CS 189 Introduction to Machine Learning
Fall 2017 HW12

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

1 Getting Started

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

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

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

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

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

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

2 ℓ_1-Regularized Linear Regression

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

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

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

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

where $\lambda > 0$.

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

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

where A_i is the i-th column of A.

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

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

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

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

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

where $\lambda > 0$.

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

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

3 Variance of Sparse Linear Models

In this question, we will analyze the variance of sparse linear models. In particular, we will analyze two procedures that perform feature selection in linear models, and show quantitatively that feature selection lowers the variance of linear models. This should make sense to you at an intuitive level: enforcing sparsity is equivalent to deliberately constraining model complexity; think about where this puts you on the bias variance trade-off.

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

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

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

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

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

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

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

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

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

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 λ in absolute value.

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

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

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

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

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

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

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

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

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

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

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.

(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 σ^2.**

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

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

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

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

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.)

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

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

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

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

4 Decision Trees and Random Forests

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

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

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

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

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

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

See `decision_tree_starter.py` for the recommended starter code. The code sample is a simplified implementation, which combines decision tree and decision node functionalities and splits only on one feature at a time. **Include your information gain and entropy code.**

Note: The sample implementation assumes that all features are continuous. You may convert all your features to be continuous or augment the implementation to handle discrete features.

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

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

(a) survived - 1 is survived; 0 is not. This is the class label.

(b) pclass - Measure of socioeconomic status: 1 is upper, 2 is middle, 3 is lower.

(c) sex - Male/Female

(d) age - Fractional if less than 1.

(e) sibsp - Number of siblings/spouses aboard the Titanic

(f) parch - Number of parents/children aboard the Titanic

(g) ticket - Ticket number

(h) fare - Fare.

(i) cabin - Cabin number.

(j) embarked - Port of Embarkation (C = Cherbourg, Q = Queenstown, S = Southampton)

(c) **Apply your decision tree to the titanic dataset. Train a shallow decision tree** (for example, a depth 3 tree, although you may choose any depth that looks good) and **visualize your tree**. Include for each non-leaf node the feature name and the split rule, and include for leaf nodes the class your decision tree would assign.

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

(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
```

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

(a) ("viagra") ≥ 3 (20 trees)

(b) ("thanks") < 4 (15 trees)

(c) ("nigeria") ≥ 1 (5 trees)

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

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

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

(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
```

(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.**

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

$$a_j = \frac{1}{2} \log \frac{1 - e_j}{e_j}$$

where e_j is the weighted error:

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

and $I_j(x_i)$ is an indicator for datum i being *incorrect*.

Then update the weights as follows:

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

Repeat until you have M trees.

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

$$z(x,c) = \sum_{j=1}^{M} a_j I_j(x,c).$$

where $I_j(x,c)$ is now an indicator variable for if tree j predicts data x with class label c. Then, the class with the highest weight is the prediction (classification result):

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

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

```
class BoostedRandomForest(RandomForest):

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

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

(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.**

(j) **Summarize the performance evaluation of each of the above trees and forests: a single decision tree, bagged trees, random forests, and boosted random forests**. For each of the 2 datasets, report your training and validation accuracies. You should be reporting 24 numbers (2 datasets $\times$ 4 classifiers $\times$ 3 data splits). Describe qualitatively which types of trees and forests performed best. Detail any parameters that worked well for you. **In addition, for each of the** 2 **datasets, train your best model and submit your predictions to Gradescope**. Your best Titanic classifier should exceed 73% accuracy and your best Spam classifier should exceed 76% accuracy for full points.

(k) You should submit

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

5 Your Own Question

Write your own question, and provide a thorough solution.

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

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

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