CS 189 Introduction to Machine Learning
Spring 2018

HW12

1 Getting Started

Read through this page carefully. You may typeset your homework in latex or submit neatly handwritten/scanned solutions. Please start each question on a new page. Deliverables:

1. Submit a PDF of your writeup, **with an appendix for your code**, to assignment on Gradescope, "HW12 Write-Up". If there are graphs, include those graphs in the correct sections. Do not simply reference your appendix.
2. If there is code, submit all code needed to reproduce your results, "HW12 Code".
3. If there is a test set, submit your test set evaluation results, "HW12 Test Set".

After you've submitted your homework, 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?

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

2 ℓ_1-Regularized Linear Regression: LASSO

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 of the LASSO in a simplified setting.

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

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

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

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.**

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

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

where $\mathbf{X}_i$ is the i-th column of $\mathbf{X}$. By having no interaction terms that link the different w_i variables, you know that the joint optimization can be decomposed into individual optimizations.

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

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

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

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

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

where $\lambda > 0$.

What is the condition for $\widehat{w}_i = 0$? 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 $\mathbf{w}$ (so $\mathbf{w}$ is a sparse vector). We have a Gaussian matrix $n \times d$ matrix $\mathbf{X}$ and an $n \times 1$ noise vector $\mathbf{z}$ where $n > 1$. Our measurements take the form $\mathbf{y} = \mathbf{X}\mathbf{w} + \mathbf{z}$. We want to extract the original image $\mathbf{w}$ given matrix $\mathbf{X}$ knowing that this image is sparse. The fact that $\mathbf{w}$ is sparse suggests using ℓ_1 regularization. Use the provided iPython notebook and apply ℓ_1 regularization.

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

Change the hyperparameter λ to extract the best looking image and report it.

3 Variance of Sparse Linear Models Obtained by Thresholding

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

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

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

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

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

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

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

(b) In this problem, we will analyze two estimators that explicitly take into account the fact that $\mathbf{w}^*$ is sparse, and consequently attain lower error than the vanilla least-squares estimate.

Let us define two operators. Given a vector $\mathbf{v} \in \mathbb{R}^d$, the operation $\tau_k(\mathbf{v})$ zeroes out all but the top k entries of $\mathbf{v}$ measured in absolute value. The operator $T_\lambda(\mathbf{v})$, 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 $\widehat{\mathbf{w}}_{\mathsf{LS}} = \mathbf{X}^\dagger \mathbf{y} = \mathbf{X}^\top \mathbf{y}$, 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

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

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

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

(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.**

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

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

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

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

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

(f) **Argue that the (random) error vector $\mathbf{e} = \widehat{\mathbf{w}}_{\mathsf{top}}(s) - \mathbf{w}^*$ 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.**

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

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

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

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

(j) Recall that we have already analyzed the performance of the simple least-squares estimator $\widehat{\mathbf{w}}_{\mathsf{LS}} = (\mathbf{X}^\top \mathbf{X})^{-1}\mathbf{X}^\top \mathbf{y}$ in earlier homework and an the earlier practice midterm. In particular, we showed that

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

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

respectively. Equations (1) and (2) represent the "prediction" error (or variance) and the mean squared error of the parameters of our OLS model, respectively. Recall that in this problem, we have been assuming that the matrix $\mathbf{X}$ has orthonormal columns, and so the bounds become

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

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

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

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

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

4 Decision Trees and Random Forests

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

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

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

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

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

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

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

(b) Before applying the decision-tree learning algorithm to the Titanic dataset, we will first pre-process the dataset. In the real-world, pre-processing the data is a very important step since real-life data can be quite imperfect. However, to make this problem easier, we have provided some code to preprocess the data. Look and the code and **describe how we deal with the following problems**:

- Some data points are misssing class labels;
- Some features are not numerical values;
- Some data points are missing some features.

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

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

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

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

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

(b) ("nigeria") ≥ 1 (5 trees)

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

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

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

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

```
class RandomForest(BaggedTrees):

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

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

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

where e_j is the weighted error:

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

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

Then update the weights as follows:

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

Repeat until you have M trees.

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

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

where $I_j(x, c)$ is now an indicator variable for whether tree j predicts class label c for data x.

Then, the class with the highest weighted votes is the prediction (classification result):

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

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

```
class BoostedRandomForest(RandomForest):

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

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

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

(j) **Summarize the performance evaluation of: a single decision tree, bagged trees, random forests, and boosted trees**. For each of the 2 datasets, report your training and validation accuracies. You should use a 3-fold cross validation, i.e., splitting the dataset into 3 parts. You should be reporting 32 numbers (2 datasets $\times$ 4 classifiers $\times(3 + 1)$ for 3 cross validation accuracies and 1 training accuracy). Describe qualitatively which types of trees and forests

performed best. Detail any parameters that worked well for you. **In addition, for each of the 2 datasets, train your best model and submit your predictions on the test data to Gradescope**. Your best Titanic classifier should exceed 73% accuracy and your best Spam classifier should exceed 76% accuracy for full points.

(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 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.