to appear in spam: ampersands, ”meter”, ”volume”. The following are likely to appear in spam: ”money”, ”private”, ”pain”.

Spam: The first splits are ”exclamation” for all trees (200).

Titanic: The first splits are all ”sex” (200).

(f) Implement random forests as follows: again, for each tree in the forest, sample with replacement from the original training set until you have as many samples as the training set. Learn a decision tree for each sample, this time using a randomly sampled subset of the features (instead of the full set of features) to find the best split on the data. Let m denote the number of features to subsample. Include your random forests code. Below is optional starter code.

class RandomForest(BaggedTrees):
    def __init__(self, params=None, n=200, m=1):
        if params is None:
            params = {}
            # TODO implement function
        pass

Solution: We use the sklearn’s feature max_features to implement the random forests.
def __init__(self, params=None, n=200, m=1):
    if params is None:
        params = {}
    # sklearn has already implemented the max_features
    # we use it to implement the random forest in the
    params['max_features'] = m
    super().__init__(params=params, n=n)

(g) Apply bagged random forests to the titanic and spam datasets. Find and state the most
common splits made at the root node of the trees.

**Solution:** Spam: The first splits are 'exclamation' for 34 trees, 'money' for 29 trees, 'meter'
for 27 trees, 'volumes' for 18 trees, ampersand for 14 trees, pain for 12 trees, prescription
for 12 trees, spam for 9 trees, parenthesis for 9 trees, etc. (200 total). Yours may vary.

Titanic: The first splits are 'sex' for 71 trees, 'pclass' for 38 trees, 'fare' for 35 trees, 'cabin'
for 35 trees, etc. (200 total). Yours may vary.

(h) Implement the AdaBoost algorithm for a boosted random forest as follows: this time, we will
build the trees sequentially. We will collect one sampling at a time and then we will change the
weights on the data after each new tree is fit to generate more trees that focus their attention
on tackling some of the more challenging data points in the training set. Let \( w \in \mathbb{R}^N \) denote
the probability vector for each datum (initially, uniform), where \( N \) denotes the number of
data points. To start off, as before, sample *with replacement* from the original training set
accordingly to \( w \) until you have as many samples as the training set. Fit a decision tree for this
sampling, again using a randomly sampled subset of the features. Compute the weight for tree
\( j \) based on its weighted accuracy:

\[
a_j = \frac{1}{2} \log \frac{1 - e_j}{e_j}
\]

where \( e_j \) is the weighted error:

\[
e_j = \frac{\sum_{i=1}^{N} I_j(x_i)w_i}{\sum_{i=1}^{N} w_i}
\]

and \( I_j(x_i) \) is an indicator for datum \( i \) being *incorrectly classified by this learned tree*.

Then update the weights as follows:

\[
w_i^+ = \begin{cases} 
    w_i \exp(a_j) & \text{if } I_j(x_i) = 1 \\
    w_i \exp(-a_j) & \text{otherwise}
\end{cases}
\]

Repeat until you have \( M \) trees.

Predict by first calculating the score \( z(x, c) \) for a data sample \( x \) and class label \( c \):

\[
z(x, c) = \sum_{j=1}^{M} a_j I_j(x, c).
\]
where $I_j(x, c)$ is now an indicator variable for whether tree $j$ predicts class label $c$ for data $x$. Then, the class with the highest weighted votes is the prediction (classification result):

$$\hat{y} = \arg \max_c z(x, c)$$

Include your boosted random forests code. Below is optional starter code. How are the trees being weighted? Describe qualitatively what this algorithm is doing. What does it mean when $a_i < 0$, and how does the algorithm handle such trees?

class BoostedRandomForest(RandomForest):

def fit(self, X, y):
    self.w = np.ones(X.shape[0]) / X.shape[0] # Weights on data
    self.a = np.zeros(self.n) # Weights on decision trees
    # TODO implement function
    return self

def predict(self, X):
    # TODO implement function
    pass

Solution:

def fit(self, X, y):
    self.w = np.ones(X.shape[0]) / X.shape[0] # Weights on data
    self.a = np.zeros(self.n) # Weights on decision trees
    i = 0
    while i < self.n:
        idx = np.random.choice(X.shape[0], size=X.shape[0], p=self.w)
        newX, newy = X[idx, :], y[idx]
        self.decision_trees[i].fit(newX, newy)
        wrong = np.abs((y - self.decision_trees[i].predict(X)))
        error = wrong.dot(self.w) / np.sum(self.w)
        self.a[i] = 0.5 * np.log((1 - error) / error)
        # Update w
        wrong_idx = np.where(wrong > 0.5)[0]
        right_idx = np.where(wrong <= 0.5)[0]
        self.w[wrong_idx] = self.w[wrong_idx] * np.exp(self.a[i])
        self.w[right_idx] = self.w[right_idx] * np.exp(-self.a[i])
        self.w /= np.sum(self.w)
        i += 1
    return self
def predict(self, X):
    yhat = [self.decision_trees[i].predict(X) for i in range(self.n)]
    p0 = self.a.dot(np.array(yhat) == 0)
    p1 = self.a.dot(np.array(yhat) == 1)
    return np.array(np.argmax(np.vstack([p0, p1]), axis=0), dtype=np.bool)

(i) Apply boosted trees to the titanic and spam datasets. For the spam dataset only: Describe what kind of data are the most challenging to classify and which are the easiest. Give a few examples. Describe your procedure for determining which data are easy or hard to classify.

Solution: Note: Your precise solutions may vary.

The data that was challenging to classify, as measured by the resulting selection probability on the data, included data that simply contained few of the pre-extracted features. For instance, of the 1000 training samples most likely to be sampled by the end of training, 84% contained fewer than 5 instances of the extracted features. On the other hand, in the 1000 training samples least likely to be sampled, 64% contained at least 5 such counts. Excluding punctuation and special characters, of the 1000 data points most likely to be sampled, 87% contains less than 2 such counts, whereas the 1000 data points least likely to be sampled has 81% which contains at least 2 such counts.

Among the examples which are difficult to classify and contain sufficient extracted features, several messages contain many special characters, such as semicolons (up to 20!), parentheses (up to 10!), ampersands (up to 6), dollars, and exclamations. Some of these messages are spam and some are ham, making them difficult to classify. Additional challenging messages include repeated mundane words such as "business" and "message" (up to 5 times).

Here are several concrete examples (the last value is the total frequency count):

Examples among the 200 data most likely to be sampled
Example: Ham [('money', 1.0), ('dollar', 12.0), ('sharp', 1.0), ('exclamation', 16.0), ('parenthesis', 5.0)] 35.0
Example: Spam [('semicolon', 20.0)] 20.0
Example: Ham [('money', 1.0), ('other', 1.0), ('planning', 4.0), ('dollar', 13.0), ('parenthesis', 3.0)] 22.0
Example: Spam [('out', 1.0), ('exclamation', 6.0), ('parenthesis', 1.0)] 8.0
Example: Ham [('out', 1.0), ('exclamation', 7.0)] 8.0
Example: Ham [('dollar', 4.0), ('sharp', 1.0), ('exclamation', 2.0), ('parenthesis', 2.0)] 9.0
Example: Ham [('out', 1.0), ('dollar', 1.0), ('exclamation', 3.0), ('parenthesis', 8.0)] 13.0
Example: Ham [('out', 1.0), ('dollar', 1.0), ('exclamation', 4.0), ('parenthesis', 2.0)] 8.0
(j) **Summarize the performance evaluation of:** a single decision tree, bagged trees, random forests, and boosted trees. For each of the 2 datasets, report your training and validation accuracies. You should use a 3-fold cross validation, i.e., splitting the dataset into 3 parts. You should be reporting 32 numbers (2 datasets × 4 classifiers × (3 + 1) for 3 cross validation accuracies and 1 training accuracy). Describe qualitatively which types of trees and forests performed best. Detail any parameters that worked well for you. In addition, for each of the 2 datasets, **train your best model and submit your predictions on the test data to Gradescope.** Your best Titanic classifier should exceed 73% accuracy and your best Spam classifier should exceed 76% accuracy for full points.

**Solution: For Titanic Dataset:**

**Part 0: constant classifier**
Accuracy 0.632258064516

**Part (a): simplified decision tree**
Accuracy 0.758064516129

**Part (a): sklearn’s decision tree**
Cross validation [ 0.79640719 0.77477477 0.78915663]
Accuracy 0.774193548387

**Part (c): bagged trees**
Cross validation [ 0.80838323 0.76276276 0.79216867]
Accuracy 0.790322580645

**Part (e): random forest**
Cross validation [ 0.81437126 0.77177177 0.79518072]
Accuracy 0.793548387097

Part (g): boosted random forest
Cross validation [ 0.79341317 0.77477477 0.77710843]
Accuracy 0.783870967742

For Spam Dataset:

Part 0: constant classifier
Accuracy 0.744579136076

Part (a): simplified decision tree
Accuracy 0.747481645894

Part (a): sklearn’s decision tree
Cross validation [ 0.80104408 0.80800464 0.77900232]
Accuracy 0.741676626259

Part (c): bagged trees
Cross validation [ 0.79930394 0.80858469 0.78828306]
Accuracy 0.742359569746

Part (e): random forest
Cross validation [ 0.80220418 0.80858469 0.78770302]
Accuracy 0.813214956462

Part (g): boosted random forest
Cross validation [ 0.8300464 0.83178654 0.79582367]
Accuracy 0.776165272324

(k) You should submit

- a PDF write-up containing your answers, plots, and code to Gradescope;
- a .zip file of your code.
- a file, named submission.txt, of your titanic predictions (one per line).
- a file, named submission.txt, of your spam predictions (one per line).

5 Your Own Question

Write your own question, and provide a thorough solution.

Writing your own problems is a very important way to really learn the material. The famous “Bloom’s Taxonomy” that lists the levels of learning is: Remember, Understand, Apply, Analyze,
Evaluate, and Create. Using what you know to create is the top-level. We rarely ask you any HW questions about the lowest level of straight-up remembering, expecting you to be able to do that yourself. (e.g. make yourself flashcards) But we don’t want the same to be true about the highest level.

As a practical matter, having some practice at trying to create problems helps you study for exams much better than simply counting on solving existing practice problems. This is because thinking about how to create an interesting problem forces you to really look at the material from the perspective of those who are going to create the exams.

Besides, this is fun. If you want to make a boring problem, go ahead. That is your prerogative. But it is more fun to really engage with the material, discover something interesting, and then come up with a problem that walks others down a journey that lets them share your discovery. You don’t have to achieve this every week. But unless you try every week, it probably won’t happen ever.