

This homework is due **Tuesday, November 21 at 10pm.**

1 Getting Started

You may typeset your homework in latex or submit neatly handwritten and scanned solutions. Please make sure to start each question on a new page, as grading (with Gradescope) is much easier that way! Deliverables:

1. Submit a PDF of your writeup to assignment on Gradescope, “HW[n] Write-Up”
2. Submit all code needed to reproduce your results, “HW[n] Code”.
3. Submit your test set evaluation results, “HW[n] Test Set”.

After you’ve submitted your homework, be sure to watch out for the self-grade form.

- (a) Before you start your homework, write down your team. Who else did you work with on this homework? List names and email addresses. In case of course events, just describe the group. How did you work on this homework? Any comments about the homework?

- (b) Please copy the following statement and sign next to it:

I certify that all solutions are entirely in my words and that I have not looked at another student’s solutions. I have credited all external sources in this write up.

2 ℓ_1 -Regularized Linear Regression

The ℓ_1 -norm is one of the popular regularizers used to enhance the robustness of regression models; this is also called Lasso regression. It promotes sparsity in the resulting solution. In this problem, we will explore the optimization objective of the Lasso.

Assume the training data points are denoted as the rows of a $n \times d$ matrix A and their corresponding output value as an $n \times 1$ vector y . The parameter vector and its optimal value are represented by $d \times 1$ vectors w and w^* , respectively. For the sake of simplicity, assume columns of data have been standardized to have mean 0 and variance 1, and are uncorrelated (i.e. $A^T A = nI$).

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

$$w^* = \operatorname{argmin}_w \{J_\lambda(w) = \frac{1}{2} \|y - Aw\|_2^2 + \lambda \|w\|_1\},$$

where $\lambda > 0$.

- (a) **Show that standardized training data nicely decouples the features, making w^* determined by the i -th feature and the output regardless of other features.** To show this, write $J_\lambda(w)$ in the following form for appropriate functions g and f :

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

where A_i is the i -th column of A .

Solution: Considering A_i as the i -th column of A :

$$\frac{1}{2} \|y - Aw\|_2^2 + \lambda \|w\|_1 = \frac{1}{2} y^T y + \sum_{i=1}^d \{-y^T A_i w_i + \frac{n}{2} w_i^2 + \lambda |w_i|\}$$

- (b) Assume that $w_i^* > 0$. **What is the value of w_i^* in this case?**

Solution: If $w_i^* > 0$, then we want to minimize

$$-y^T A_i w_i + \frac{n}{2} w_i^2 + \lambda w_i$$

Take the derivative and equate to zero, we have:

$$w_i^* = \frac{1}{n} (y^T A_i - \lambda)$$

- (c) Assume that $w_i^* < 0$. **What is the value of w_i^* in this case?**

Solution: If $w_i^* < 0$, then we want to minimize

$$-y^T A_i w_i + \frac{n}{2} w_i^2 - \lambda w_i$$

Take the derivative and equate to zero, we have:

$$w_i^* = \frac{1}{n}(y^T A_i + \lambda)$$

- (d) From the previous two parts, **what is the condition for w_i^* to be zero?**

Solution: From the previous parts, we know $w_i^* = 0$ if none of the above conditions hold, that is;

$$y^T A_i + \lambda \geq 0, \quad y^T A_i - \lambda \leq 0$$

Combining them, we get

$$-\lambda \leq y^T A_i \leq \lambda$$

- (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:

$$w^* = \operatorname{argmin}_w \{J_\lambda(w) = \frac{1}{2} \|y - Aw\|_2^2 + \lambda \|w\|_2^2\},$$

where $\lambda > 0$.

What is the condition for $w_i^* = 0$? How does it differ from the condition you obtained in the previous part? Can you see why the ℓ_1 norm promotes sparsity?

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

$$-y^T A_i w_i + \frac{n}{2} w_i^2 + \lambda w_i^2$$

take the derivative and equate to zero:

$$w_i^* = \frac{y^T A_i}{n + 2\lambda}$$

It is equal to zero if $y^T X_i = 0$ or λ goes to infinity. In contrast, $w_i^* = 0$ when $|y^T A_i| < \lambda$ in 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 an orthonormal noise matrix $n \times d$ matrix A and an $n \times d$ noise vector z where $n > d$. Our measurements take the form $y = Aw + z$. We want to extract the original image w given matrix A knowing that this image is sparse. The fact that w is sparse suggests using ℓ_1 regularization. Use the provided iPython notebook and apply ℓ_1 regularization. **Change the hyperparameter λ to extract the best looking image and report it.**

Solution: Please refer to the provided code.

3 Variance of Sparse Linear Models

In this question, we will analyze the variance of sparse linear models. In particular, we will analyze two procedures 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.

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

$$y = Aw^* + z,$$

where $y \in \mathbb{R}^n$ denotes a vector of responses, $A \in \mathbb{R}^{n \times d}$ is our data matrix, $w^* \in \mathbb{R}^d$ is an unknown, s -sparse vector (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 variance σ^2 .

Before we begin analyzing this model, recall that we have already analyzed the performance of the simple least-squares estimator $\hat{w}_{\text{LS}} = (A^\top A)^{-1} A^\top y$. In particular, we showed in (Problem 4)[HW 3] and the (Problem 3)[Practice Midterm] that

$$\mathbb{E} \left[\frac{1}{n} \|A(\hat{w}_{\text{LS}} - w^*)\|_2^2 \right] = \sigma^2 \frac{d}{n}, \text{ and} \quad (1)$$

$$\mathbb{E} [\|\hat{w}_{\text{LS}} - w^*\|_2^2] = \sigma^2 \text{trace} \left[(A^\top A)^{-1} \right], \quad (2)$$

respectively. Equations (1) and (2) represent the “prediction” error (or variance) and parameter estimation error of our model, respectively. For algebraic convenience, *we will assume in this problem that the matrix A has orthonormal columns*, and so the bounds become

$$\mathbb{E} \left[\frac{1}{n} \|A(\hat{w}_{\text{LS}} - w^*)\|_2^2 \right] = \sigma^2 \frac{d}{n}, \text{ and} \quad (3)$$

$$\mathbb{E} [\|\hat{w}_{\text{LS}} - w^*\|_2^2] = \sigma^2 d, \quad (4)$$

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 λ in absolute value.

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

$$\begin{aligned} \hat{w}_{\text{top}}(s) &= \tau_s(\hat{w}_{\text{LS}}) \\ \hat{w}_T(\lambda) &= T_\lambda(\hat{w}_{\text{LS}}), \end{aligned}$$

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

The first three parts of the problem can be filled out 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 least squares algorithm.**

Solution: See solution notebook.

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

Solution: See solution notebook.

- (c) **Now generate data from a non-sparse linear model, and simulate the estimators. Explain the behavior you are seeing in these plots in terms of the bias-variance tradeoff.**

Solution: See solution notebook.

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

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

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

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.

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

$$\left\{ \max_{i \in \{1, 2, \dots, d\}} |Z_i| \geq t \right\} \subseteq \cup_{i \in \{1, 2, \dots, d\}} \{|Z_i| \geq t\}.$$

Using the union bound, we therefore have

$$\begin{aligned} \Pr \left\{ \max_{i \in \{1, 2, \dots, d\}} |Z_i| \geq t \right\} &\leq \sum_{i=1}^d \Pr \{|Z_i| \geq t\} \\ &\leq \sum_{i=1}^d e^{-\frac{t^2}{2\sigma^2}} \\ &= d e^{-\frac{t^2}{2\sigma^2}} \\ &= e^{-\frac{t^2}{2\sigma^2} + \log d}, \end{aligned}$$

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

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

which completes the proof.

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

Solution: In order to show this, it is sufficient to show by definition of the thresholding operation, that $\widehat{w}_{LS} = w^* + z'$. This is easy to do, since

$$\begin{aligned} \widehat{w}_{LS} &= A^\top y \\ &= A^\top (Aw^* + z) \\ &= w^* + A^\top z \\ &= w^* + z', \end{aligned}$$

where we have used the fact that $A^\top A = I$. 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, $\mathbb{E}[z'z'^\top] = A^\top \mathbb{E}[zz^\top]A = \sigma^2 I$, and a multivariate Gaussian with covariance equal to the scaled identity is i.i.d.

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

Solution: This is true by definition, since both vectors w^* and $\widehat{w}_{\text{top}}(s)$ 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

$$e_i = \begin{cases} 0 & \text{if } w_i^* = [\widehat{w}_{\text{top}}(s)]_i = 0 \\ z_i & \text{if } w_i^* = 0 \text{ and } [\widehat{w}_{\text{top}}(s)]_i \neq 0 \\ -w_i^* & \text{if } w_i^* \neq 0 \text{ and } [\widehat{w}_{\text{top}}(s)]_i = 0 \end{cases}$$

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

If $e_i = z_i$, 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 ℓ) was such that $|(w^* + z')_\ell| > |(w^* + z')_k|$,

which implies that $|z'_\ell| > |w_k^* + z'_k|$. 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**

$$\|\widehat{w}_{\text{top}}(s) - w^*\|_2^2 \leq 32\sigma^2 s \log d.$$

Compare this with equation (4) (assuming for the moment that the above quantity is actually an expectation, which we can also show but haven't). When is parameter estimation with the top- s procedure better than the least squares estimator according to these calculations? Pay particular attention to the regime in which s is constant (we have 10 features in our problem that we consider important, although we continue to collect tons of additional ones.)

Solution: We know that conditioned on the event \mathcal{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

$$\begin{aligned} \|\widehat{w}_{\text{top}}(s) - w^*\|_2^2 &= \sum_{i=1}^d e_i^2 \\ &\leq (2s) \cdot 16\sigma^2 \log d \\ &= 32\sigma^2 s \log d. \end{aligned}$$

By our previous calculations, the event \mathcal{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**

$$\frac{1}{n} \|A(\widehat{w}_{\text{top}}(s) - w^*)\|_2^2 \leq 32\sigma^2 \frac{s \log d}{n},$$

and conclude that we have smaller variance as long as $s \leq \frac{d}{32 \log d}$.

Solution: By unitary invariance of the ℓ_2 norm (note that A has orthonormal columns), we have from the calculations of the previous part that

$$\begin{aligned} \frac{1}{n} \|A(\widehat{w}_{\text{top}}(s) - w^*)\|_2^2 &= \frac{1}{n} \|\widehat{w}_{\text{top}}(s) - w^*\|_2^2 \\ &\leq 32\sigma^2 \frac{s \log d}{n}. \end{aligned}$$

Again, we see gains when $s \leq \frac{d}{32 \log d}$.

(j) **BONUS: Derive the variance of the threshold estimator** $\widehat{w}_T(\lambda)$ with $\lambda = 2\sigma\sqrt{\log d}$.

Solution: In this case, we again condition on the event \mathcal{E} , and notice that if $w_i^* = 0$, then $[\widehat{w}_T(\lambda)]_i$ must also be zero, since $|w_i^* + z'_i| < 2\sigma\sqrt{\log d}$. The resulting error vector therefore has at most s non-zero entries, corresponding to where w^* was non-zero. Additionally, for indices k such that both w_k^* and $[\widehat{w}_T(\lambda)]_k$ are non-zero, we know that the error vector obeys $|e_k| = |z'_k| \leq 2\sigma\sqrt{\log d}$.

It remains to consider indices k such that $w_k^* \neq 0$ and $[\widehat{w}_T(\lambda)]_k = 0$, as before. Note that this is only possible provided $|w_k^* + z'_k| \leq 2\sigma\sqrt{\log d}$, which implies by triangle inequality that $|w_k^*| \leq |w_k^* + z'_k| + |z'_k| \leq 4\sigma\sqrt{\log d}$.

We have thus shown that the error vector is at most s -sparse, and satisfies $|e_i| \leq 4\sigma\sqrt{\log d}$ for each index i . The calculations from part (i) therefore follow, and we have

$$\begin{aligned} \frac{1}{n} \|A(\widehat{w}_T(\lambda) - w^*)\|_2^2 &= \frac{1}{n} \|\widehat{w}_T(\lambda) - w^*\|_2^2 \\ &\leq s \cdot (4\sigma\sqrt{\log d})^2 / n \\ &= 16\sigma^2 \frac{s \log d}{n}. \end{aligned}$$

4 Decision Trees and Random Forests

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

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

The data is available on Piazza, and read the appendix carefully for details. All starter code is available in `decision_tree_starter.py`. Note that we have not provided you with test labels for this homework. You will submit your test labels on Gradescope for evaluation.

In lectures, you were given a basic introduction to decision trees and how such trees are trained. You were also introduced to random forests. Feel free to research different decision tree techniques online.

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

(a) **Implement the information gain splitting rule for a decision tree.**

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 information gain and entropy code.**

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:

```
@staticmethod
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.log2(p0) - (1-p0) * np.log2(1-p0)

@staticmethod
def information_gain(X, y, thresh):
    base = DecisionTree.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
    entropy = p0 * DecisionTree.entropy(y0)
        + p1 * DecisionTree.entropy(y1)
    return base - entropy
```

- (b) Before applying the decision tree to the Titanic dataset, described below, you will need to first preprocess the dataset. See below for the data format. You may use any Python package for this part. **Describe what you did to preprocess the data and why; tell us your design decisions.**

Data Processing for Titanic Here is a brief overview of the fields in the Titanic dataset. You will need to preprocess the dataset in `titanic_training.csv`.

- (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:

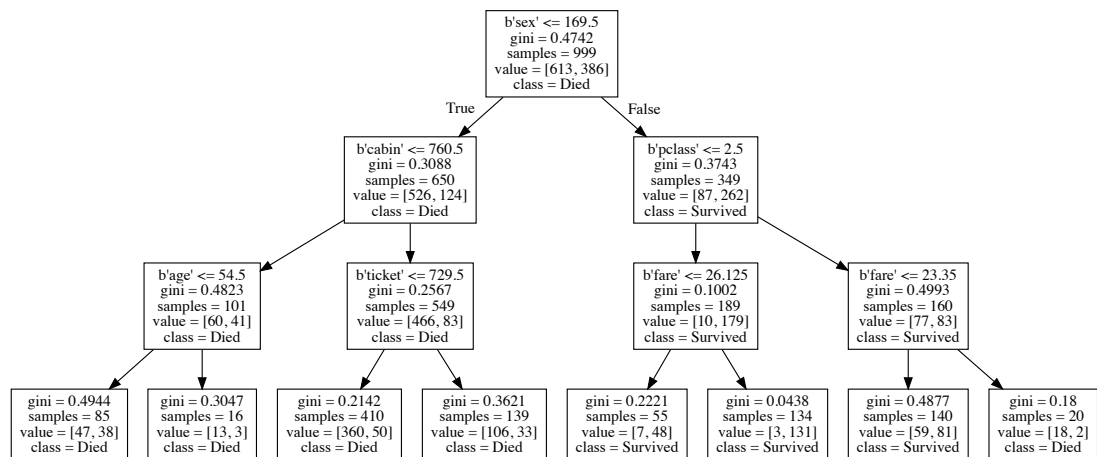
- Some data are missing class labels or are blank. For those, we simply remove that data.
- Some features are not numerical values, which is needed for thresholding, such as gender or the Port of Embarkation. For these, we hash the feature value to convert it into a unique number. Any other conversion of the feature values into numerical values is also valid.
- Some data is missing features. For those, we arbitrarily fill it in as 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.

See solution code for example preprocessing implementation.

- (c) **Apply your decision tree to the titanic dataset. Train a shallow decision tree** (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.

Tip: To see if your decision tree implementation is working without first the preprocessing the Titanic dataset, use the already pre-processed spam dataset.

Solution: NOTE: The following tree is generated by `sklearn`, but yours should not be.



The following function can be added to the `DecisionTree` class for visualization:

```

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.

```
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):
        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:

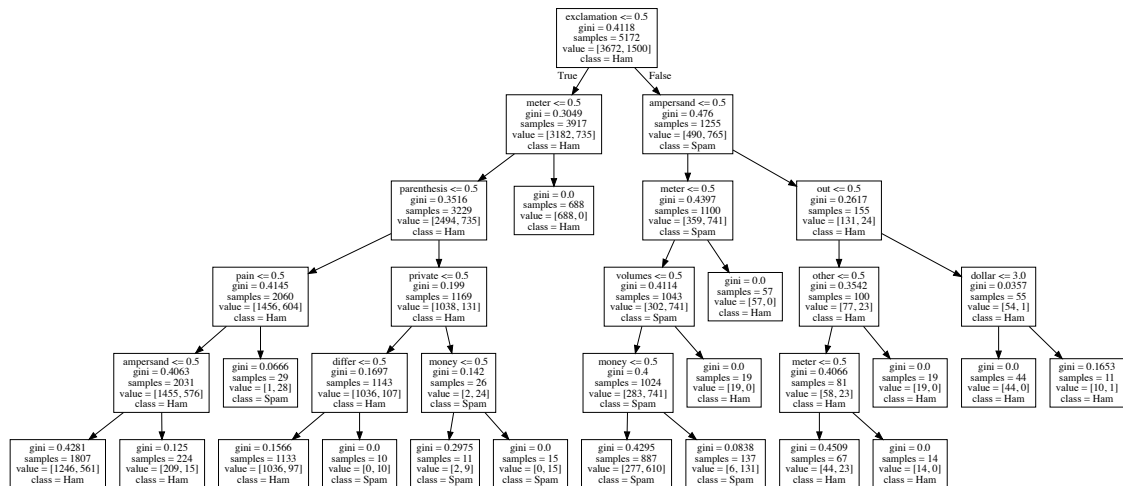
- (a) (“viagra”) ≥ 3 (20 trees)
- (b) (“thanks”) < 4 (15 trees)
- (c) (“nigeria”) ≥ 1 (5 trees)

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. Fit 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:

```
def __init__(self, params=None, n=200, m=1):  
    if params is None:  
        params = {}  
    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 boosted random forests as follows: this time, we will collect one sampling at a time and we will change the weights on the data after each new tree is fit to generate more trees that tackle some of the more challenging data. 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 sample, again using a randomly sampled subset of the features. Compute the weight for tree j based on its 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 *incorrect*.

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 weight $z(x, c)$ 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 if tree j predicts data x with class label c . Then, the class with the highest weight 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.log2((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 random forests 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 [('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

Example: Ham [('business', 1.0), ('dollar', 4.0), ('exclamation', 2.0), ('parenthesis', 2.0)] 9.0

Example: Ham [('out', 1.0), ('exclamation', 8.0)] 9.0

Example: Ham [('out', 1.0), ('dollar', 1.0), ('exclamation', 9.0)] 11.0

Example: Spam [('other', 1.0), ('message', 2.0), ('out', 1.0), ('exclamation', 4.0), ('parenthesis', 2.0)] 10.0

Example: Spam [('business', 1.0), ('dollar', 3.0), ('parenthesis', 12.0), ('ampersand', 3.0)] 19.0

Example: Spam [('message', 5.0), ('parenthesis', 3.0)] 8.0

Example: Ham [('money', 1.0), ('record', 2.0), ('out', 1.0), ('dollar', 1.0), ('exclamation', 4.0), ('parenthesis', 1.0)] 10.0

Example: Ham [('exclamation', 12.0)] 12.0

Example: Spam [('out', 2.0), ('ampersand', 6.0)] 8.0

Example: Ham [('money', 2.0), ('dollar', 2.0), ('exclamation', 4.0), ('parenthesis', 2.0)] 10.0

- (j) **Summarize the performance evaluation of each of the above trees and forests: a single decision tree, bagged trees, random forests, and boosted random forests.** For each of the 2 datasets, report your training and validation accuracies. You should be reporting 24 numbers (2 datasets \times 4 classifiers \times 3 data splits). 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 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 titantic 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 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.