**Question 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) Who else did you work with on this homework? In case of course events, just describe the group. How did you work on this homework? Any comments about the homework?

I worked with Weiran Liu and Katherine Li. I hope this HW goes smoothly.

(b) Please copy the following statement and sign next to it. We just want to make it *extra* clear so that no one inadvertently cheats.

*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.*

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.

Signature: 

**Question 2.** $l_1$-Regularized Linear Regression

The $l_1$-norm is one of the popular regularizers used to enhance the robustness of regression models. Regression with an $l_1$-penalty is referred to as the 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 $\mathbf{X}$ and their corresponding output value as an $n \times 1$ vector $\vec{y}$. The parameter vector and its optimal value are represented by $d \times 1$ vectors $\vec{w}$ and $\vec{w}^*$, 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. $\mathbf{X}^\top \mathbf{X} = n\mathbb{I}$). (*We center the data mean to zero, so that the lasso penalty treats all features similarly. We assume uncorrelated features in order to reasons about Lasso in the upcoming parts.*)

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

$$\vec{w}^* = \arg\min_{\vec{w}} \{J_\lambda(\vec{w}) = \frac{1}{2}\|\vec{y} - \mathbf{X}\vec{w}\|_2^2 + \lambda\|\vec{w}\|_1\},$$

where $\lambda > 0$.

(a) **Show that for data with uncorrelated features, one can learn the parameter $w_i$ corresponding to each $i$-th feature independently from the other features, one at a time, and get a solution which is equivalent to having learned them all jointly as we normally do. To show this, write $J_\lambda(\vec{w})$ in the following form for appropriate functions $g$ and $f$:**

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

**where $\mathbf{X}_i$ is the $i$-th column of $\mathbf{X}$.**

$$\begin{aligned}
&J_\lambda(\vec{w}) = \frac{1}{2}\|\vec{y} - \mathbf{X}\vec{w}\|_2^2 + \lambda\|\vec{w}\|_1 \\
=&\frac{1}{2}(\vec{y} - \mathbf{X}\vec{w})^\top(\vec{y} - \mathbf{X}\vec{w}) + \lambda\sum_{i=1}^{d}|w_i| \\
=&\frac{1}{2}(\vec{y}^\top\vec{y} - 2\vec{y}^\top\mathbf{X}\vec{w} + \vec{w}^\top\mathbf{X}^\top\mathbf{X}\vec{w}) + \lambda\sum_{i=1}^{d}|w_i| \\
=&\frac{1}{2}\vec{y}^\top\vec{y} - \vec{y}^\top\mathbf{X}\vec{w} + \frac{1}{2}\vec{w}^\top\mathbf{X}^\top\mathbf{X}\vec{w} + \lambda\sum_{i=1}^{d}|w_i| \\
=&\frac{1}{2}\vec{y}^\top\vec{y} - \sum_{i=1}^{d}\vec{y}^\top\mathbf{X}_i w_i + \frac{1}{2}\vec{w}^\top n\mathbf{I}\vec{w} + \lambda\sum_{i=1}^{d}|w_i| \\
=&\frac{1}{2}\vec{y}^\top\vec{y} - \sum_{i=1}^{d}\vec{y}^\top\mathbf{X}_i w_i + \sum_{i=1}^{d}\frac{n}{2}w_i^2 + \sum_{i=1}^{d}\lambda|w_i| \\
=&\frac{1}{2}\vec{y}^\top\vec{y} + \sum_{i=1}^{d}\left(-\vec{y}^\top\mathbf{X}_i w_i + \frac{n}{2}w_i^2 + \lambda|w_i|\right) \\
=&g(\vec{y}) + \sum_{i=1}^{d} f(\mathbf{X}_i, \vec{y}, w_i, \lambda)
\end{aligned}$$

As we can see, the loss function can be written as a sum of individual features. That is, the derivative of j-th feature only depends on the data of j-th feature.

$$\frac{\partial}{\partial w_j} J_\lambda(\vec{w}) = \frac{d}{dw_j} f(\mathbf{X}_i, \vec{y}, w_i, \lambda)$$

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

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

$$\frac{\partial}{\partial w_i} J_\lambda(\vec{w}) = 0$$
$$-\vec{y}^\top \mathbf{X}_i + n w_i^* + \lambda = 0$$
$$w_i^* = \frac{1}{n}\left(\vec{y}^\top \mathbf{X}_i - \lambda\right)$$

However, $\vec{y}^\top \mathbf{X}_i - \lambda$ is not guaranteed to be positive. Thus, we have:

$$w_i^* = \begin{cases} 0 & \vec{y}^\top \mathbf{X}_i - \lambda \leq 0 \\ \frac{1}{n}\left(\vec{y}^\top \mathbf{X}_i - \lambda\right) & \vec{y}^\top \mathbf{X}_i - \lambda > 0 \end{cases}$$

But $w_i^* > 0$, so we have $w_i^* = \frac{1}{n}\left(\vec{y}^\top \mathbf{X}_i - \lambda\right) > 0$

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

$$\frac{\partial}{\partial w_i} J_\lambda(\vec{w}) = 0$$
$$-\vec{y}^\top \mathbf{X}_i + n w_i^* - \lambda = 0$$
$$w_i^* = \frac{1}{n}\left(\vec{y}^\top \mathbf{X}_i + \lambda\right)$$

However, $\vec{y}^\top \mathbf{X}_i + \lambda$ is not guaranteed to be negative. Thus, we have:

$$w_i^* = \begin{cases} 0 & \vec{y}^\top \mathbf{X}_i + \lambda \geq 0 \\ \frac{1}{n}\left(\vec{y}^\top \mathbf{X}_i + \lambda\right) & \vec{y}^\top \mathbf{X}_i + \lambda < 0 \end{cases}$$

But $w_i^* < 0$, so we have $w_i^* = \frac{1}{n}\left(\vec{y}^\top \mathbf{X}_i + \lambda\right) < 0$

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

Let's copy the solutions to the previous two parts here:
**Part (b):**

$$w_i^* = \begin{cases} 0 & \vec{y}^\top \mathbf{X}_i - \lambda \leq 0 \\ \frac{1}{n}\left(\vec{y}^\top \mathbf{X}_i - \lambda\right) & \vec{y}^\top \mathbf{X}_i - \lambda > 0 \end{cases}$$

**Part (c):**

$$w_i^* = \begin{cases} 0 & \vec{y}^\top \mathbf{X}_i + \lambda \geq 0 \\ \frac{1}{n}\left(\vec{y}^\top \mathbf{X}_i + \lambda\right) & \vec{y}^\top \mathbf{X}_i + \lambda < 0 \end{cases}$$

Amalgamating these two solutions together, we have:

$$w_i^* = \begin{cases} 0 & |\vec{y}^\top \mathbf{X}_i| \leq \lambda \\ \frac{1}{n}\left(\vec{y}^\top \mathbf{X}_i + \lambda\right) & |\vec{y}^\top \mathbf{X}_i| > \lambda \end{cases}$$

Therefore, the condition for $w_i^*$ to be zero is $|\vec{y}^\top \mathbf{X}_i| \leq \lambda$.

**(e) Now consider the ridge regression problem where the regularization term is replaced by $\lambda\|\vec{w}\|_2^2$ where the optimal parameter vector is now given by:**

$$\vec{w}^* = \mathbf{arg}\min_{\vec{w}}\{J_\lambda(\vec{w}) = \frac{1}{2}\|\vec{y} - \mathbf{X}\vec{w}\|_2^2 + \lambda\|\vec{w}\|_2^2\},$$

**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?**

$$\begin{aligned} J_\lambda(\vec{w}) &= \frac{1}{2}\|\vec{y} - \mathbf{X}\vec{w}\|_2^2 + \lambda\|\vec{w}\|_1 \\ &= \frac{1}{2}\vec{y}^\top\vec{y} + \sum_{i=1}^{d}\left(-\vec{y}^\top\mathbf{X}_i w_i + \frac{n}{2}w_i^2 + \lambda w_i^2\right) \end{aligned}$$

Taking the derivative of the loss function, we have:

$$\begin{aligned} \frac{\partial}{\partial w_i}J_\lambda(\vec{w}) &= 0 \\ -\vec{y}^\top\mathbf{X}_i + nw_i^* + 2\lambda w_i^* &= 0 \\ w_i^* &= \frac{\vec{y}^\top\mathbf{X}_i}{n + 2\lambda} \end{aligned}$$

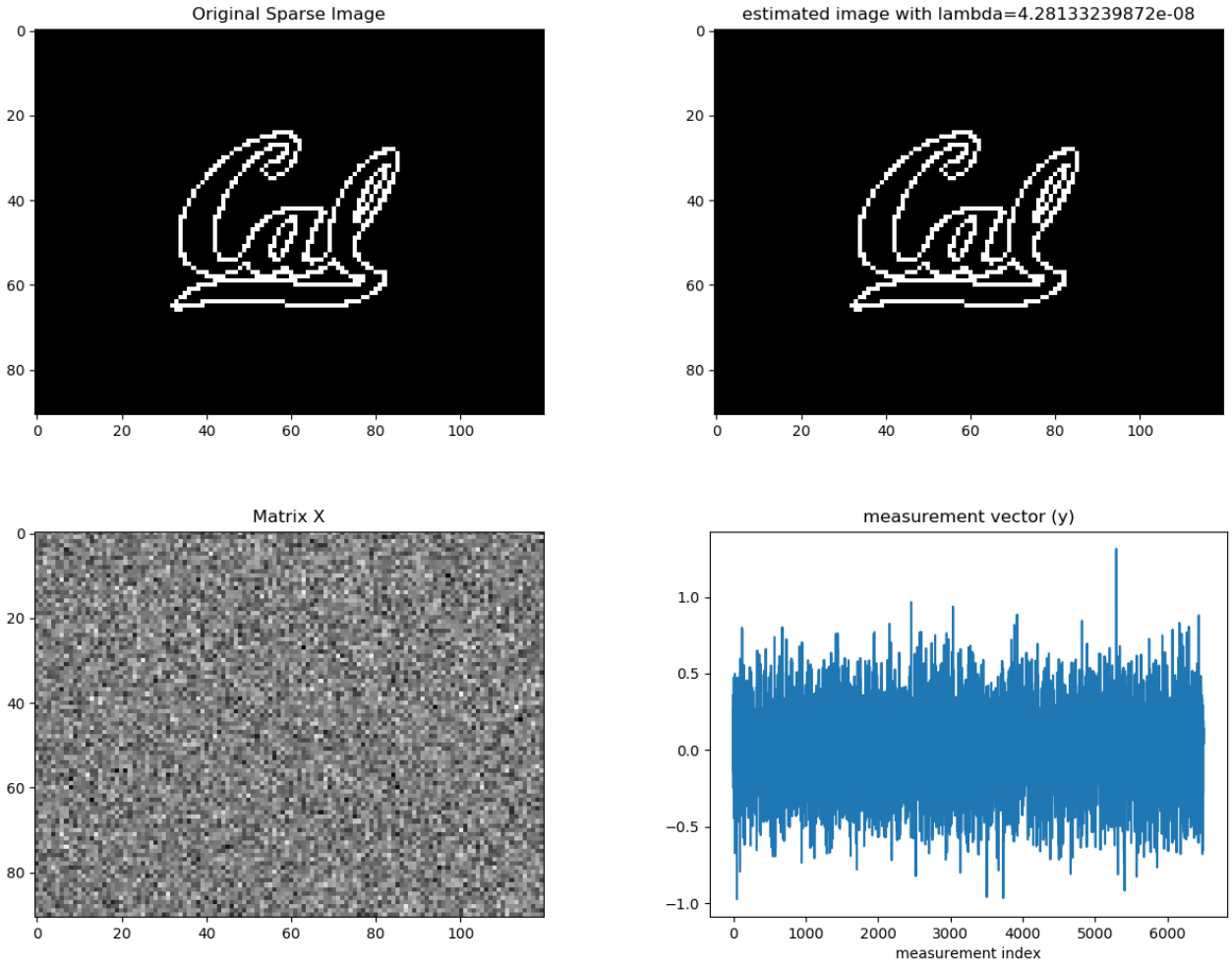
The condition for $w_i^* = 0$ is:

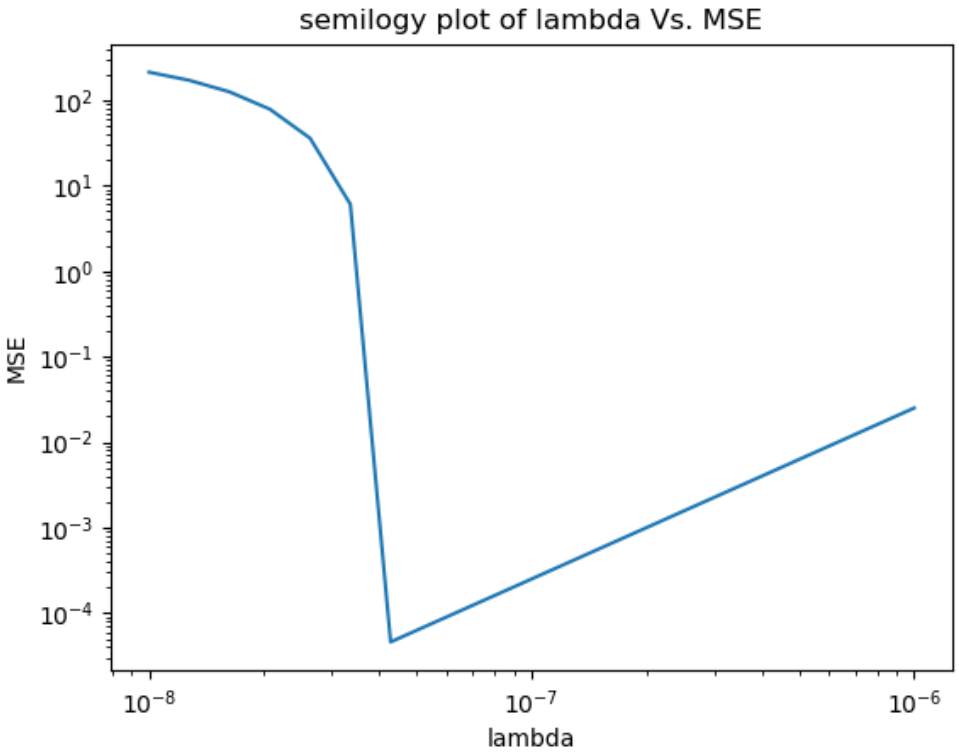
$$\begin{aligned} \frac{\vec{y}^\top\mathbf{X}_i}{n + 2\lambda} &= 0 \\ \vec{y}^\top\mathbf{X}_i &= 0 \end{aligned}$$

The condition for $w_i^* = 0$ is very strict and unlikely in practice. On the other hand, $l_1$ norm regularization gives a range where $w_i^* = 0$. This property makes $l_1$ norm regularization promote sparsity.

**(f) Assume that we have a sparse image vectorized in the vector $\vec{w}$ (so $\vec{w}$ is a sparse vector). We have a gaussian matrix $n \times d$ matrix X and an $n \times 1$ noise vector $\vec{F}$ where $n > 1$. Our measurements take the form $\vec{y} = \mathbf{X}\vec{w} + \vec{F}$. We want to extract the original image $\vec{w}$ given matrix X knowing that this image is sparse. The fact that $\vec{w}$ is sparse suggests using $l_1$ regularization. Use the provided iPython notebook and apply $l_1$ regularization. Change the hyperparameter $\lambda$ to extract the best looking image and report it.**

The one on the left is the original image and the one on the right is the reconstructed image with a $\lambda \approx 10^{-7}$.






---

```
import numpy as np
import matplotlib.pyplot as plt
#matplotlib inline


def ground_truth(n, d, s):
"""
Input: Two positive integers n, d. Requires n >= d >=s. If d<s, we let s = d
Output: A tuple containing i) random matrix of dimension n X d with orthonormal columns. and
ii) a d-dimensional, s-sparse wstar with (large) Gaussian entries
"""
if d > n:
print("Too many dimensions")
return None

if d < s:
s = d
A = np.random.randn(n, d) # random Gaussian matrix
U, S, V = np.linalg.svd(A, full_matrices=False) # reduced SVD of Gaussian matrix
wstar = np.zeros(d)
wstar[:s] = 10 * np.random.randn(s)

np.random.shuffle(wstar)
return U, wstar


def get_obs(U, wstar):
"""
Input: U is an n X d matrix and wstar is a d X 1 vector.
Output: Returns the n-dimensional noisy observation y = U * wstar + z.
"""
n, d = np.shape(U)
z = np.random.randn(n) # i.i.d. noise of variance 1
y = np.dot(U, wstar) + z
```

```
    return y


def LS(U, y):
"""
Input: U is an n X d matrix with orthonormal columns and y is an n X 1 vector.
Output: The OLS estimate what_{LS}, a d X 1 vector.
"""
wls = np.dot(U.T, y) # pseudoinverse of orthonormal matrix is its transpose
return wls


def thresh(U, y, lmbda):
"""
Input: U is an n X d matrix and y is an n X 1 vector; lambda is a scalar threshold of the entries.
Output: The estimate what_{T}(lambda), a d X 1 vector that is hard-thresholded (in absolute value) at
    level lambda.
When code is unfilled, returns the all-zero d-vector.
"""
n, d = np.shape(U)
wls = LS(U, y)
what = np.zeros(d)

# print np.shape(wls)
##########
# TODO: Fill in thresholding function; store result in what
######################
# YOUR CODE HERE:
what[abs(wls) < lmbda] = wls[abs(wls) < lmbda]
################
return what


def topk(U, y, s):
"""
Input: U is an n X d matrix and y is an n X 1 vector; s is a positive integer.
Output: The estimate what_{top}(s), a d X 1 vector that has at most s non-zero entries.
When code is unfilled, returns the all-zero d-vector.
"""
n, d = np.shape(U)
what = np.zeros(d)
wls = LS(U, y)

##########
# TODO: Fill in thresholding function; store result in what
######################
# YOUR CODE HERE: Remember the absolute value!
sorted_indices = np.argsort(abs(wls))
what[sorted_indices[-(s+1):-1]] = wls[[sorted_indices[-(s+1):-1]]]
################
return what


def error_calc(num_iters=10, param='n', n=1000, d=100, s=5, s_model=True, true_s=5):
"""
Plots the prediction error 1/n || U(what - wstar)||^2 = 1/n || what - wstar ||^2 for the three estimators
averaged over num_iter experiments.

Input:
Output: 4 arrays -- range of parameters, errors of LS, topk, and thresh estimator, respectively. If thresh
    and topk
functions have not been implemented yet, then these errors are simply the norm of wstar.
"""
```

```
wls_error = []
wtopk_error = []
wthresh_error = []

if param == 'n':
arg_range = np.arange(100, 2000, 50)
lmbda = 2 * np.sqrt(np.log(d))
for n in arg_range:
U, wstar = ground_truth(n, d, s) if s_model else ground_truth(n, d, true_s)
error_wls = 0
error_wtopk = 0
error_wthresh = 0
for count in range(num_iters):
y = get_obs(U, wstar)
wls = LS(U, y)
wtopk = topk(U, y, s)
wthresh = thresh(U, y, lmbda)
error_wls += np.linalg.norm(wstar - wls) ** 2
error_wtopk += np.linalg.norm(wstar - wtopk) ** 2
error_wthresh += np.linalg.norm(wstar - wthresh) ** 2
wls_error.append(float(error_wls) / n / num_iters)
wtopk_error.append(float(error_wtopk) / n / num_iters)
wthresh_error.append(float(error_wthresh) / n / num_iters)

elif param == 'd':
arg_range = np.arange(10, 1000, 50)
for d in arg_range:
lmbda = 2 * np.sqrt(np.log(d))
U, wstar = ground_truth(n, d, s) if s_model else ground_truth(n, d, true_s)
error_wls = 0
error_wtopk = 0
error_wthresh = 0
for count in range(num_iters):
y = get_obs(U, wstar)
wls = LS(U, y)
wtopk = topk(U, y, s)
wthresh = thresh(U, y, lmbda)
error_wls += np.linalg.norm(wstar - wls) ** 2
error_wtopk += np.linalg.norm(wstar - wtopk) ** 2
error_wthresh += np.linalg.norm(wstar - wthresh) ** 2
wls_error.append(float(error_wls) / n / num_iters)
wtopk_error.append(float(error_wtopk) / n / num_iters)
wthresh_error.append(float(error_wthresh) / n / num_iters)

elif param == 's':
arg_range = np.arange(5, 55, 5)
lmbda = 2 * np.sqrt(np.log(d))
for s in arg_range:
U, wstar = ground_truth(n, d, s) if s_model else ground_truth(n, d, true_s)
error_wls = 0
error_wtopk = 0
error_wthresh = 0
for count in range(num_iters):
y = get_obs(U, wstar)
wls = LS(U, y)
wtopk = topk(U, y, s)
wthresh = thresh(U, y, lmbda)
error_wls += np.linalg.norm(wstar - wls) ** 2
error_wtopk += np.linalg.norm(wstar - wtopk) ** 2
error_wthresh += np.linalg.norm(wstar - wthresh) ** 2
wls_error.append(float(error_wls) / n / num_iters)
wtopk_error.append(float(error_wtopk) / n / num_iters)
wthresh_error.append(float(error_wthresh) / n / num_iters)
```

```
return arg_range, wls_error, wtopk_error, wthresh_error


#nrange contains the range of n used, ls_error the corresponding errors for the OLS estimate
nrange, ls_error, _, _ = error_calc(num_iters=10, param='n', n=1000, d=100, s=5, s_model=True, true_s=5)

########
#TODO: Your code here: call the helper function for d and s, and plot everything
########
#YOUR CODE HERE:


#TODO: Part (b)
###############
#YOUR CODE HERE:


#TODO: Part (c)
###############
#YOUR CODE HERE:
```

**Question 3.** Variance of Sparse Linear Models Obtained by Thresholding

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.

However note that there is a subtle difference between feature selection before training and after training. If we use less number of 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) does not hurt either, if done in certain ways. In other words, although there is a 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

$$y = \mathbf{X}w^* + z,$$

where $y \in \mathbb{R}^n$ denotes a vector of responses, $\mathbf{X} \in \mathbb{R}^{n \times d}$ 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 $\sigma^2$.

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

$$\mathbb{E}\left[\frac{1}{n}\|\mathbf{X}(\widehat{w}_{\mathsf{LS}} - w^*)\|_2^2\right] = \sigma^2 \frac{d}{n}, \text{ and} \tag{1}$$

$$\mathbb{E}\left[\|\widehat{w}_{\mathsf{LS}} - w^*\|_2^2\right] = \sigma^2 \mathsf{trace}\left[(\mathbf{X}^\top \mathbf{X})^{-1}\right], \tag{2}$$

respectively. Equations () and (2) represent the "prediction" error (or variance) and the mean squared error of the parameters of our model, respectively. For algebraic convenience, *we will assume in this problem that the matrix* $\mathbf{X}$ *has orthonormal columns*, and so the bounds become

$$\mathbb{E}\left[\frac{1}{n}\|\mathbf{X}(\widehat{w}_{\mathsf{LS}} - w^*)\|_2^2\right] = \sigma^2 \frac{d}{n}, \text{ and} \tag{3}$$

$$\mathbb{E}\left[\|\widehat{w}_{\mathsf{LS}} - w^*\|_2^2\right] = \sigma^2 d, \tag{4}$$

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 $\lambda$ in absolute value.

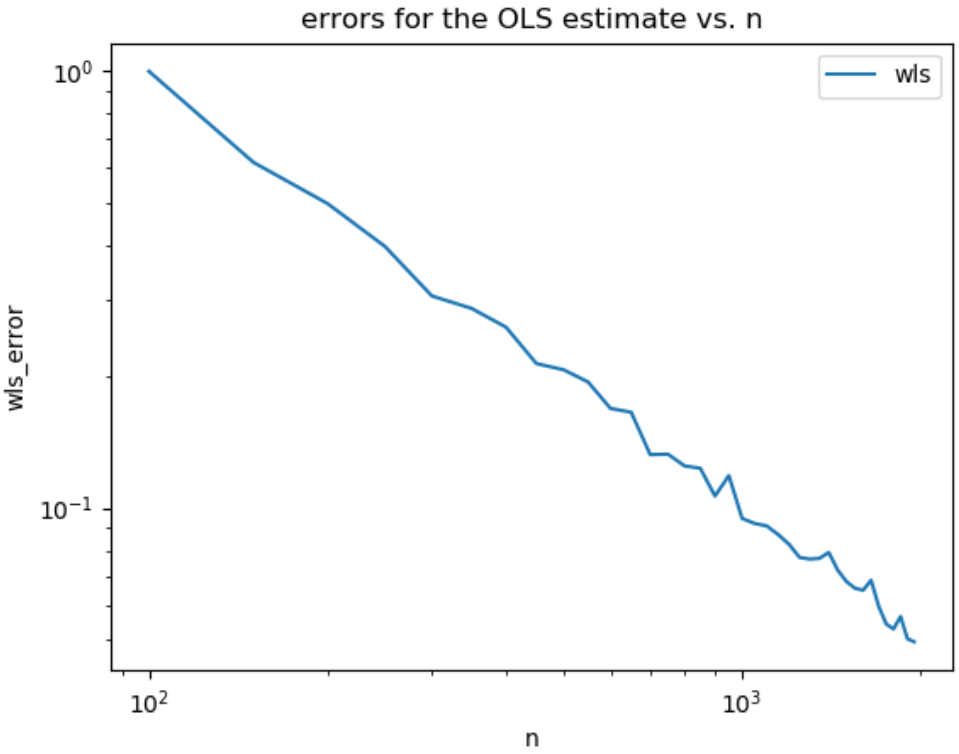
Recall that the least squares estimate was given by $\widehat{w}_{\mathsf{LS}} = \mathbf{X}^\dagger y = \mathbf{X}^\top y$, where $\mathbf{X}^\dagger$ is the pseudo-inverse of $\mathbf{X}$ (it is equal to the transpose since $\mathbf{X}$ has orthonormal columns). We now define

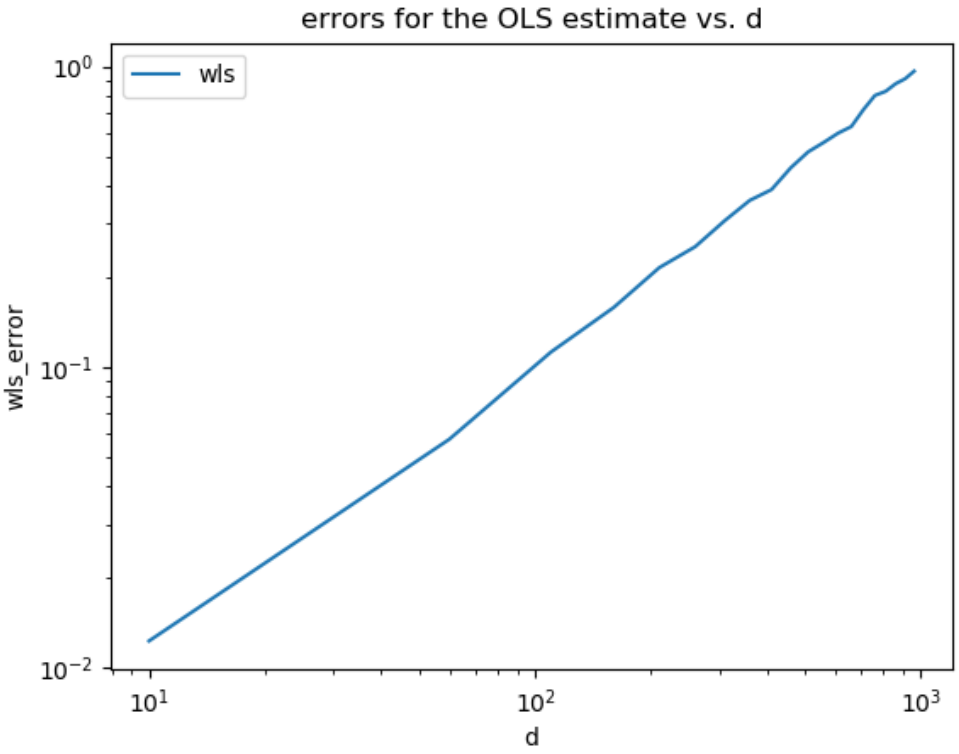
$$\widehat{w}_{\mathsf{top}}(s) = \tau_s(\widehat{w}_{\mathsf{LS}})$$
$$\widehat{w}_T(\lambda) = T_\lambda(\widehat{w}_{\mathsf{LS}}),$$

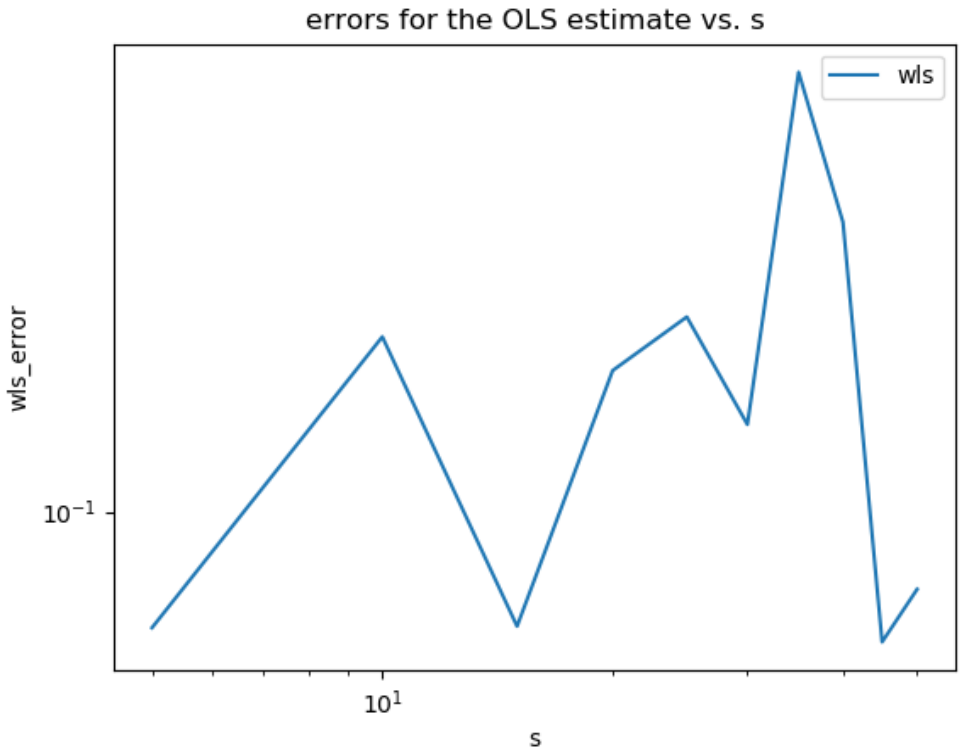
which are the two sparsity-inducing estimators that we will consider.

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 least squares algorithm.**





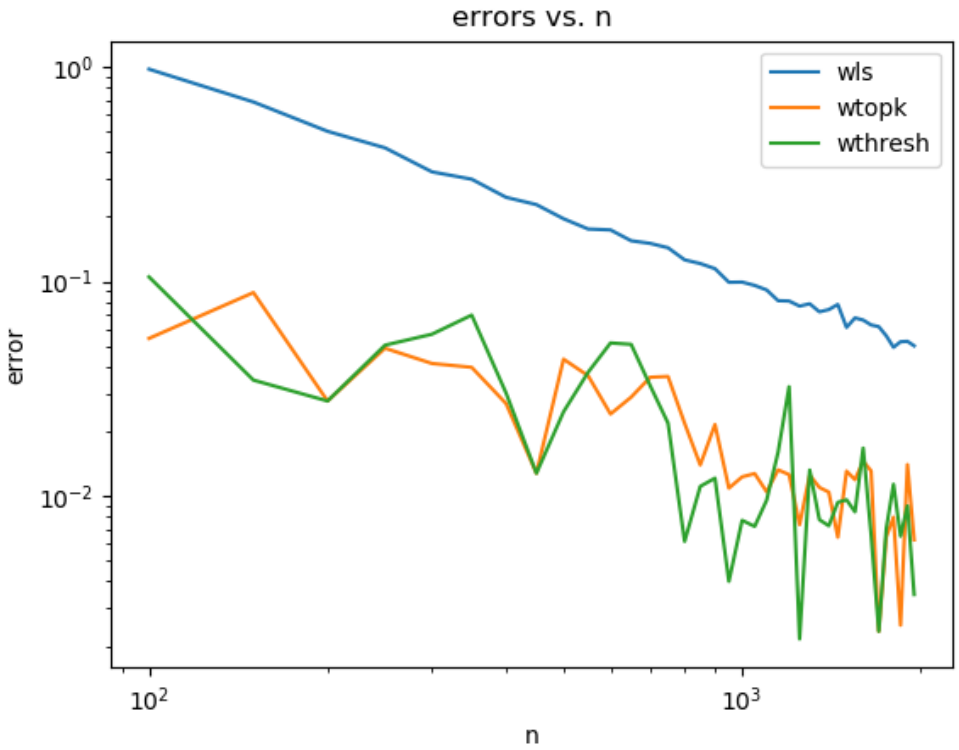

```
import numpy as np
import matplotlib.pyplot as plt
#matplotlib inline


def ground_truth(n, d, s):
"""
Input: Two positive integers n, d. Requires n >= d >=s. If d<s, we let s = d
Output: A tuple containing i) random matrix of dimension n X d with orthonormal columns. and
ii) a d-dimensional, s-sparse wstar with (large) Gaussian entries
"""
if d > n:
print("Too many dimensions")
return None

if d < s:
s = d
A = np.random.randn(n, d) # random Gaussian matrix
U, S, V = np.linalg.svd(A, full_matrices=False) # reduced SVD of Gaussian matrix
wstar = np.zeros(d)
wstar[:s] = 10 * np.random.randn(s)

np.random.shuffle(wstar)
return U, wstar


def get_obs(U, wstar):
"""
Input: U is an n X d matrix and wstar is a d X 1 vector.
Output: Returns the n-dimensional noisy observation y = U * wstar + z.
"""
n, d = np.shape(U)
z = np.random.randn(n) # i.i.d. noise of variance 1
y = np.dot(U, wstar) + z
```

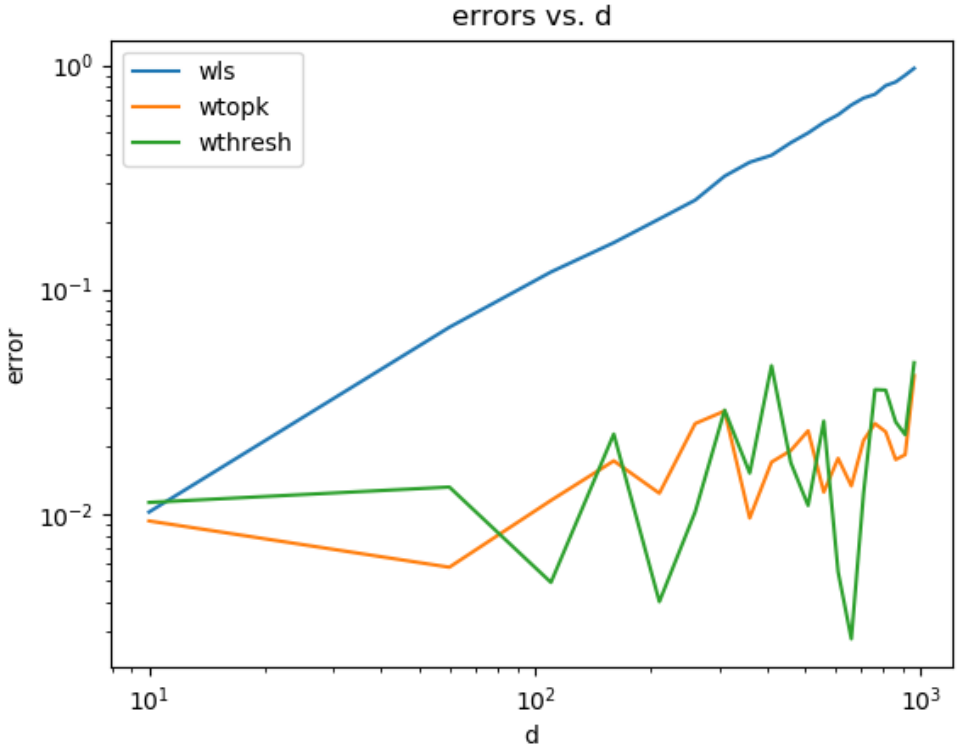
```
    return y


def LS(U, y):
    """
    Input: U is an n X d matrix with orthonormal columns and y is an n X 1 vector.
    Output: The OLS estimate what_{LS}, a d X 1 vector.
    """
    wls = np.dot(U.T, y) # pseudoinverse of orthonormal matrix is its transpose
    return wls


def thresh(U, y, lmbda):
    """
    Input: U is an n X d matrix and y is an n X 1 vector; lambda is a scalar threshold of the entries.
    Output: The estimate what_{T}(lambda), a d X 1 vector that is hard-thresholded (in absolute value) at
        level lambda.
    When code is unfilled, returns the all-zero d-vector.
    """
    n, d = np.shape(U)
    wls = LS(U, y)
    what = np.zeros(d)

    # print np.shape(wls)
    ##########
    # TODO: Fill in thresholding function; store result in what
    ######################
    # YOUR CODE HERE:
    what[abs(wls) >= lmbda] = wls[abs(wls) >= lmbda]
    ################
    return what


def topk(U, y, s):
    """
    Input: U is an n X d matrix and y is an n X 1 vector; s is a positive integer.
    Output: The estimate what_{top}(s), a d X 1 vector that has at most s non-zero entries.
    When code is unfilled, returns the all-zero d-vector.
    """
    n, d = np.shape(U)
    what = np.zeros(d)
    wls = LS(U, y)

    ##########
    # TODO: Fill in thresholding function; store result in what
    ######################
    # YOUR CODE HERE: Remember the absolute value!
    sorted_indices = np.argsort(abs(wls))
    what[sorted_indices[-s:]] = wls[[sorted_indices[-s:]]]
    ################
    return what


def error_calc(num_iters=10, param='n', n=1000, d=100, s=5, s_model=True, true_s=5):
    """
    Plots the prediction error 1/n || U(what - wstar)||^2 = 1/n || what - wstar ||^2 for the three estimators
    averaged over num_iter experiments.

    Input:
    Output: 4 arrays -- range of parameters, errors of LS, topk, and thresh estimator, respectively. If thresh
        and topk
    functions have not been implemented yet, then these errors are simply the norm of wstar.
    """
```

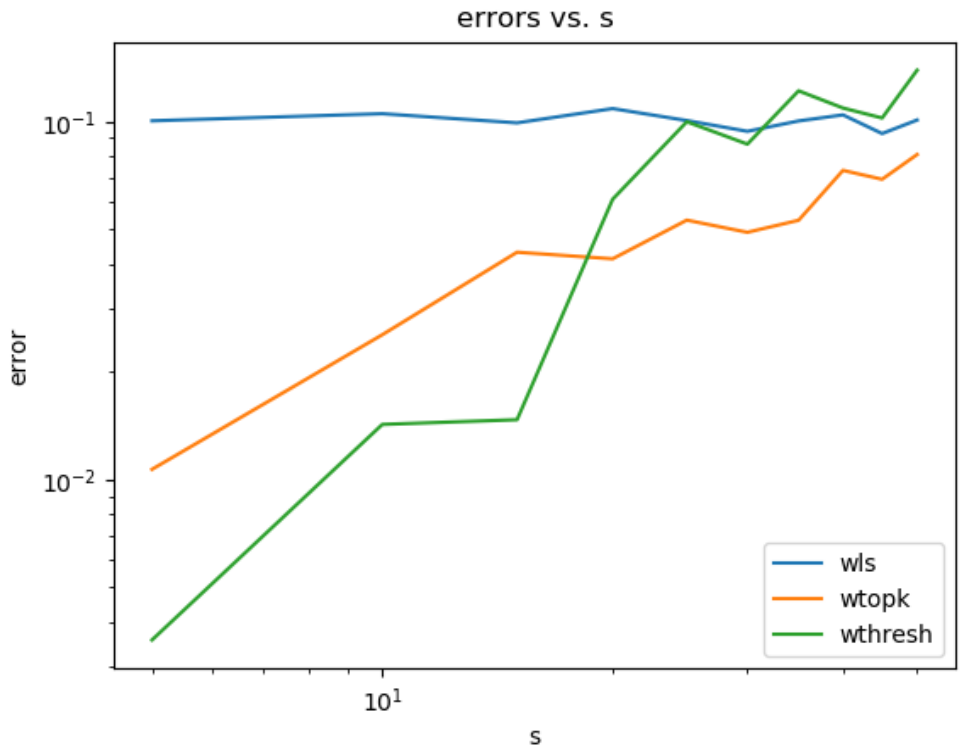
```
wls_error = []
wtopk_error = []
wthresh_error = []

if param == 'n':
arg_range = np.arange(100, 2000, 50)
lmbda = 2 * np.sqrt(np.log(d))
for n in arg_range:
U, wstar = ground_truth(n, d, s) if s_model else ground_truth(n, d, true_s)
error_wls = 0
error_wtopk = 0
error_wthresh = 0
for count in range(num_iters):
y = get_obs(U, wstar)
wls = LS(U, y)
wtopk = topk(U, y, s)
wthresh = thresh(U, y, lmbda)
error_wls += np.linalg.norm(wstar - wls) ** 2
error_wtopk += np.linalg.norm(wstar - wtopk) ** 2
error_wthresh += np.linalg.norm(wstar - wthresh) ** 2
wls_error.append(float(error_wls) / n / num_iters)
wtopk_error.append(float(error_wtopk) / n / num_iters)
wthresh_error.append(float(error_wthresh) / n / num_iters)

elif param == 'd':
arg_range = np.arange(10, 1000, 50)
for d in arg_range:
lmbda = 2 * np.sqrt(np.log(d))
U, wstar = ground_truth(n, d, s) if s_model else ground_truth(n, d, true_s)
error_wls = 0
error_wtopk = 0
error_wthresh = 0
for count in range(num_iters):
y = get_obs(U, wstar)
wls = LS(U, y)
wtopk = topk(U, y, s)
wthresh = thresh(U, y, lmbda)
error_wls += np.linalg.norm(wstar - wls) ** 2
error_wtopk += np.linalg.norm(wstar - wtopk) ** 2
error_wthresh += np.linalg.norm(wstar - wthresh) ** 2
wls_error.append(float(error_wls) / n / num_iters)
wtopk_error.append(float(error_wtopk) / n / num_iters)
wthresh_error.append(float(error_wthresh) / n / num_iters)

elif param == 's':
arg_range = np.arange(5, 55, 5)
lmbda = 2 * np.sqrt(np.log(d))
for s in arg_range:
U, wstar = ground_truth(n, d, s) if s_model else ground_truth(n, d, true_s)
error_wls = 0
error_wtopk = 0
error_wthresh = 0
for count in range(num_iters):
y = get_obs(U, wstar)
wls = LS(U, y)
wtopk = topk(U, y, s)
wthresh = thresh(U, y, lmbda)
error_wls += np.linalg.norm(wstar - wls) ** 2
error_wtopk += np.linalg.norm(wstar - wtopk) ** 2
error_wthresh += np.linalg.norm(wstar - wthresh) ** 2
wls_error.append(float(error_wls) / n / num_iters)
wtopk_error.append(float(error_wtopk) / n / num_iters)
wthresh_error.append(float(error_wthresh) / n / num_iters)
```

```
return arg_range, wls_error, wtopk_error, wthresh_error


# nrange contains the range of n used, ls_error the corresponding errors for the OLS estimate
# nrange, ls_error, _, _ = error_calc(num_iters=10, param='n', n=1000, d=100, s=5, s_model=True, true_s=5)


########
# TODO: Your code here: call the helper function for d and s, and plot everything
########
#YOUR CODE HERE:
all_paras = ['n', 'd', 's']
for para in all_paras:
arg_range, wls_error, _, _ = error_calc(num_iters=10, param=para, n=1000, d=100, s=5, s_model=True,
    true_s=5)
plt.semilogy(arg_range, wls_error, label='wls')
plt.title('errors for the OLS estimate vs. ' + para)
plt.xlabel(para)
plt.ylabel('wls_error')
plt.legend()
plt.savefig('Figure_3a-' + para + '.png')
plt.close()

# TODO: Part (b)
###############
#YOUR CODE HERE:
all_paras = ['n', 'd', 's']
for para in all_paras:
arg_range, wls_error, wtopk_error, wthresh_error = error_calc(num_iters=10, param=para, n=1000, d=100,
    s=5, s_model=True, true_s=5)
plt.semilogy(arg_range, wls_error, label='wls')
plt.semilogy(arg_range, wtopk_error, label='wtopk')
plt.semilogy(arg_range, wthresh_error, label='wthresh')
plt.title('errors vs. ' + para)
plt.xlabel(para)
plt.ylabel('error')
plt.legend()
plt.savefig('Figure_3b-' + para + '.png')
plt.close()

# TODO: Part (c)
###############
#YOUR CODE HERE:
all_paras = ['n', 'd', 's']
for para in all_paras:
arg_range, wls_error, wtopk_error, wthresh_error = error_calc(num_iters=10, param=para, n=1000, d=100,
    s=5, s_model=False, true_s=100)
plt.semilogy(arg_range, wls_error, label='wls')
plt.semilogy(arg_range, wtopk_error, label='wtopk')
plt.semilogy(arg_range, wthresh_error, label='wthresh')
plt.title('errors vs. ' + para)
plt.xlabel(para)
plt.ylabel('error')
plt.legend()
plt.savefig('Figure_3c-' + para + '.png')
plt.close()
```

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



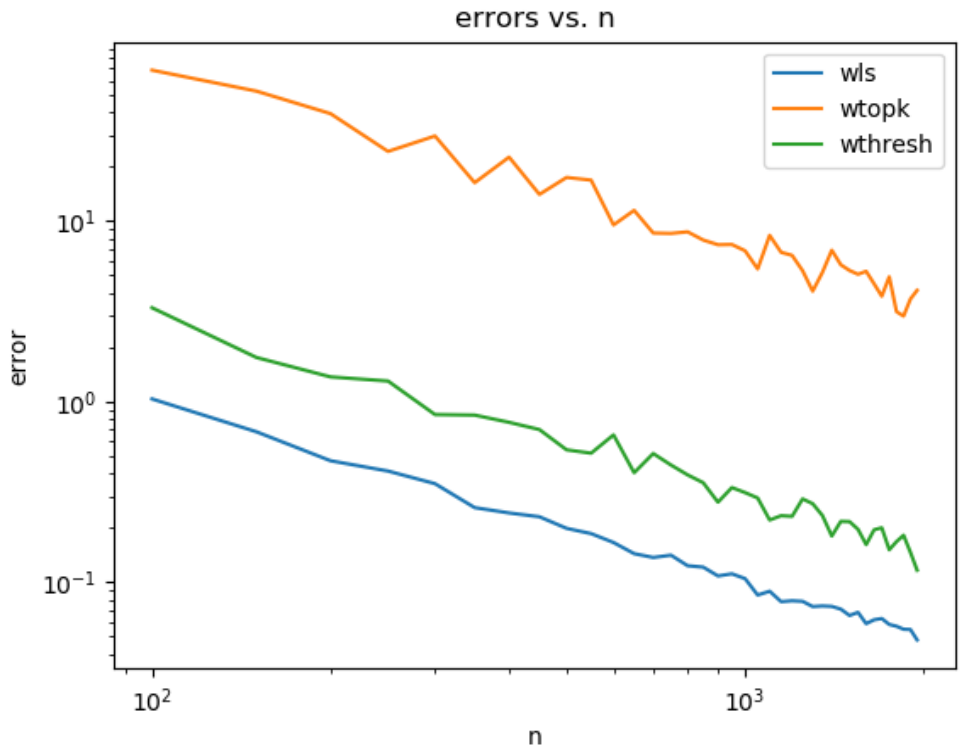



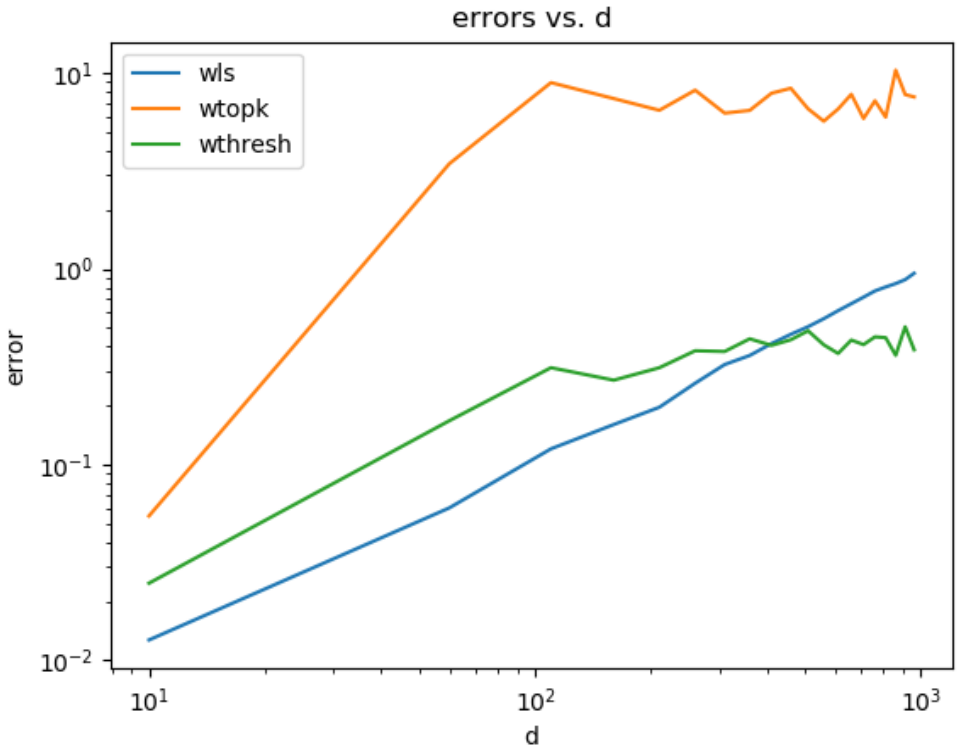


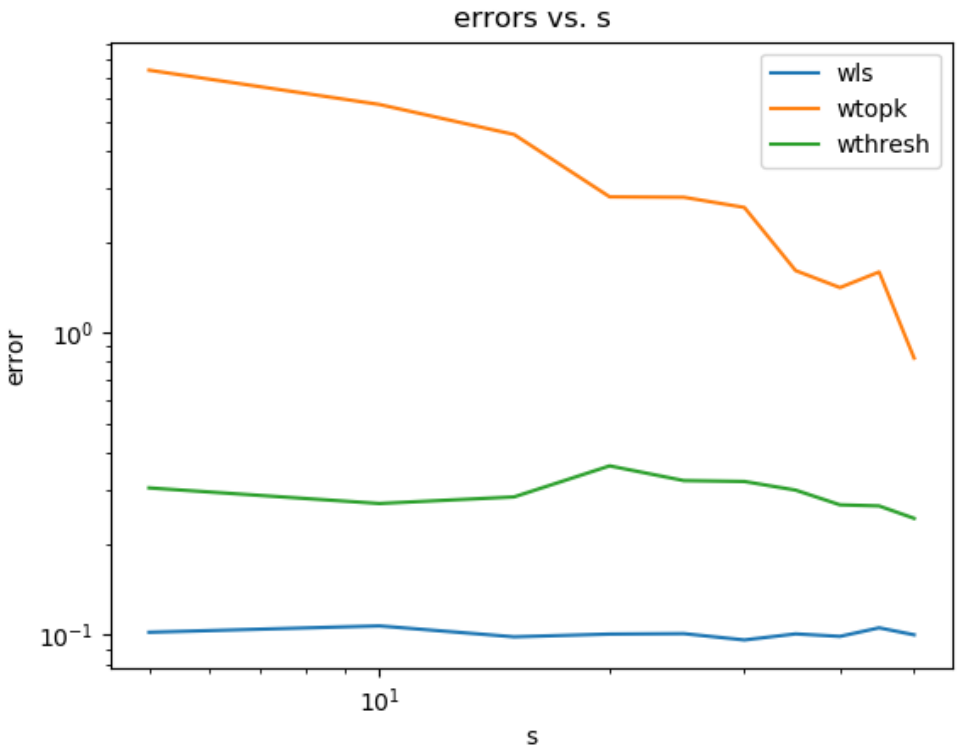

```
# TODO: Part (b)
###############
#YOUR CODE HERE:
all_paras = ['n', 'd', 's']
for para in all_paras:
arg_range, wls_error, wtopk_error, wthresh_error = error_calc(num_iters=10, param=para, n=1000, d=100,
    s=5, s_model=True, true_s=5)
plt.semilogy(arg_range, wls_error, label='wls')
plt.semilogy(arg_range, wtopk_error, label='wtopk')
plt.semilogy(arg_range, wthresh_error, label='wthresh')
plt.title('errors vs. ' + para)
plt.xlabel(para)
plt.ylabel('error')
plt.legend()
plt.savefig('Figure_3b-' + para + '.png')
plt.close()
```

(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.**

There is no bias-variance trade-off in $n$ because more data always gives you better estimates.
There is a bias-variance trade-off in $d$ just like higher polynomial degree increases the variance but decreases the bias. However, in the plot, we have already passed the optimal $d$ so the error keeps increasing.
In $s$, OLS estimator doesn't change because it is independent of $s$. there is a bias-variance trade-off in wtopk for the same reason as in $d$. However, we don't see it because $s > 50$.
All in all, if the original model is sparse, the more extra features we add in, the higher the variance would be because those extra features are used to fit the noise; however, if we don't have enough features, we wouldn't be able to fit the model so that bias would be high.






```
# TODO: Part (c)
###############
#YOUR CODE HERE:
all_paras = ['n', 'd', 's']
for para in all_paras:
```

```
arg_range, wls_error, wtopk_error, wthresh_error = error_calc(num_iters=10, param=para, n=1000, d=100,
    s=5, s_model=False, true_s=100)
plt.semilogy(arg_range, wls_error, label='wls')
plt.semilogy(arg_range, wtopk_error, label='wtopk')
plt.semilogy(arg_range, wthresh_error, label='wthresh')
plt.title('errors vs. ' + para)
plt.xlabel(para)
plt.ylabel('error')
plt.legend()
plt.savefig('Figure_3c-' + para + '.png')
plt.close()
```

**(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 $\sigma^2$, 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.**

$$\begin{aligned}
&\Pr\left\{\max_{i\in\{1,2,\ldots,d\}} |Z_i| \geq 2\sigma\sqrt{\log d}\right\} \\
=&\Pr\left\{\exists i, |Z_i| \geq 2\sigma\sqrt{\log d}\right\} \\
=&\cup_{i\in\{1,2,\ldots,d\}} \Pr\left\{|Z_i| \geq 2\sigma\sqrt{\log d}\right\} \\
\leq&\sum_{i=1}^{d} \Pr\left\{|Z_i| \geq 2\sigma\sqrt{\log d}\right\} \\
\leq&\sum_{i=1}^{d} e^{-\frac{(2\sigma\sqrt{\log d})^2}{2\sigma^2}} \\
=&de^{-\frac{4\sigma^2\log d}{2\sigma^2}} \\
=&de^{-2\log d} \\
=&\frac{d}{d^2} \\
=&\frac{1}{d}
\end{aligned}$$

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

$$
\begin{aligned}
&\widehat{w}_{\mathsf{LS}} \\
=&\mathbf{X}^\dagger y \\
=&\mathbf{X}^\top y \\
=&\mathbf{X}^\top(\mathbf{X}w^* + z) \\
=&\mathbf{X}^\top\mathbf{X}w^* + \mathbf{X}^\top z \\
=&w^* + \mathbf{X}^\top z \\
&\text{Denote } z' = \mathbf{X}^\top z, \\
&\text{we know that } \mathbb{E}[z'z'^\top] = \mathbf{X}^\top\mathbb{E}[zz^\top]\mathbf{X} = \sigma^2\mathbb{I} \\
=&w^* + z'
\end{aligned}
$$

$$
\begin{aligned}
&\widehat{w}_{\mathsf{top}}(s) \\
=&\tau_s(\widehat{w}_{\mathsf{LS}}) \\
=&\tau_s(w^* + z')
\end{aligned}
$$

Here $\tau_s(\cdot)$ means the top $s$ entries of the vector in absolute values. Therefore, we have shown that $\widehat{w}_{\mathsf{top}}(s)$ returns the top $s$ entries of the vector $w^* + z'$ in absolute value.

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

$$
\begin{aligned}
& e \\
=& \widehat{w}_{\mathsf{top}}(s) - w^* \\
=& \tau_s(w^* + z') - w^* \\
& \text{Without losing generality, we assume entries of } w^* + z' \text{ are sorted} \\
=& \begin{bmatrix} z'_1 \\ z'_2 \\ \vdots \\ z'_s \\ -w^*_{s+1} \\ \vdots \\ -w^*_d \end{bmatrix}
\end{aligned}
$$

As we can see $e$ is at least $s$-sparse, because the entries of $z'$ has zero probability to be zero. Because we know $w^*$ is $s$-sparse, the worst case scenario is that there exists $s$ nonzero entries between $w^*_{s+1}$ and $w^*_d$. That is, in the worst case scenario, $e$ is always 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 $i$.**

We still assume that $\widehat{w}_{\mathsf{top}}(s)$ is sorted.

$$e = \begin{bmatrix} z_1' \\ z_2' \\ \vdots \\ z_s' \\ -w_{s+1}^* \\ \vdots \\ -w_d^* \end{bmatrix} \quad \forall i \quad |w_i^* + z_i'| < |w_{i-1}^* + z_{i-1}'|$$

We know the first $s$ entries are bounded by $2\sigma\sqrt{\log d}$. We only need to look at the $m$ entry assuming that $|w_m^*| > |w_n^*|, \forall n \in s+1, \ldots, d$.
If $w_m^* \neq 0$, because $w^*$ is $s$-sparse there must be a $w_k^* = 0$ where $k \leq s$. Therefore we have:

$$\begin{aligned} |w_m^* + z_m'| &< |w_k^* + z_k'| = |z_k'| \\ |w_m^* + z_m'| &< |z_k'| \\ |w_m^*| - |z_m'| &< |z_k'| \\ |w_m^*| &< |z_k'| + |z_m'| \\ |w_m^*| &< 4\sigma\sqrt{\log d} \end{aligned}$$

As a summary, we have proved that $\forall i, \quad |e_i| \leq 4\sigma\sqrt{\log d}$.

**(h) Conclude that with probability at least $1 - 1/d$, we have**

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

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

Therefore, we have proved the probability that the conditions of the previous part is true is $1 - \frac{1}{d}$. Now we just need to show that the previous conditions lead to the inequality we want to prove.

$$\begin{aligned}
&\|\widehat{w}_{\mathsf{top}}(s) - w^*\|_2^2 \\
=&\|e\|_2^2 \\
=&\sum_{i=1}^{d} e_i^2 \\
\leq&\sum_{i=1}^{d} (4\sigma\sqrt{\log d})^2 \\
&\text{because } e \text{ is at most } 2s\text{-sparse, we only need to sum } 2s \text{ upper bounds} \\
\leq&2s \times 16\sigma^2 \log d \\
=&32\sigma^2 s \log d
\end{aligned}$$

Equation (4) is copied here:

$$\mathbb{E}\left[\|\widehat{w}_{\mathsf{LS}} - w^*\|_2^2\right] = \sigma^2 d$$

We want:

$$\begin{aligned}
32\sigma^2 s \log d &< \sigma^2 d \\
32 s \log d &< d \\
\frac{\log d}{d} &< 32s
\end{aligned}$$

The LHS increases monotonically with $d$ so once the above inequality is true, sparse model performs better.

**(i) Use the above part to show that with probability at least $1 - 1/d$, we have**

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

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

$$\begin{aligned}
&\frac{1}{n}\|\mathbf{X}(\widehat{w}_{\mathsf{top}}(s) - w^*)\|_2^2 \\
=&\frac{1}{n}(\widehat{w}_{\mathsf{top}}(s) - w^*)^\top \mathbf{X}^\top \mathbf{X}(\widehat{w}_{\mathsf{top}}(s) - w^*) \\
&\text{Recall that } \mathbf{X} \text{ has orthonormal columns} \\
=&\frac{1}{n}(\widehat{w}_{\mathsf{top}}(s) - w^*)^\top \mathbb{I}(\widehat{w}_{\mathsf{top}}(s) - w^*) \\
=&\frac{1}{n}(\widehat{w}_{\mathsf{top}}(s) - w^*)^\top (\widehat{w}_{\mathsf{top}}(s) - w^*) \\
=&\frac{1}{n}\|\widehat{w}_{\mathsf{top}}(s) - w^*\|_2^2 \\
\leq&32\sigma^2 \frac{s \log d}{n}
\end{aligned}$$

We know the probability of the last inequality to be true is $1 - \frac{1}{d}$ so the probability of the inequality is true is $1 - \frac{1}{d}$.

**(j) 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.)**

**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.**

Copy from above:

$$\mathbb{E}\left[\frac{1}{n}\|\mathbf{X}(\widehat{w}_{\mathsf{LS}} - w^*)\|_2^2\right] = \sigma^2 \frac{d}{n}$$

We notice that the error is the same as variance in this case because the true $w^*$ is $s$-sparse so both $\widehat{w}_{\mathsf{top}}(s)$ and $\widehat{w}_{\mathsf{LS}}$ can predict the true $w^*$ with no bias.
We want:

$$\begin{aligned}
\frac{1}{n}\|\mathbf{X}(\widehat{w}_{\mathsf{top}}(s) - w^*)\|_2^2 &\leq \mathbb{E}\left[\frac{1}{n}\|\mathbf{X}(\widehat{w}_{\mathsf{LS}} - w^*)\|_2^2\right] \\
32\sigma^2 \frac{s \log d}{n} &\leq \sigma^2 \frac{d}{n} \\
s &\leq \frac{d}{32 \log d}
\end{aligned}$$

**(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 using the $s$ important features? How does this variance compare to the variance bound for the sparse $\widehat{w}_{\mathsf{top}}(s)$ derived above?**

Recall in part (g), we had $\forall i, \quad |e_i| \leq 4\sigma\sqrt{\log d}$. The bound can be shrunk to be $\forall i, \quad |e_i| \leq 2\sigma\sqrt{\log d}$ if we now the most important $s$ features.
We also know that $e$ would be $s$-sparse instead of $2s$-sparse. Thus, in part (h), we would have $\|\widehat{w}_{\mathsf{top}}(s) - w^*\|_2^2 \leq 4\sigma^2 s \log d$
Finally, we can write the new bound of the variance of the sparse OLS estimator above:

$$\frac{1}{n}\|\mathbf{X}(\widehat{w}_{\mathsf{top}}(s) - w^*)\|_2^2 \leq 4\sigma^2 \frac{s \log d}{n}$$

This variance is much smaller than the variance bound for the sparse $\widehat{w}_{\mathsf{top}}(s)$ derived above.

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

For $\widehat{w}_T(\lambda)$, the previous parts will change.
For part (g), we would have the same bound $\forall i, \quad |e_i| \leq 4\sigma\sqrt{\log d}$. This is because:

$$\begin{aligned} |w_i^* + z_i'| &< 2\sigma\sqrt{\log d} \\ |w_i^*| - |z_i'| &< 2\sigma\sqrt{\log d} \\ |w_i^*| &< 4\sigma\sqrt{\log d} \end{aligned}$$

However, $e$ would be $s$-sparse. Let's look at $\widehat{w}_{LS}$ (all entries are also assumed to be sorted and we know which $s$ features to keep):

$$\widehat{w}_{LS} = \begin{bmatrix} w_1^* + z_1' \\ w_2^* + z_2' \\ \vdots \\ w_s^* + z_s' \\ w_{s+1}^* + z_{s+1}' \\ \vdots \\ w_d^* + z_d' \end{bmatrix} = \begin{bmatrix} w_1^* + z_1' \\ w_2^* + z_2' \\ \vdots \\ w_s^* + z_s' \\ z_{s+1}' \\ \vdots \\ z_d' \end{bmatrix} \quad \forall i \quad |w_i^* + z_i'| < |w_{i-1}^* + z_{i-1}'|$$

Because $w^*$ is $s$-sparse, none of the noisy zero entry of $w^*$ would be kept given that $\forall i, |z_i'| \leq 2\sigma\sqrt{\log d}$, we would only keep at most $s$ entries.

$$\widehat{w}_T(\lambda) = \begin{bmatrix} w_1^* + z_1' \\ w_2^* + z_2' \\ \vdots \\ w_s^* + z_s' \\ 0 \\ \vdots \\ 0 \end{bmatrix} \quad \forall i \quad |w_i^* + z_i'| < |w_{i-1}^* + z_{i-1}'|$$

Therefore, we get the bound:

$$\frac{1}{n}\|\mathbf{X}(\widehat{w}_{\mathsf{top}}(s) - w^*)\|_2^2 \leq 16\sigma^2 \frac{s \log d}{n}$$

**Question 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

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) **Implement the information gain and gini impurity splitting rules 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 code for information gain and gini impurity.**

**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.

```
@staticmethod
def information_gain(X, y, thresh):
# TODO implement information gain function
class_labels = np.unique(y)
y0, y1 = y[X < thresh], y[X >= thresh]
n0 = len(y0)
n1 = len(y1)
n = len(y)
entropy = 0.0
for label in class_labels:
if np.count_nonzero(y == label) > 0:
entropy -= np.count_nonzero(y == label) * np.log2(np.count_nonzero(y == label) / n)
if np.count_nonzero(y0 == label) > 0:
entropy += np.count_nonzero(y0 == label) * np.log2(np.count_nonzero(y0 == label) / n0)
if np.count_nonzero(y1 == label) > 0:
entropy += np.count_nonzero(y1 == label) * np.log2(np.count_nonzero(y1 == label) / n1)
entropy /= n
return entropy

@staticmethod
def gini_impurity(X, y, thresh):
# TODO implement gini_impurity function
class_labels = np.unique(y)
y0, y1 = y[X < thresh], y[X >= thresh]
n0 = len(y0)
n1 = len(y1)
n = len(y)
gini = n0 + n1
for label in class_labels:
gini -= np.count_nonzero(y0 == label) ** 2 / n0
gini -= np.count_nonzero(y1 == label) ** 2 / n1
gini /= n
return -gini
```

(b) Before applying the decision tree to the Titanic dataset, described below, we will first preprocess the dataset. We provide the code to preprocess the data. Describe how we deal with the following problems:

- Some data miss class labels;
- Some features are not numerical values;
- Some data miss features.

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

1. survived - 1 is survived; 0 is not. This is the class label.
2. pclass - Measure of socioeconomic status: 1 is upper, 2 is middle, 3 is lower.
3. sex - Male/Female
4. age - Fractional if less than 1.
5. sibsp - Number of siblings/spouses aboard the Titanic
6. parch - Number of parents/children aboard the Titanic
7. ticket - Ticket number
8. fare - Fare.
9. cabin - Cabin number.
10. embarked - Port of Embarkation (C = Cherbourg, Q = Queenstown, S = Southampton)

Describe how you deal with the following problems:

- Some data miss class labels;
  You throw out those data without class labels:

```
labeled_idx = np.where(y != b'')[0]
y = np.array(y[labeled_idx], dtype=np.int)
```

- Some features are not numerical values;
  You convert all values to float. Then you pick the string features which appear more than **min_freq** times in the dataset and set those data points with the features to one and others to zeros. You disregard all non-numerical features that appear less than **min_freq** times

```
# Hash the columns (used for handling strings)
onehot_encoding = []
for col in onehot_cols:
counter = Counter(data[:, col])
for term in counter.most_common():
if term[-1] <= min_freq:
break
onehot_encoding.append((data[:, col] == term[0]).astype(np.float))
data[:, col] = '0'
onehot_encoding = np.array(onehot_encoding).T
data = np.hstack([np.array(data, dtype=np.float), np.array(onehot_encoding)])
```

- Some data miss features.
  You first temporarily assigned $-1$ to all missing features and then assign the mode to those data.

```
# Temporarily assign -1 to missing data
data[data == b''] = '-1'
...
# Replace missing data with the mode value. We use the mode instead of
# the mean or median because this makes more sense for categorical
# features such as gender or cabin type, which are not ordered.
if fill_mode:
for i in range(data.shape[-1]):
mode = stats.mode(data[((data[:, i] < -1 - eps) +
(data[:, i] > -1 + eps))][:, i]).mode[0]
data[(data[:, i] > -1 - eps) * (data[:, i] < -1 + eps)][:, i] = mode
```

(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.

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

Please find the tree below: Part (b): preprocessing the titanic dataset
Features: [b'pclass', b'sex', b'age', b'sibsp', b'parch', b'ticket', b'fare', b'cabin', b'embarked']
Train/test size: (999, 9) (310, 9)

Part 0: constant classifier
Accuracy

$$[0.6136136136136137]$$

simplified decision tree
Accuracy

$$[0.7967967967967968]$$

Part (a-b): simplified decision tree
Predictions

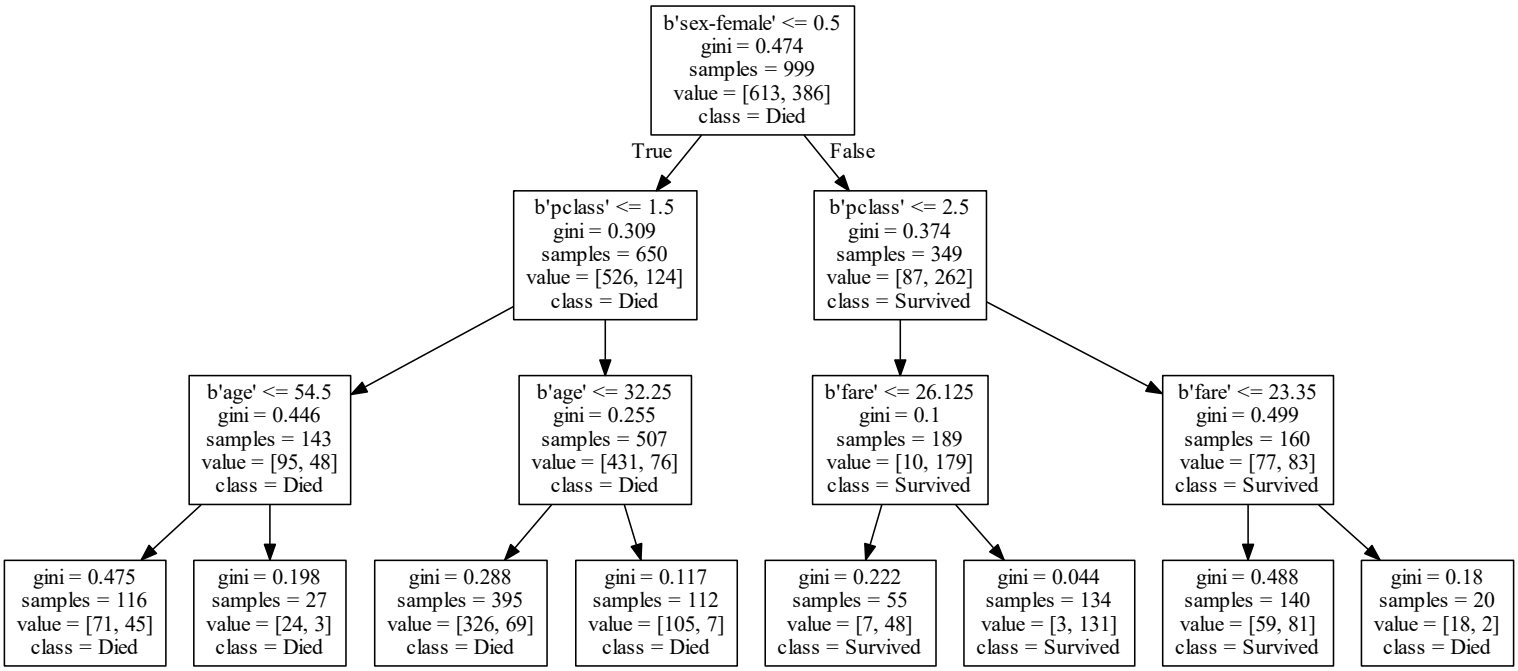
$$\begin{bmatrix}
1. & 1. & 0. & 0. & 0. & 1. & 0. & 0. & 0. & 0. & 1. & 1. & 1. & 1. & 1. & 1. & 1. & 0. \\
1. & 0. & 0. & 0. & 1. & 1. & 0. & 0. & 1. & 1. & 1. & 0. & 0. & 0. & 0. & 0. & 0. & 0. \\
0. & 0. & 0. & 0. & 0. & 1. & 0. & 0. & 1. & 0. & 0. & 0. & 1. & 0. & 0. & 0. & 0. & 0. \\
1. & 1. & 0. & 1. & 0. & 0. & 0. & 1. & 0. & 0. & 1. & 1. & 0. & 1. & 1. & 0. & 1. & 0. \\
1. & 0. & 0. & 0. & 0. & 0. & 0. & 1. & 0. & 0. & 1. & 0. & 0. & 0. & 0. & 0. & 1. & 0. \\
1. & 0. & 0. & 1. & 0. & 0. & 1. & 0. & 0. & 0. & & & & & & & & \\
\end{bmatrix}$$

---

```
def preprocess(data, fill_mode=True, min_freq=10, onehot_cols=[]):
# fill_mode = False

# Temporarily assign -1 to missing data
data[data == b''] = '-1'

# Hash the columns (used for handling strings)
onehot_encoding = []
onehot_features = []
for col in onehot_cols:
counter = Counter(data[:, col])
for term in counter.most_common():
if term[0] == b'-1':
continue
if term[-1] <= min_freq:
break
onehot_features.append(features[col] + b'-' + term[0])
```

```
onehot_encoding.append((data[:, col] == term[0]).astype(np.float))
data[:, col] = '0'
onehot_encoding = np.array(onehot_encoding).T
data = np.hstack([np.array(data, dtype=np.float), np.array(onehot_encoding)])

# Replace missing data with the mode value. We use the mode instead of
# the mean or median because this makes more sense for categorical
# features such as gender or cabin type, which are not ordered.
if fill_mode:
for i in range(data.shape[-1]):
mode = stats.mode(data[((data[:, i] < -1 - eps) +
(data[:, i] > -1 + eps))][:, i]).mode[0]
data[(data[:, i] > -1 - eps) * (data[:, i] < -1 + eps)][:, i] = mode

return data, onehot_features


def evaluate(clf):
print("Cross validation", sklearn.model_selection.cross_val_score(clf, X, y))
if hasattr(clf, "decision_trees"):
# counter = Counter([t.tree_.feature[0] for t in clf.decision_trees])
# first_splits = [(features[term[0]], term[1]) for term in counter.most_common()]
# print("First splits", first_splits)
roots = []
for _, t in enumerate(clf.decision_trees):
roots.append(str(t.tree_.feature[0]) + '-' + str(t.tree_.threshold[0]))
counter = Counter(roots)
first_splits = [(term[0], term[1]) for term in
counter.most_common()]
print('The most common splits at the root node of the tree are: ')
for i, this_split in enumerate(first_splits):
name, thold = this_split[0].split('-', 1)
name = features[int(name)]
if isinstance(name, bytes):
name = str(name, "utf-8")
else:
name = str(name)
print(str(i + 1) + '. ("' + name + '")' + '<' + str(thold) + ' (' + str(this_split[1]) + ' trees)')


if __name__ == "__main__":
os.environ["PATH"] += os.pathsep + 'C:/Program Files (x86)/graphviz-2.38/bin/'
# dataset = "titanic"
# dataset = 'spam'
all_datasets = ['titanic', 'spam']
params = {
"max_depth": 5,
# "random_state": 6,
"min_samples_leaf": 10,
}
N = 100
for dataset in all_datasets:

if dataset == "titanic":
# Load titanic data
path_train = 'datasets/titanic/titanic_training.csv'
data = genfromtxt(path_train, delimiter=',', dtype=None)
path_test = 'datasets/titanic/titanic_testing_data.csv'
test_data = genfromtxt(path_test, delimiter=',', dtype=None)
y = data[1:, 0] # label = survived
class_names = ["Died", "Survived"]
features = list(data[0, 1:])
```

```
labeled_idx = np.where(y != b'')[0]
y = np.array(y[labeled_idx], dtype=np.int)
print("\n\nPart (b): preprocessing the titanic dataset")
X, onehot_features = preprocess(data[1:, 1:], onehot_cols=[1, 5, 7, 8])
X = X[labeled_idx, :]
Z, _ = preprocess(test_data[1:, :], onehot_cols=[1, 5, 7, 8])
assert X.shape[1] == Z.shape[1]
features = list(data[0, 1:]) + onehot_features

elif dataset == "spam":
features = [
"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",
"semicolon", "dollar", "sharp", "exclamation", "parenthesis", "square_bracket",
"ampersand"
]
assert len(features) == 32

# Load spam data
path_train = 'datasets/spam_data/spam_data.mat'
data = scipy.io.loadmat(path_train)
X = data['training_data']
y = np.squeeze(data['training_labels'])
Z = data['test_data']
class_names = ["Ham", "Spam"]

else:
raise NotImplementedError("Dataset %s not handled" % dataset)

print("Features:", features)
print("Train/test size:", X.shape, Z.shape)

print("\n\nPart 0: constant classifier")
print("Accuracy", 1 - np.sum(y) / y.size)

# Basic decision tree
print("\n\nsimplified decision tree")
dt = DecisionTree(max_depth=3, feature_labels=features)
dt.fit(X, y)
# print("Predictions", dt.predict(Z)[:100])
print("Accuracy", 1 - np.sum(abs(dt.predict(X) - y)) / y.size)

print("\n\nsklearn's decision tree")
this_params = params
this_params['max_depth'] = 3
clf = sklearn.tree.DecisionTreeClassifier(random_state=0, **this_params)
clf.fit(X, y)
evaluate(clf)
out = io.StringIO()
sklearn.tree.export_graphviz(
clf, out_file=out, feature_names=features, class_names=class_names)
graph = pydot.graph_from_dot_data(out.getvalue())
pydot.graph_from_dot_data(out.getvalue())[0].write_pdf("%s-tree.pdf" % dataset)
print("Accuracy", 1 - np.sum(abs(clf.predict(X) - y)) / y.size)
```


b'sex-female' <= 0.5
gini = 0.474
samples = 999
value = [613, 386]
class = Died
True
False
b'pclass' <= 1.5
gini = 0.309
samples = 650
value = [526, 124]
class = Died
b'pclass' <= 2.5
gini = 0.374
samples = 349
value = [87, 262]
class = Survived
b'age' <= 54.5
gini = 0.446
samples = 143
value = [95, 48]
class = Died
b'age' <= 32.25
gini = 0.255
samples = 507
value = [431, 76]
class = Died
b'fare' <= 26.125
gini = 0.1
samples = 189
value = [10, 179]
class = Survived
b'fare' <= 23.35
gini = 0.499
samples = 160
value = [77, 83]
class = Survived
gini = 0.475
samples = 116
value = [71, 45]
class = Died
gini = 0.198
samples = 27
value = [24, 3]
class = Died
gini = 0.288
samples = 395
value = [326, 69]
class = Died
gini = 0.117
samples = 112
value = [105, 7]
class = Died
gini = 0.222
samples = 55
value = [7, 48]
class = Survived
gini = 0.044
samples = 134
value = [3, 131]
class = Survived
gini = 0.488
samples = 140
value = [59, 81]
class = Survived
gini = 0.18
samples = 20
value = [18, 2]
class = Died

(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
```

---

```
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 = [
sklearn.tree.DecisionTreeClassifier(random_state=i, **self.params)
for i in range(self.n)
]

def fit(self, X, y):
# TODO implement function
for i in range(self.n):
idx = np.random.choice(len(y), len(y), replace=True)
X_rand = X[idx]
y_rand = y[idx]
self.decision_trees[i].fit(X_rand, y_rand)
return self
```

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

def print_common_split(self):
roots = []
for _, t in enumerate(self.decision_trees):
roots.append(str(t.tree_.feature[0]) + '-' + str(t.tree_.threshold[0]))
counter = Counter(roots)
first_splits = [(term[0], term[1]) for term in
counter.most_common()]
print('The most common splits at the root node of the tree are: ')
for i, this_split in enumerate(first_splits):
name, thold = this_split[0].split('-', 1)
name = features[int(name)]
print(str(i + 1) + '. ("' + str(name, "utf-8") + '")' + '<' + str(thold) + ' (' + str(this_split[1]) + '
    trees)')
```

(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:

1. ("viagra") $\geq 3$ (20 trees)
2. ("thanks") $< 4$ (15 trees)
3. ("nigeria") $\geq 1$ (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.

BaggedTrees on titanic
Cross validation

$$\begin{bmatrix}0.79341317 & 0.77477477 & 0.81927711\end{bmatrix}$$

The most common splits at the root node of the tree are:
1. ("sex-male")<0.5 (52 trees)
2. ("sex-female")<0.5 (48 trees)
(It should be noted that the above two are equivalent because we don't have the third gender.)
Accuracy

$$\begin{bmatrix}0.8048048048048049\end{bmatrix}$$

---

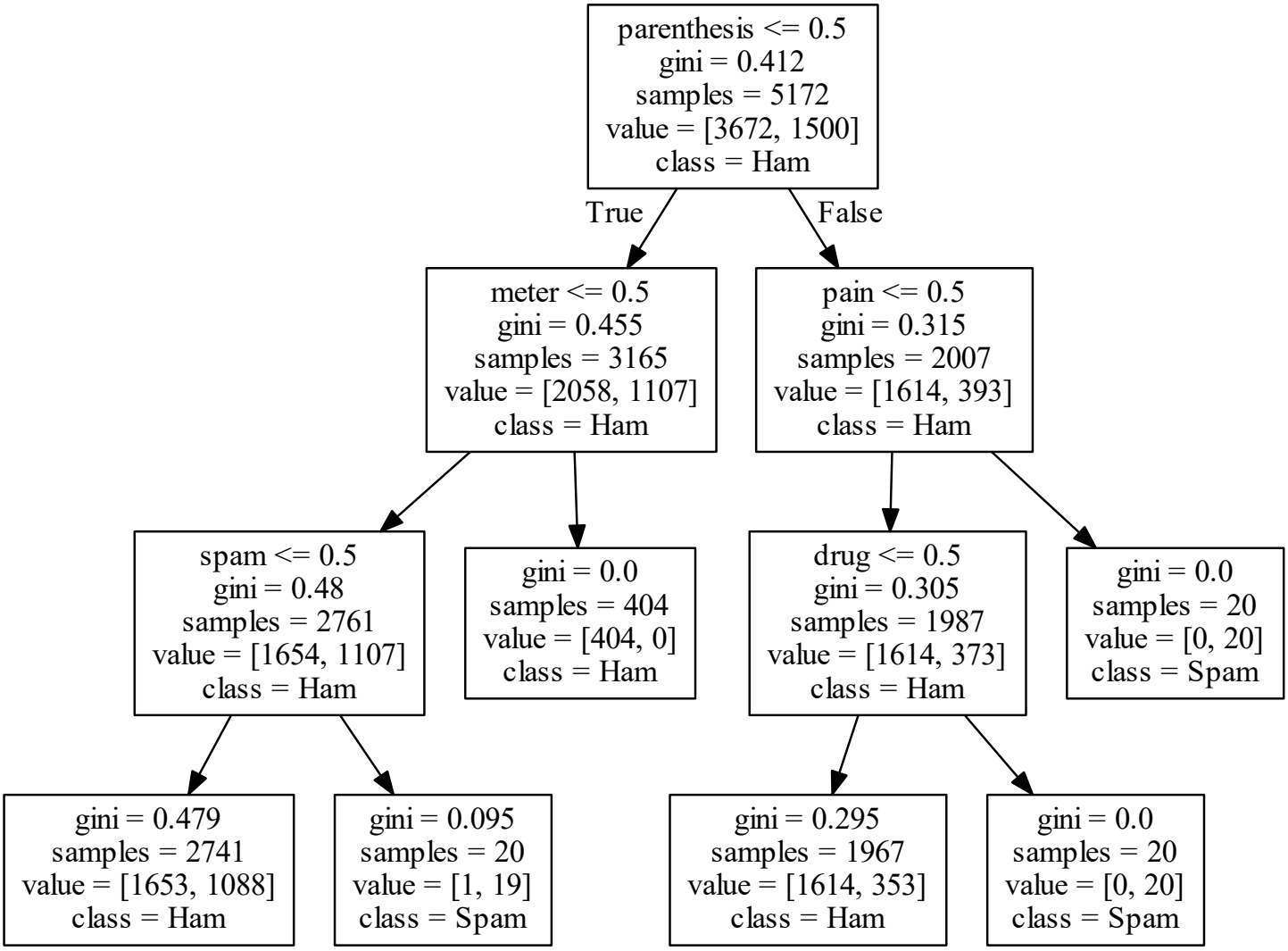
BaggedTrees on spam
Cross validation

$$\begin{bmatrix}0.79756381 & 0.80742459 & 0.77958237\end{bmatrix}$$

The most common splits at the root node of the tree are:
1. ("exclamation")<0.5 (100 trees)
Accuracy

$$\begin{bmatrix}0.7948569218870842\end{bmatrix}$$

```
# Part e
print('\n\nBaggedTrees on ' + dataset)
bt = BaggedTrees(params, n=N)
bt.fit(X, y)
evaluate(bt)
print("Accuracy", 1 - np.sum(abs(bt.predict(X) - y)) / y.size)
```


parenthesis <= 0.5
gini = 0.412
samples = 5172
value = [3672, 1500]
class = Ham
True
False
meter <= 0.5
gini = 0.455
samples = 3165
value = [2058, 1107]
class = Ham
pain <= 0.5
gini = 0.315
samples = 2007
value = [1614, 393]
class = Ham
spam <= 0.5
gini = 0.48
samples = 2761
value = [1654, 1107]
class = Ham
gini = 0.0
samples = 404
value = [404, 0]
class = Ham
drug <= 0.5
gini = 0.305
samples = 1987
value = [1614, 373]
class = Ham
gini = 0.0
samples = 20
value = [0, 20]
class = Spam
gini = 0.479
samples = 2741
value = [1653, 1088]
class = Ham
gini = 0.095
samples = 20
value = [1, 19]
class = Spam
gini = 0.295
samples = 1967
value = [1614, 353]
class = Ham
gini = 0.0
samples = 20
value = [0, 20]
class = Spam

(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
```

```
class RandomForest(BaggedTrees):
def __init__(self, params=None, n=200, m=1):
if params is None:
params = {'max_features': m}
else:
params['max_features'] = m
# TODO implement function
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.**

RandomForest on titanic
Cross validation

$$\begin{bmatrix}0.79341317 & 0.77477477 & 0.82228916\end{bmatrix}$$

The most common splits at the root node of the tree are:
1. ("sex-male")<0.5 (27 trees)
2. ("sex-female")<0.5 (14 trees)
3. ("embarked-S")<0.5 (10 trees)
4. ("pclass")<2.5 (9 trees)
5. ("parch")<0.5 (9 trees)
6. ("pclass")<1.5 (4 trees)
7. ("age")<-0.416649997234 (3 trees)
8. ("fare")<51.9312515259 (3 trees)
9. ("fare")<15.1729001999 (3 trees)
10. ("embarked-C")<0.5 (3 trees)
11. ("fare")<15.6604499817 (2 trees)
12. ("sibsp")<3.5 (2 trees)
13. ("fare")<15.6646003723 (1 trees)
14. ("fare")<26.2666511536 (1 trees)
15. ("sibsp")<0.5 (1 trees)
16. ("embarked-Q")<0.5 (1 trees)
17. ("fare")<15.1457996368 (1 trees)
18. ("sibsp")<2.5 (1 trees)
19. ("age")<-0.166649997234 (1 trees)
20. ("fare")<9.49164962769 (1 trees)
21. ("fare")<8.83749961853 (1 trees)
22. ("fare")<15.6458501816 (1 trees)
23. ("fare")<75.7708511353 (1 trees)

Accuracy

$$\begin{bmatrix}0.7967967967967968\end{bmatrix}$$

---

RandomForest on spam
Cross validation

$$\begin{bmatrix}0.79756381 & 0.80742459 & 0.77958237\end{bmatrix}$$

The most common splits at the root node of the tree are:
1. ("money")<0.5 (15 trees)
2. ("meter")<0.5 (15 trees)
3. ("exclamation")<0.5 (13 trees)
4. ("pain")<0.5 (8 trees)

5. ("volumes")<0.5 (8 trees)
6. ("parenthesis")<0.5 (7 trees)
7. ("ampersand")<0.5 (6 trees)
8. ("featured")<0.5 (5 trees)
9. ("dollar")<0.5 (4 trees)
10. ("prescription")<0.5 (4 trees)
11. ("differ")<0.5 (3 trees)
12. ("spam")<0.5 (3 trees)
13. ("dollar")<1.5 (3 trees)
14. ("creative")<0.5 (2 trees)
15. ("sharp")<0.5 (1 trees)
16. ("sharp")<1.5 (1 trees)
17. ("width")<0.5 (1 trees)
18. ("business")<0.5 (1 trees)

Accuracy

$$\begin{bmatrix} 0.7594740912606341 \end{bmatrix}$$

```
# Part g
print('\n\nRandomForest on ' + dataset)
rf = RandomForest(params, n=N, m=np.int(np.sqrt(X.shape[1])))
rf.fit(X, y)
evaluate(rf)
print("Accuracy", 1 - np.sum(abs(rf.predict(X) - y)) / y.size)
```

(h) Implement the AdaBoost 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:

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

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 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
```

**Describe qualitatively what this algorithm is doing.**
This algorithm is trying to do something like adaptive random forest. It tries to make sure new random trees are used to work out those uncertain data points of the previous random trees. If a tree has a higher accuracy, we believe it's closer to the real decision tree by giving it a higher weight.
**What does it mean when $a_i < 0$?**
It means at least half of the predictions are wrong because we have:

$$\begin{aligned} a_j &< 0 \\ \frac{1}{2}\log\frac{1-e_j}{e_j} &< 0 \\ \frac{1-e_j}{e_j} &< 1 \\ \frac{1}{e_j} &< 2 \\ e_j &> \frac{1}{2} \\ \frac{\sum_{i=1}^{N} I_j(x_i)w_i}{\sum_{i=1}^{N} w_i} &> \frac{1}{2} \end{aligned}$$

**How does the algorithm handle such trees?**
The algorithm gives a negative confidence of the predictions of such trees. So that it favors the other class labels.

---

```
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
for j in range(self.n):
idx = np.random.choice(len(y), len(y), replace=True, p=self.w)
X_rand = X[idx]
y_rand = y[idx]
self.decision_trees[j].fit(X_rand, y_rand)
yhat = self.decision_trees[j].predict(X)
Ihat = np.array(y != yhat, dtype=np.float)
ej = sum(self.w * Ihat) / sum(self.w)
self.a[j] = 1 / 2 * np.log2((1 - ej) / ej)
self.w[Ihat == 1] *= np.exp(self.a[j])
self.w[Ihat != 1] *= np.exp(-self.a[j])
self.w /= sum(self.w)
self.a /= sum(self.a)
return self

def predict(self, X):
# TODO implement function
ypred = []
yhat = []
for j in range(self.n):
yhat.append(self.decision_trees[j].predict(X))
yhat = np.vstack(yhat)
for i in range(X.shape[0]):
class_labels = np.unique(yhat[:, i])
yscores = []
for label in class_labels:
yscores.append(self.a.dot(yhat[:, i] == label))
```

```
ypred.append(yhat[np.argmax(yscores, axis=0), i])
return ypred
```

(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.**

We can use the final probabilities of data points "self.w" to determine which data is more challenging. The higher the probability, the more challenging is the data.
I just use "numpy.argsort" to get the sorted indices of "self.w" and then extract the features and labels from the original data matrix.
By examining the top 10 easiest samples to classify, I found that they all have many features and many of them have key features like 'volumes', 'meter', 'revision'.
By examining the top 10 hardest samples to classify, I found that they all have very few features and many of them have key features like 'parenthesis', 'semicolon', and 'ampersand'.

**The top 10 easiest samples to classify**
1. Ham: [('volumes', 1.0), ('revision', 1.0), ('meter', 1.0), ('dollar', 1.0), ('sharp', 5.0), ('exclamation', 1.0)]
2. Ham: [('volumes', 1.0), ('revision', 2.0), ('meter', 1.0), ('planning', 1.0), ('sharp', 1.0), ('parenthesis', 3.0)]
3. Ham: [('volumes', 1.0), ('revision', 2.0), ('meter', 1.0), ('sharp', 1.0), ('parenthesis', 2.0)]
4. Ham: [('other', 1.0), ('energy', 1.0), ('volumes', 1.0), ('meter', 8.0), ('parenthesis', 1.0), ('ampersand', 2.0)]
5. Ham: [('other', 1.0), ('energy', 1.0), ('volumes', 1.0), ('meter', 7.0), ('parenthesis', 1.0), ('ampersand', 2.0)]
6. Ham: [('energy', 1.0), ('volumes', 1.0), ('meter', 2.0), ('dollar', 2.0), ('sharp', 3.0)]
7. Ham: [('energy', 1.0), ('volumes', 1.0), ('meter', 1.0), ('dollar', 1.0), ('sharp', 2.0)]
8. Ham: [('other', 1.0), ('energy', 1.0), ('volumes', 1.0), ('meter', 8.0), ('parenthesis', 1.0), ('ampersand', 2.0)]
9. Ham: [('other', 1.0), ('energy', 1.0), ('volumes', 1.0), ('meter', 8.0), ('parenthesis', 1.0), ('ampersand', 2.0)]
10. Ham: [('message', 3.0), ('volumes', 1.0), ('meter', 5.0), ('sharp', 4.0), ('exclamation', 3.0), ('parenthesis', 5.0)]

**The top 10 hardest samples to classify**
1. Ham: [('volumes', 1.0), ('revision', 1.0), ('meter', 1.0), ('dollar', 1.0), ('sharp', 5.0), ('exclamation', 1.0)]
2. Spam: [('semicolon', 1.0), ('parenthesis', 1.0), ('ampersand', 1.0)]
3. Spam: [('parenthesis', 1.0), ('ampersand', 1.0)]
4. Spam: [('out', 1.0), ('semicolon', 6.0), ('parenthesis', 3.0), ('square_bracket', 1.0), ('ampersand', 2.0)]
5. Spam: [('sharp', 2.0), ('parenthesis', 5.0)]
6. Ham: [('money', 1.0), ('dollar', 12.0), ('sharp', 1.0), ('exclamation', 16.0), ('parenthesis', 5.0)]
7. Spam: [('semicolon', 1.0), ('sharp', 1.0), ('ampersand', 1.0)]
8. Spam: [('sharp', 1.0), ('ampersand', 2.0)]
9. Spam: [('sharp', 1.0), ('parenthesis', 1.0)]

10. Spam: [('other', 1.0), ('message', 1.0), ('sharp', 1.0), ('parenthesis', 1.0)]

```
# Part g
print('\n\nBoostedRandomForest on ' + dataset)
brf = BoostedRandomForest(params, n=N, m=np.int(np.sqrt(X.shape[1])))
brf.fit(X, y)
evaluate(brf)
print("Accuracy", 1 - np.sum(abs(brf.predict(X) - y)) / y.size)
if dataset == 'spam':
sorted_indices = np.argsort(brf.w)
print_n = 10
print_data = np.zeros((print_n, len(features) + 1), dtype=object)
print_features = features.copy()
print_features.insert(0, 'label')

print('\\textbf{{The top ' + str(print_n) + ' easiest samples to classify}}')
for ip in range(print_n):
print(str(ip+1) + '. ' + str(class_names[y[sorted_indices[ip]]]) + ': ' +
str([x for x in zip(features, X[sorted_indices[ip], :]) if x[1] > 0]))
print()
print('\\textbf{{The top ' + str(print_n) + ' hardest samples to classify}}')
for ip in range(print_n):
print(str(ip+1) + '. ' + str(class_names[y[sorted_indices[-ip]]]) + ': ' +
str([x for x in zip(features, X[sorted_indices[-ip], :]) if x[1] > 0]))

'''
print('The top ' + str(print_n) + ' hardest samples to classify')
for ip in range(print_n):
print_data[ip, 0] = class_names[y[sorted_indices[ip]]]
print_data[ip, 1:] = X[sorted_indices[ip]]
print(pd.DataFrame(data=print_data, columns=print_features))
'''
```

(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

**On titanic, the best choice is the simple sklearn's decision tree**

Part 0: constant classifier
Accuracy

$$\begin{bmatrix}0.6136136136136137\end{bmatrix}$$

simplified decision tree
Accuracy

$$\begin{bmatrix}0.7967967967967968\end{bmatrix}$$

sklearn's decision tree
Cross validation

$$\begin{bmatrix}0.7994012 & 0.78378378 & 0.8313253\end{bmatrix}$$

Accuracy

$$\begin{bmatrix}0.8048048048048049\end{bmatrix}$$

BaggedTrees on titanic
Cross validation

$$\begin{bmatrix}0.79640719 & 0.77177177 & 0.8253012\end{bmatrix}$$

Accuracy

$$\begin{bmatrix}0.8048048048048049\end{bmatrix}$$

RandomForest on titanic
Cross validation

$$\begin{bmatrix}0.79341317 & 0.77477477 & 0.82228916\end{bmatrix}$$

Accuracy

$$\begin{bmatrix}0.7967967967967968\end{bmatrix}$$

BoostedRandomForest on titanic
Cross validation

$$\begin{bmatrix}0.76047904 & 0.73573574 & 0.78313253\end{bmatrix}$$

Accuracy

$$\begin{bmatrix}0.8248248248248249\end{bmatrix}$$

---

**On spam, the best choice is also BoostedRandomForest(N=100, m=5):**

Part 0: constant classifier
Accuracy

$$\begin{bmatrix}0.7099767981438515\end{bmatrix}$$

simplified decision tree
Accuracy

$$\begin{bmatrix}0.7948569218870842\end{bmatrix}$$

sklearn's decision tree
Cross validation

$$\begin{bmatrix}0.71113689 & 0.72157773 & 0.71287703\end{bmatrix}$$

Accuracy

$$\begin{bmatrix}0.7211910286156226\end{bmatrix}$$

BaggedTrees on spam
Cross validation

$$\begin{bmatrix}0.72563805 & 0.73259861 & 0.73723898\end{bmatrix}$$

Accuracy

$$\begin{bmatrix}0.7310518174787317\end{bmatrix}$$

RandomForest on spam
Cross validation

$$\begin{bmatrix}0.79756381 & 0.80742459 & 0.77958237\end{bmatrix}$$

Accuracy

$$\begin{bmatrix}0.7594740912606341\end{bmatrix}$$

BoostedRandomForest on spam
Cross validation

$$\begin{bmatrix} 0.80858469 & 0.73607889 & 0.75928074 \end{bmatrix}$$

Accuracy

$$\begin{bmatrix} 0.758507347254447 \end{bmatrix}$$

I choose sklearn_decision_tree for titanic and BoostedRandomForest for spam. Here is my code:

```
# Part j
if dataset == 'titanic':
test_hat = clf.predict(Z)
test_hat = np.array(test_hat, dtype=bool)
np.savetxt('datasets/titanic/submission.txt', test_hat, fmt='%d', delimiter='\n')
elif dataset == 'spam':
test_hat = brf.predict(Z)
test_hat = np.array(test_hat, dtype=bool)
np.savetxt('datasets/spam_data/submission.txt', test_hat, fmt='%d', delimiter='\n')
```

(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).

Nothing to say.

---

```
# You may want to install "gprof2dot"
import io
from collections import Counter

import numpy as np
import scipy.io
import sklearn.model_selection
import sklearn.tree
from numpy import genfromtxt
from scipy import stats
from sklearn.base import BaseEstimator, ClassifierMixin

import pydot
import gprof2dot
import os
import builtins as __builtin__
import pandas as pd

eps = 1e-5 # a small number


def print(*args, **kwargs):
"""My custom print() function."""
# Adding new arguments to the print function signature
# is probably a bad idea.
# Instead consider testing if custom argument keywords
# are present in kwargs
tempargs = list(args)
for iarg, arg in enumerate(tempargs):
if (type(arg).__module__ == np.__name__):
tempargs[iarg] = bmatrix(arg)
elif isinstance(arg, pd.DataFrame):
tempargs[iarg] = btabu(arg)
elif isinstance(arg, str):
#if '\\' in arg:
#    arg = arg.replace('\\', r' \textbackslash ')
if '_' in arg:
arg = arg.replace('_', r'\_')
if '<' in arg:
arg = arg.replace('<', r'\textless ')
if '>' in arg:
arg = arg.replace('>', r'\textgreater ')
if '<=' in arg:
arg = arg.replace('<=', r'\le ')
if '>=' in arg:
arg = arg.replace('>=', r'\ge ')
tempargs[iarg] = arg
else:
tempargs[iarg] = str(arg).replace('_', r'\_')
```

```
tempargs = tuple(tempargs)
__builtin__.print(*tempargs, **kwargs, end='')
__builtin__.print(r' \\')


def bmatrix(a):
"""Returns a LaTeX bmatrix
Retrieved from https://stackoverflow.com/questions/17129290/numpy-2d-and-1d-array-to-latex-bmatrix
:a: numpy array
:returns: LaTeX bmatrix as a string
"""
a = np.array(a)
if len(a.shape) > 2:
raise ValueError('bmatrix can at most display two dimensions')
lines = str(a).replace('[', '').replace(']', '').splitlines()
rv = [r'\[']
rv += [r'\begin{bmatrix}']
rv += [' ' + ' & '.join(l.split()) + r'\\' for l in lines]
rv += [r'\end{bmatrix}']
rv += [r'\]']
return '\n'.join(rv)


def btabu(a):
nCol = len(a.columns)
rv = [r'\begin{tabu} to 1.0\textwidth { ' + '|X[c] ' * (nCol + 1) + '| }']
rv += [r'\hline']
currentRow = ' '
for idx, column in enumerate(a.columns):
currentRow += ' & ' + str(column)
rv += [currentRow + '\\\\']
for idx, row in a.iterrows():
currentRow = str(idx) + ' '
for _, column in enumerate(a.columns):
currentRow += ' & ' + str(row[column])
rv += [r'\hline']
rv += [currentRow + '\\\\']
rv += [r'\hline']
rv += [r'\end{tabu}\\']
return '\n'.join(rv)


class DecisionTree:
def __init__(self, max_depth=3, feature_labels=None):
self.max_depth = max_depth
self.features = feature_labels
self.left, self.right = None, None # for non-leaf nodes
self.split_idx, self.thresh = None, None # for non-leaf nodes
self.data, self.pred = None, None # for leaf nodes

@staticmethod
def information_gain(X, y, thresh):
# TODO implement information gain function
class_labels = np.unique(y)
y0, y1 = y[X < thresh], y[X >= thresh]
n0 = len(y0)
n1 = len(y1)
n = len(y)
entropy = 0.0
for label in class_labels:
if np.count_nonzero(y == label) > 0:
entropy -= np.count_nonzero(y == label) * np.log2(np.count_nonzero(y == label) / n)
if np.count_nonzero(y0 == label) > 0:
```

```
entropy += np.count_nonzero(y0 == label) * np.log2(np.count_nonzero(y0 == label) / n0)
if np.count_nonzero(y1 == label) > 0:
entropy += np.count_nonzero(y1 == label) * np.log2(np.count_nonzero(y1 == label) / n1)
entropy /= n
return entropy

@staticmethod
def gini_impurity(X, y, thresh):
# TODO implement gini_impurity function
class_labels = np.unique(y)
y0, y1 = y[X < thresh], y[X >= thresh]
n0 = len(y0)
n1 = len(y1)
n = len(y)
gini = n0 + n1
for label in class_labels:
gini -= np.count_nonzero(y0 == label) ** 2 / n0
gini -= np.count_nonzero(y1 == label) ** 2 / n1
gini /= n
return -gini

def split(self, X, y, idx, thresh):
X0, idx0, X1, idx1 = self.split_test(X, idx=idx, thresh=thresh)
y0, y1 = y[idx0], y[idx1]
return X0, y0, X1, y1

def split_test(self, X, idx, thresh):
idx0 = np.where(X[:, idx] < thresh)[0]
idx1 = np.where(X[:, idx] >= thresh)[0]
X0, X1 = X[idx0, :], X[idx1, :]
return X0, idx0, X1, idx1

def fit(self, X, y):
if self.max_depth > 0:
# compute entropy gain for all single-dimension splits,
# thresholding with a linear interpolation of 10 values
gains = []
# The following logic prevents thresholding on exactly the minimum
# or maximum values, which may not lead to any meaningful node
# splits.
thresh = np.array([
np.linspace(np.min(X[:, i]) + eps, np.max(X[:, i]) - eps, num=10)
for i in range(X.shape[1])
])
for i in range(X.shape[1]):
gains.append([self.information_gain(X[:, i], y, t) for t in thresh[i, :]])

gains = np.nan_to_num(np.array(gains))
self.split_idx, thresh_idx = np.unravel_index(np.argmax(gains), gains.shape)
self.thresh = thresh[self.split_idx, thresh_idx]
X0, y0, X1, y1 = self.split(X, y, idx=self.split_idx, thresh=self.thresh)
if X0.size > 0 and X1.size > 0:
self.left = DecisionTree(
max_depth=self.max_depth - 1, feature_labels=self.features)
self.left.fit(X0, y0)
self.right = DecisionTree(
max_depth=self.max_depth - 1, feature_labels=self.features)
self.right.fit(X1, y1)
else:
self.max_depth = 0
self.data, self.labels = X, y
self.pred = stats.mode(y).mode[0]
else:
```

```
self.data, self.labels = X, y
self.pred = stats.mode(y).mode[0]
return self

def predict(self, X):
if self.max_depth == 0:
return self.pred * np.ones(X.shape[0])
else:
X0, idx0, X1, idx1 = self.split_test(X, idx=self.split_idx, thresh=self.thresh)
yhat = np.zeros(X.shape[0])
yhat[idx0] = self.left.predict(X0)
yhat[idx1] = self.right.predict(X1)
return yhat


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 = [
sklearn.tree.DecisionTreeClassifier(random_state=i, **self.params)
for i in range(self.n)
]

def fit(self, X, y):
# TODO implement function
for i in range(self.n):
idx = np.random.choice(len(y), len(y), replace=True)
X_rand = X[idx]
y_rand = y[idx]
self.decision_trees[i].fit(X_rand, y_rand)
return self

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

def print_common_split(self):
roots = []
for _, t in enumerate(self.decision_trees):
roots.append(str(t.tree_.feature[0]) + '-' + str(t.tree_.threshold[0]))
counter = Counter(roots)
first_splits = [(term[0], term[1]) for term in
counter.most_common()]
print('The most common splits at the root node of the tree are: ')
for i, this_split in enumerate(first_splits):
name, thold = this_split[0].split('-', 1)
name = features[int(name)]
if isinstance(name, bytes):
name = str(name, "utf-8")
else:
name = str(name)
print(str(i + 1) + '. ("' + name + '")' + '<' + str(thold) + ' (' + str(this_split[1]) + ' trees)')


class RandomForest(BaggedTrees):
def __init__(self, params=None, n=200, m=1):
if params is None:
```

```
params = {'max_features': m}
# TODO implement function
super().__init__(params=params, n=n)


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
for j in range(self.n):
idx = np.random.choice(len(y), len(y), replace=True, p=self.w)
X_rand = X[idx]
y_rand = y[idx]
self.decision_trees[j].fit(X_rand, y_rand)
yhat = self.decision_trees[j].predict(X)
Ihat = np.array(y != yhat, dtype=np.float)
ej = sum(self.w * Ihat) / sum(self.w)
self.a[j] = 1 / 2 * np.log((1 - ej) / ej)
self.w[Ihat == 1] *= np.exp(self.a[j])
self.w[Ihat != 1] *= np.exp(-self.a[j])
self.w /= sum(self.w)
return self

def predict(self, X):
# TODO implement function
ypred = []
yhat = []
for j in range(self.n):
yhat.append(self.decision_trees[j].predict(X))
yhat = np.vstack(yhat)
for i in range(X.shape[0]):
class_labels = np.unique(yhat[:, i])
yscores = []
for label in class_labels:
yscores.append(self.a.dot(yhat[:, i] == label))
ypred.append(yhat[np.argmax(yscores, axis=0), i])
return ypred


def preprocess(data, fill_mode=True, min_freq=10, onehot_cols=[]):
# fill_mode = False

# Temporarily assign -1 to missing data
data[data == b''] = '-1'

# Hash the columns (used for handling strings)
onehot_encoding = []
onehot_features = []
for col in onehot_cols:
counter = Counter(data[:, col])
for term in counter.most_common():
if term[0] == b'-1':
continue
if term[-1] <= min_freq:
break
onehot_features.append(features[col] + b'-' + term[0])
onehot_encoding.append((data[:, col] == term[0]).astype(np.float))
data[:, col] = '0'
onehot_encoding = np.array(onehot_encoding).T
data = np.hstack([np.array(data, dtype=np.float), np.array(onehot_encoding)])

# Replace missing data with the mode value. We use the mode instead of
```

```
# the mean or median because this makes more sense for categorical
# features such as gender or cabin type, which are not ordered.
if fill_mode:
for i in range(data.shape[-1]):
mode = stats.mode(data[((data[:, i] < -1 - eps) +
(data[:, i] > -1 + eps))][:, i]).mode[0]
data[(data[:, i] > -1 - eps) * (data[:, i] < -1 + eps)][:, i] = mode

return data, onehot_features


def evaluate(clf):
print("Cross validation", sklearn.model_selection.cross_val_score(clf, X, y))
if hasattr(clf, "decision_trees"):
# counter = Counter([t.tree_.feature[0] for t in clf.decision_trees])
# first_splits = [(features[term[0]], term[1]) for term in counter.most_common()]
# print("First splits", first_splits)
roots = []
for _, t in enumerate(clf.decision_trees):
roots.append(str(t.tree_.feature[0]) + '-' + str(t.tree_.threshold[0]))
counter = Counter(roots)
first_splits = [(term[0], term[1]) for term in
counter.most_common()]
print('The most common splits at the root node of the tree are: ')
for i, this_split in enumerate(first_splits):
name, thold = this_split[0].split('-', 1)
name = features[int(name)]
if isinstance(name, bytes):
name = str(name, "utf-8")
else:
name = str(name)
print(str(i + 1) + '. ("' + name + '")' + '<' + str(thold) + ' (' + str(this_split[1]) + ' trees)')


if __name__ == "__main__":
os.environ["PATH"] += os.pathsep + 'C:/Program Files (x86)/graphviz-2.38/bin/'
# dataset = "titanic"
# dataset = 'spam'
all_datasets = ['titanic', 'spam']
params = {
"max_depth": 5,
# "random_state": 6,
"min_samples_leaf": 10,
}
N = 100
for dataset in all_datasets:

if dataset == "titanic":
# Load titanic data
path_train = 'datasets/titanic/titanic_training.csv'
data = genfromtxt(path_train, delimiter=',', dtype=None)
path_test = 'datasets/titanic/titanic_testing_data.csv'
test_data = genfromtxt(path_test, delimiter=',', dtype=None)
y = data[1:, 0] # label = survived
class_names = ["Died", "Survived"]
features = list(data[0, 1:])

labeled_idx = np.where(y != b'')[0]
y = np.array(y[labeled_idx], dtype=np.int)
print("\n\nPart (b): preprocessing the titanic dataset")
X, onehot_features = preprocess(data[1:, 1:], onehot_cols=[1, 5, 7, 8])
X = X[labeled_idx, :]
Z, _ = preprocess(test_data[1:, :], onehot_cols=[1, 5, 7, 8])
```

```
assert X.shape[1] == Z.shape[1]
features = list(data[0, 1:]) + onehot_features

elif dataset == "spam":
features = [
"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",
"semicolon", "dollar", "sharp", "exclamation", "parenthesis", "square_bracket",
"ampersand"
]
assert len(features) == 32

# Load spam data
path_train = 'datasets/spam_data/spam_data.mat'
data = scipy.io.loadmat(path_train)
X = data['training_data']
y = np.squeeze(data['training_labels'])
Z = data['test_data']
class_names = ["Ham", "Spam"]

else:
raise NotImplementedError("Dataset %s not handled" % dataset)

print("Features:", features)
print("Train/test size:", X.shape, Z.shape)

print("\n\nPart 0: constant classifier")
print("Accuracy", 1 - np.sum(y) / y.size)

# Basic decision tree
print("\n\nsimplified decision tree")
dt = DecisionTree(max_depth=3, feature_labels=features)
dt.fit(X, y)
# print("Predictions", dt.predict(Z)[:100])
print("Accuracy", 1 - np.sum(abs(dt.predict(X) - y)) / y.size)

print("\n\nsklearn's decision tree")
this_params = params
this_params['max_depth'] = 3
clf = sklearn.tree.DecisionTreeClassifier(random_state=0, **this_params)
clf.fit(X, y)
evaluate(clf)
out = io.StringIO()
sklearn.tree.export_graphviz(
clf, out_file=out, feature_names=features, class_names=class_names)
graph = pydot.graph_from_dot_data(out.getvalue())
pydot.graph_from_dot_data(out.getvalue())[0].write_pdf("%s-tree.pdf" % dataset)
print("Accuracy", 1 - np.sum(abs(clf.predict(X) - y)) / y.size)

# TODO implement and evaluate parts c-h
# Part e
print('\n\nBaggedTrees on ' + dataset)
bt = BaggedTrees(params, n=N)
bt.fit(X, y)
evaluate(bt)
print("Accuracy", 1 - np.sum(abs(bt.predict(X) - y)) / y.size)

# Part g
print('\n\nRandomForest on ' + dataset)
rf = RandomForest(params, n=N, m=np.int(np.sqrt(X.shape[1])))
rf.fit(X, y)
evaluate(rf)
```

```
print("Accuracy", 1 - np.sum(abs(rf.predict(X) - y)) / y.size)

# Part g
print('\n\nBoostedRandomForest on ' + dataset)
brf = BoostedRandomForest(params, n=N, m=np.int(np.sqrt(X.shape[1])))
brf.fit(X, y)
evaluate(brf)
print("Accuracy", 1 - np.sum(abs(brf.predict(X) - y)) / y.size)
if dataset == 'spam':
sorted_indices = np.argsort(brf.w)
print_n = 10
print_data = np.zeros((print_n, len(features) + 1), dtype=object)
print_features = features.copy()
print_features.insert(0, 'label')

print('\\textbf{{The top ' + str(print_n) + ' easiest samples to classify}}')
for ip in range(print_n):
print(str(ip+1) + '. ' + str(class_names[y[sorted_indices[ip]]]) + ': ' +
str([x for x in zip(features, X[sorted_indices[ip], :]) if x[1] > 0]))
print()
print('\\textbf{{The top ' + str(print_n) + ' hardest samples to classify}}')
for ip in range(print_n):
print(str(ip+1) + '. ' + str(class_names[y[sorted_indices[-ip]]]) + ': ' +
str([x for x in zip(features, X[sorted_indices[-ip], :]) if x[1] > 0]))

'''
print('The top ' + str(print_n) + ' hardest samples to classify')
for ip in range(print_n):
print_data[ip, 0] = class_names[y[sorted_indices[ip]]]
print_data[ip, 1:] = X[sorted_indices[ip]]
print(pd.DataFrame(data=print_data, columns=print_features))
'''

# Part j
if dataset == 'titanic':
test_hat = clf.predict(Z)
test_hat = np.array(test_hat, dtype=bool)
np.savetxt('datasets/titanic/submission.txt', test_hat, fmt='%d', delimiter='\n')
elif dataset == 'spam':
test_hat = brf.predict(Z)
test_hat = np.array(test_hat, dtype=bool)
np.savetxt('datasets/spam_data/submission.txt', test_hat, fmt='%d', delimiter='\n')
```

**Question 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.

**Given that Decision Tree is this useful, let's try to use it to predict whether a student is from one section or the other section based on their grades of different assignments in Bio 1B.**

Below is the result I get. I think we simple decision tree performs really well for this kind of simple task. The best predictor for section number is the students' lab assignment grades.

Part 0: constant classifier

Accuracy

$$\begin{bmatrix}0.47058823529411764\end{bmatrix}$$

simplified decision tree

Accuracy

$$\begin{bmatrix}0.8529411764705882\end{bmatrix}$$

sklearn's decision tree

Cross validation

$$\begin{bmatrix}0.5 & 0.63636364 & 0.81818182\end{bmatrix}$$

Accuracy

$$\begin{bmatrix}0.7941176470588236\end{bmatrix}$$

BaggedTrees on bio1b

Cross validation

$$\begin{bmatrix}0.58333333 & 0.63636364 & 0.81818182\end{bmatrix}$$

The most common splits at the root node of the tree are:

1. ("Labs")<8.39347839355 (27 trees)
Accuracy

$$\begin{bmatrix}0.7941176470588236\end{bmatrix}$$

RandomForest on bio1b
Cross validation

$$\begin{bmatrix}0.5 & 0.63636364 & 0.81818182\end{bmatrix}$$

The most common splits at the root node of the tree are:
1. ("Labs")<8.39347839355 (25 trees)
Accuracy

$$\begin{bmatrix}0.7941176470588236\end{bmatrix}$$

BoostedRandomForest on bio1b
Cross validation

$$\begin{bmatrix}0.66666667 & 0.54545455 & 0.72727273\end{bmatrix}$$

The most common splits at the root node of the tree are:
1. ("SIS Login ID")<1513155.5 (6 trees)
Accuracy

$$\begin{bmatrix}0.7941176470588236\end{bmatrix}$$

---

```
# You may want to install "gprof2dot"
import io
from collections import Counter

import numpy as np
import scipy.io
import sklearn.model_selection
import sklearn.tree
from numpy import genfromtxt
from scipy import stats
from sklearn.base import BaseEstimator, ClassifierMixin

import pydot
import gprof2dot
import os
import builtins as __builtin__
import pandas as pd

eps = 1e-5 # a small number


def print(*args, **kwargs):
"""My custom print() function."""
# Adding new arguments to the print function signature
# is probably a bad idea.
# Instead consider testing if custom argument keywords
```

```
# are present in kwargs
tempargs = list(args)
for iarg, arg in enumerate(tempargs):
if (type(arg).__module__ == np.__name__):
tempargs[iarg] = bmatrix(arg)
elif isinstance(arg, pd.DataFrame):
tempargs[iarg] = btabu(arg)
elif isinstance(arg, str):
#if '\\' in arg:
#    arg = arg.replace('\\', r' \textbackslash ')
if '_' in arg:
arg = arg.replace('_', r'\_')
if '<' in arg:
arg = arg.replace('<', r'\textless ')
if '>' in arg:
arg = arg.replace('>', r'\textgreater ')
if '<=' in arg:
arg = arg.replace('<=', r'\le ')
if '>=' in arg:
arg = arg.replace('>=', r'\ge ')
tempargs[iarg] = arg
else:
tempargs[iarg] = str(arg).replace('_', r'\_')
tempargs = tuple(tempargs)
__builtin__.print(*tempargs, **kwargs, end='')
__builtin__.print(r' \\')


def bmatrix(a):
"""Returns a LaTeX bmatrix
Retrieved from https://stackoverflow.com/questions/17129290/numpy-2d-and-1d-array-to-latex-bmatrix
:a: numpy array
:returns: LaTeX bmatrix as a string
"""
a = np.array(a)
if len(a.shape) > 2:
raise ValueError('bmatrix can at most display two dimensions')
lines = str(a).replace('[', '').replace(']', '').splitlines()
rv = [r'\[']
rv += [r'\begin{bmatrix}']
rv += [' ' + ' & '.join(l.split()) + r'\\' for l in lines]
rv += [r'\end{bmatrix}']
rv += [r'\]']
return '\n'.join(rv)


def btabu(a):
nCol = len(a.columns)
rv = [r'\begin{tabu} to 1.0\textwidth { ' + '|X[c] ' * (nCol + 1) + '| }']
rv += [r'\hline']
currentRow = ' '
for idx, column in enumerate(a.columns):
currentRow += ' & ' + str(column)
rv += [currentRow + '\\\\']
for idx, row in a.iterrows():
currentRow = str(idx) + ' '
for _, column in enumerate(a.columns):
currentRow += ' & ' + str(row[column])
rv += [r'\hline']
rv += [currentRow + '\\\\']
rv += [r'\hline']
rv += [r'\end{tabu}\\']
return '\n'.join(rv)
```

```
class DecisionTree:
def __init__(self, max_depth=3, feature_labels=None):
self.max_depth = max_depth
self.features = feature_labels
self.left, self.right = None, None # for non-leaf nodes
self.split_idx, self.thresh = None, None # for non-leaf nodes
self.data, self.pred = None, None # for leaf nodes

@staticmethod
def information_gain(X, y, thresh):
# TODO implement information gain function
class_labels = np.unique(y)
y0, y1 = y[X < thresh], y[X >= thresh]
n0 = len(y0)
n1 = len(y1)
n = len(y)
entropy = 0.0
for label in class_labels:
if np.count_nonzero(y == label) > 0:
entropy -= np.count_nonzero(y == label) * np.log2(np.count_nonzero(y == label) / n)
if np.count_nonzero(y0 == label) > 0:
entropy += np.count_nonzero(y0 == label) * np.log2(np.count_nonzero(y0 == label) / n0)
if np.count_nonzero(y1 == label) > 0:
entropy += np.count_nonzero(y1 == label) * np.log2(np.count_nonzero(y1 == label) / n1)
entropy /= n
return entropy

@staticmethod
def gini_impurity(X, y, thresh):
# TODO implement gini_impurity function
class_labels = np.unique(y)
y0, y1 = y[X < thresh], y[X >= thresh]
n0 = len(y0)
n1 = len(y1)
n = len(y)
gini = n0 + n1
for label in class_labels:
gini -= np.count_nonzero(y0 == label) ** 2 / n0
gini -= np.count_nonzero(y1 == label) ** 2 / n1
gini /= n
return -gini

def split(self, X, y, idx, thresh):
X0, idx0, X1, idx1 = self.split_test(X, idx=idx, thresh=thresh)
y0, y1 = y[idx0], y[idx1]
return X0, y0, X1, y1

def split_test(self, X, idx, thresh):
idx0 = np.where(X[:, idx] < thresh)[0]
idx1 = np.where(X[:, idx] >= thresh)[0]
X0, X1 = X[idx0, :], X[idx1, :]
return X0, idx0, X1, idx1

def fit(self, X, y):
if self.max_depth > 0:
# compute entropy gain for all single-dimension splits,
# thresholding with a linear interpolation of 10 values
gains = []
# The following logic prevents thresholding on exactly the minimum
# or maximum values, which may not lead to any meaningful node
# splits.
```

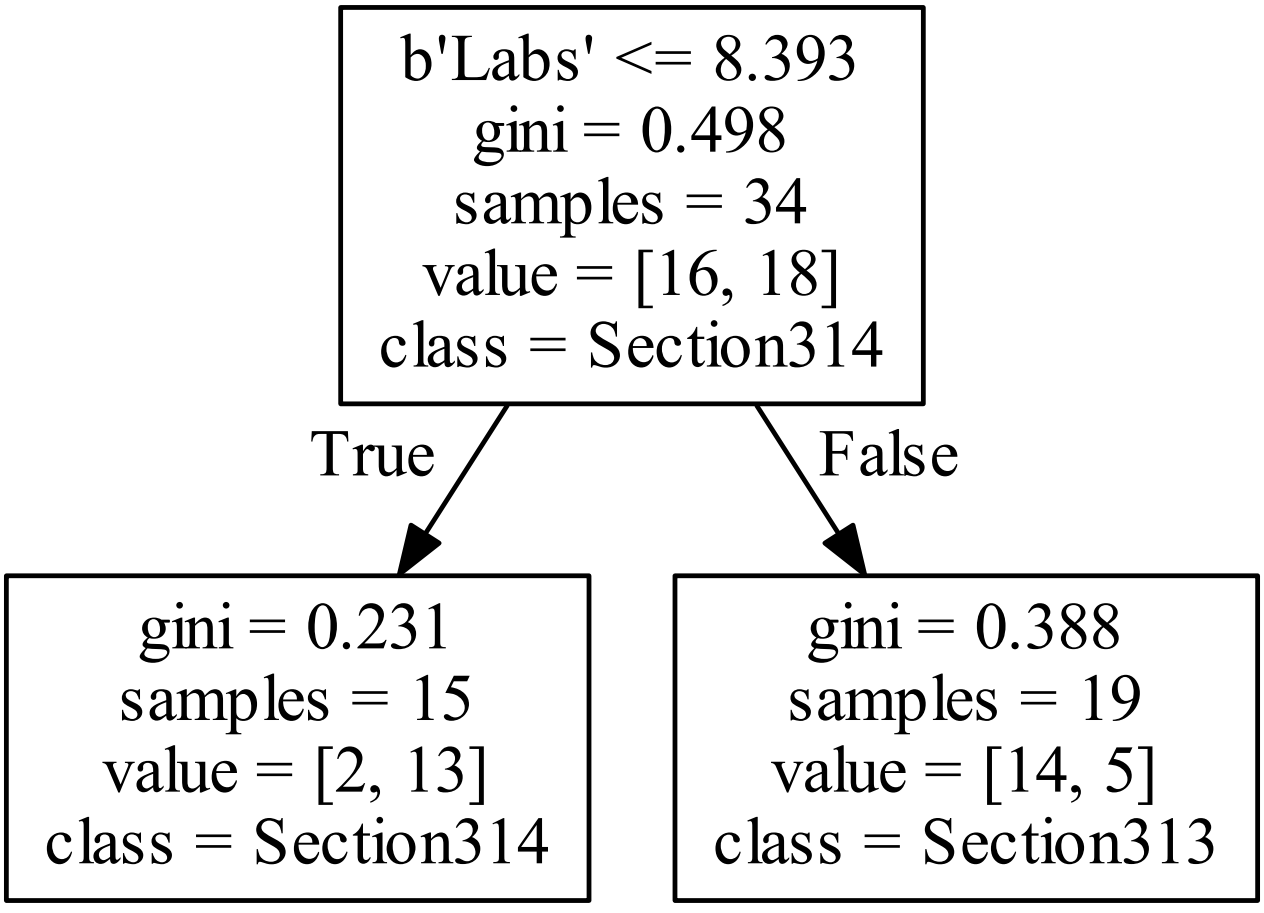
```
thresh = np.array([
np.linspace(np.min(X[:, i]) + eps, np.max(X[:, i]) - eps, num=10)
for i in range(X.shape[1])
])
for i in range(X.shape[1]):
gains.append([self.information_gain(X[:, i], y, t) for t in thresh[i, :]])

gains = np.nan_to_num(np.array(gains))
self.split_idx, thresh_idx = np.unravel_index(np.argmax(gains), gains.shape)
self.thresh = thresh[self.split_idx, thresh_idx]
X0, y0, X1, y1 = self.split(X, y, idx=self.split_idx, thresh=self.thresh)
if X0.size > 0 and X1.size > 0:
self.left = DecisionTree(
max_depth=self.max_depth - 1, feature_labels=self.features)
self.left.fit(X0, y0)
self.right = DecisionTree(
max_depth=self.max_depth - 1, feature_labels=self.features)
self.right.fit(X1, y1)
else:
self.max_depth = 0
self.data, self.labels = X, y
self.pred = stats.mode(y).mode[0]
else:
self.data, self.labels = X, y
self.pred = stats.mode(y).mode[0]
return self

def predict(self, X):
if self.max_depth == 0:
return self.pred * np.ones(X.shape[0])
else:
X0, idx0, X1, idx1 = self.split_test(X, idx=self.split_idx, thresh=self.thresh)
yhat = np.zeros(X.shape[0])
yhat[idx0] = self.left.predict(X0)
yhat[idx1] = self.right.predict(X1)
return yhat


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 = [
sklearn.tree.DecisionTreeClassifier(random_state=i, **self.params)
for i in range(self.n)
]

def fit(self, X, y):
# TODO implement function
for i in range(self.n):
idx = np.random.choice(len(y), len(y), replace=True)
X_rand = X[idx]
y_rand = y[idx]
self.decision_trees[i].fit(X_rand, y_rand)
return self

def predict(self, X):
yhat = []
for i in range(self.n):
yhat.append(self.decision_trees[i].predict(X))
yhat = np.vstack(yhat)
```

```
return np.round(np.mean(yhat, axis=0))

def print_common_split(self):
roots = []
for _, t in enumerate(self.decision_trees):
roots.append(str(t.tree_.feature[0]) + '-' + str(t.tree_.threshold[0]))
counter = Counter(roots)
first_splits = [(term[0], term[1]) for term in
counter.most_common()]
print('The most common splits at the root node of the tree are: ')
for i, this_split in enumerate(first_splits):
name, thold = this_split[0].split('-', 1)
name = features[int(name)]
if isinstance(name, bytes):
name = str(name, "utf-8")
else:
name = str(name)
print(str(i + 1) + '. ("' + name + '")' + '<' + str(thold) + ' (' + str(this_split[1]) + ' trees)')


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


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
for j in range(self.n):
idx = np.random.choice(len(y), len(y), replace=True, p=self.w)
X_rand = X[idx]
y_rand = y[idx]
self.decision_trees[j].fit(X_rand, y_rand)
yhat = self.decision_trees[j].predict(X)
Ihat = np.array(y != yhat, dtype=np.float)
ej = sum(self.w * Ihat) / sum(self.w)
if ej == 0:
self.n = j + 1
break
self.a[j] = 1 / 2 * np.log((1 - ej) / ej)
self.w[Ihat == 1] *= np.exp(self.a[j])
self.w[Ihat != 1] *= np.exp(-self.a[j])
self.w /= sum(self.w)
return self

def predict(self, X):
# TODO implement function
ypred = []
yhat = []
for j in range(self.n):
yhat.append(self.decision_trees[j].predict(X))
yhat = np.vstack(yhat)
for i in range(X.shape[0]):
class_labels = np.unique(yhat[:, i])
yscores = []
for label in class_labels:
yscores.append(self.a.dot(yhat[:, i] == label))
ypred.append(yhat[np.argmax(yscores, axis=0), i])
```

```
return ypred


def preprocess(data, fill_mode=True, min_freq=10, onehot_cols=[]):
# fill_mode = False

# Temporarily assign -1 to missing data
data[data == b''] = '-1'

# Hash the columns (used for handling strings)
onehot_encoding = []
onehot_features = []
for col in onehot_cols:
counter = Counter(data[:, col])
for term in counter.most_common():
if term[0] == b'-1':
continue
if term[-1] <= min_freq:
break
onehot_features.append(features[col] + b'-' + term[0])
onehot_encoding.append((data[:, col] == term[0]).astype(np.float))
data[:, col] = '0'
onehot_encoding = np.array(onehot_encoding).T
if onehot_cols:
data = np.hstack([np.array(data, dtype=np.float), np.array(onehot_encoding)])
else:
data = np.array(data, dtype=np.float)

# Replace missing data with the mode value. We use the mode instead of
# the mean or median because this makes more sense for categorical
# features such as gender or cabin type, which are not ordered.
if fill_mode:
for i in range(data.shape[-1]):
mode = stats.mode(data[((data[:, i] < -1 - eps) +
(data[:, i] > -1 + eps))][:, i]).mode[0]
data[(data[:, i] > -1 - eps) * (data[:, i] < -1 + eps)][:, i] = mode

return data, onehot_features


def evaluate(clf):
print("Cross validation", sklearn.model_selection.cross_val_score(clf, X, y))
if hasattr(clf, "decision_trees"):
# counter = Counter([t.tree_.feature[0] for t in clf.decision_trees])
# first_splits = [(features[term[0]], term[1]) for term in counter.most_common()]
# print("First splits", first_splits)
roots = []
for _, t in enumerate(clf.decision_trees):
roots.append(str(t.tree_.feature[0]) + '-' + str(t.tree_.threshold[0]))
counter = Counter(roots)
first_splits = [(term[0], term[1]) for term in
counter.most_common()]
print('The most common splits at the root node of the tree are: ')
for i, this_split in enumerate(first_splits):
if i > 0:
break
name, thold = this_split[0].split('-', 1)
name = features[int(name)]
if isinstance(name, bytes):
name = str(name, "utf-8")
else:
name = str(name)
print(str(i + 1) + '. ("' + name + '")' + '<' + str(thold) + ' (' + str(this_split[1]) + ' trees)')
```

```
if __name__ == "__main__":
os.environ["PATH"] += os.pathsep + 'C:/Program Files (x86)/graphviz-2.38/bin/'
params = {
"max_depth": 5,
# "random_state": 6,
"min_samples_leaf": 10,
}
N = 100

dataset='bio1b'
# Load titanic data
path_train = 'bio1b_grades.csv'
data = genfromtxt(path_train, delimiter=',', dtype=None)
path_test = 'bio1b_grades.csv'
# test_data = genfromtxt(path_test, delimiter=',', dtype=None)
y = data[1:, 0] # label = survived
class_names = ["Section313", "Section314"]
features = list(data[0, 1:])

labeled_idx = np.where(y != b'')[0]
y = np.array(y[labeled_idx], dtype=np.int)
X, onehot_features = preprocess(data[1:, 1:])
X = X[labeled_idx, :]
# Z, _ = preprocess(test_data[1:, :])
# assert X.shape[1] == Z.shape[1]
features = list(data[0, 1:]) + onehot_features

print("\n\nPart 0: constant classifier")
print("Accuracy", 1 - np.sum(y) / y.size)

# Basic decision tree
print("\n\nsimplified decision tree")
dt = DecisionTree(max_depth=3, feature_labels=features)
dt.fit(X, y)
# print("Predictions", dt.predict(Z)[:100])
print("Accuracy", 1 - np.sum(abs(dt.predict(X) - y)) / y.size)

print("\n\nsklearn's decision tree")
this_params = params
this_params['max_depth'] = 3
clf = sklearn.tree.DecisionTreeClassifier(random_state=0, **this_params)
clf.fit(X, y)
evaluate(clf)
out = io.StringIO()
sklearn.tree.export_graphviz(
clf, out_file=out, feature_names=features, class_names=class_names)
graph = pydot.graph_from_dot_data(out.getvalue())
pydot.graph_from_dot_data(out.getvalue())[0].write_pdf("%s-tree.pdf" % dataset)
print("Accuracy", 1 - np.sum(abs(clf.predict(X) - y)) / y.size)

# TODO implement and evaluate parts c-h
# Part e
print('\n\nBaggedTrees on ' + dataset)
bt = BaggedTrees(params, n=N)
bt.fit(X, y)
evaluate(bt)
print("Accuracy", 1 - np.sum(abs(bt.predict(X) - y)) / y.size)

# Part g
print('\n\nRandomForest on ' + dataset)
rf = RandomForest(params, n=N, m=np.int(np.sqrt(X.shape[1])))
```

```
rf.fit(X, y)
evaluate(rf)
print("Accuracy", 1 - np.sum(abs(rf.predict(X) - y)) / y.size)

# Part g
print('\n\nBoostedRandomForest on ' + dataset)
brf = BoostedRandomForest(params, n=N, m=np.int(np.sqrt(X.shape[1])))
brf.fit(X, y)
evaluate(brf)
print("Accuracy", 1 - np.sum(abs(brf.predict(X) - y)) / y.size)
```


b'Labs' <= 8.393
gini = 0.498
samples = 34
value = [16, 18]
class = Section314
True
False
gini = 0.231
samples = 15
value = [2, 13]
class = Section314
gini = 0.388
samples = 19
value = [14, 5]
class = Section313