CS 189 Introduction to Machine Learning
Fall 2019 Jennifer Listgarten & Stella Yu

HW08

Due: 11/26 at 10 p.m.

0 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, "HW[X] 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, "HW[X] Code".
3. If there is a test set, submit your test set evaluation results, "HW[X] Test Set".
4. In all cases, replace "[X]" with the number of the assignment you are submitting.

1 Kernel SVM

Learning Goal: Derive the objective function for soft-margin kernel SVMs.

Assume we are doing classification or regression over $\mathbb{R}^d$. We first introduce the following abstract vector space:

$$H = \{f : \mathbb{R}^d \to \mathbb{R} \text{ such that } f(x) = \sum_{m=1}^{M} \alpha_m k(x, y_m) | \alpha_i \in \mathbb{R}, M \in \mathbb{N}, y_m \in \mathbb{R}^d\}, \tag{1}$$

where $k(x, y) : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$ is a kernel function that satisfies the property $k(x, y) = k(y, x)$ for any $x, y \in \mathbb{R}^d$ and for any distinct $y_1, y_2, \ldots, y_M \in \mathbb{R}^d$, the matrix $K \in \mathbb{R}^{M \times M}$ defined by $K_{ij} = k(y_i, y_j)$ is positive definite. Note that H can be described as an infinite dimensional vector space.

In this problem, we will give a derivation of kernel SVM from the view of function approximation with penalization. The objective function of a linear soft-margin SVM can be interpreted as Hinge Loss combined with L_2 regularization over the space of linear functions:

$$\min_{w \in R^d} \sum_{i=1}^{N} \max(0, 1 - y_i(w^T x_i)) + \lambda ||w||_2^2, \tag{2}$$

where $\lambda > 0$. We will show that the objective function of a kernel SVM can be interpreted in the same way: Hinge Loss plus L_2 regularization over the space H of functions defined by a kernel

function:

$$\min_{f \in H} \frac{1}{N} \sum_{i=1}^{N} \max(0, 1 - y_i f(x_i)) + \lambda \|f\|_H^2. \tag{3}$$

(a) Now we introduce an inner product on the above vector space H. We define the inner product between any two functions

$$f(x) = \sum_{m=1}^{M} \alpha_m k(x, y_m), \; g(x) = \sum_{s=1}^{S} \beta_s k(x, y_s) \in H \tag{4}$$

in H as

$$\langle f, g \rangle_H = \sum_{m=1}^{M} \sum_{s=1}^{S} \alpha_m \beta_s k(y_m, x_s). \tag{5}$$

Show that the defined inner product is valid. That is, it satisfies the symmetry, linearity and positive-definiteness properties stated below: For any for any $f, g, h \in H$ and any $a \in \mathbb{R}$, we have

- $\langle f, g \rangle_H = \langle g, f \rangle_H$.
- $\langle af, g \rangle_H = a\langle f, g \rangle_H$ and $\langle f + h, g \rangle_H = \langle f, g \rangle_H + \langle h, g \rangle_H$.
- $\langle f, f \rangle_H \geq 0$; $\langle f, f \rangle = 0$ if and only if f is an constant zero function.

What is the norm of the function f? (The natural norm in an inner product space is defined as $\|f\|_H = \sqrt{\langle f, f \rangle_H}$.)

(b) **Show that the defined inner product has the reproducing property** $\langle k(x, \cdot), k(y, \cdot) \rangle_H = k(x, y)$, where we take $h : \cdot \to k(x, \cdot)$ and $g : \cdot \to k(y, \cdot)$ as two functions in H. In other words, the inner-product is natural for the vector space as defined. **Conclude that**

$$\langle k(\cdot, x_i), f \rangle_H = f(x_i). \tag{6}$$

(For those who have taken signal processing courses, you can see here that what this family of definitions is trying to do is to parallel the example of the sifting property that we know from signal processing.)

(c) A *Hilbert space S* is defined to be a (possibly infinite dimensional) complete vector space with a defined norm. We extend H to be a complete metric space, such that this new space (which we will call H for simplicity) is a Hilbert space, as we defined its norm above.

We assume we have the following knowledge of a Hilbert space: Suppose M is a finite-dimensional subspace of a Hilbert space H. Then any element f in the full space H has a unique representation as the sum of an element of M and an element that is orthogonal to any element in M. That is,

$$f = m + g, \tag{7}$$

for some $m \in M$ and some g such that $\langle m', g \rangle_H = 0$ for all $m' \in M$. (In other words, it behaves exactly like the vector spaces that you are used to.)

Below we introduce a general optimization problem over the Hilbert space H. We will see many kernelized machine learning algorithms, including kernel SVM, can be written in the following form: Given a data set with N points $x_i, y_i, i = 1, \ldots, N$, and some $\lambda > 0$, the optimization problem is

$$\min_{f \in H} \frac{1}{N} \sum_{i=1}^{N} L(y_i, f(x_i)) + \lambda \|f\|_H^2, \tag{8}$$

where L is any loss function on pairs of real numbers. (Remember, the y_i here in this part are real numbers. They are not training points in d-dimensional space. Those are the x_i.)

Show that the minimizing solution to the problem has the form

$$f(x) = \sum_{i=1}^{N} \alpha_i k(x, x_i).$$

That is, the solution can be expressed as a weighted sum of kernel functions based on the training points. This phenomenon that reduces an infinite-dimensional optimization problem to be finite-dimensional is called the *kernel property*. Hint: Define $M = \{\sum_{n=1}^{N} \alpha_n k(x, x_n) : \alpha_i \in \mathbb{R}\}$ to be the subspace of interest.

(d) The kernel SVM defines the loss function L concretely as a Hinge loss:

$$L(y, f(x)) = \max(0, 1 - yf(x)). \tag{9}$$

for any $f \in H$. In other words, the solution for the kernel SVM is

$$\min_{f \in H} \frac{1}{N} \sum_{i=1}^{N} \max(0, 1 - y_i f(x_i)) + \lambda \|f\|_H^2. \tag{10}$$

Show kernel SVM is of the form

$$\min_{\alpha \in \mathbb{R}^d} \frac{1}{N} \sum_{i=1}^{N} \max(0, 1 - y_i \sum_{j=1}^{N} \alpha_j k(x_i, x_j)) + \lambda \alpha^T K \alpha. \tag{11}$$

2 ℓ_1-Regularization

Learning Goal: Derive a mathematical understanding of how ℓ_1-regularization encourages feature selection.

The ℓ_1-norm is one of the popular regularizers used to enhance the robustness of regression models. As you learned from lecture, regression with an ℓ_1-penalty is referred to as the LASSO regression, and it is known for promoting 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 centered and whitened 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.)*

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 $\widehat{w}_i = 0$? Explain how this shows ℓ_1-regularization is adapted for feature selection.

3 Decision Trees for Classification

In this problem, you will implement decision trees and random forests for classification on the spam dataset, which is attached to the assignment .zip. 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. You do not have to implement boosting, though it might help improve your model performance.

3.1 Implement Decision Trees

See the Appendix for more information. You are not allowed to use any off-the-shelf decision tree implementation. Be aware that some of the later questions might require special functionality that you need to implement (e.g., max depth stopping criterion, visualizing the tree, tracing the path of a sample through the tree). You can use any programming language you wish as long as we can read and run your code with minimal effort. In this part of your writeup, **include your decision tree code.**

(Optional) Implementation suggestions:

1. Implementing a stopping criteria is essential in preventing overfitting. Here are several candidates when considering stopping criteria:
   - Limited depth: don't split if the node is beyond some fixed depth in the tree
   - Node purity: don't split if the proportion of training points in some class is suciently high
   - Information gain criterion: don't split if the gained information/purity is sufficiently close to zero
2. Improving run time:
   - Vectorize using numpy functions.
   - We recommend investigating how we can compute which feature to split at any given node in $O(n'd)$ time, where d is the number of features and n' is the number of training points we have at any given node. *Hint:* How can we update entropy quickly by keeping track of what values we have seen already?

3.2 Implement Random Forests

You are not allowed to use any off-the-shelf random forest implementation. If you architected your code well, this part should be a (relatively) easy encapsulation of the previous part. In this part of your writeup, **include your random forest code.**

3.3 Performance Evaluation

Train both a decision tree and random forest and report your best training and validation accuracies. You should be reporting 4 numbers (2 classifiers $\times$ training/validation). Be careful not to overfit to the validation set. For reference, you should exceed 70% (although we will dock points if your scores are below this).

3.4 Writeup Requirements

1. (Optional) If you use any other features or feature transformations, explain what you did in your report. You may choose to use something like bag-of-words. You can implement any

custom feature extraction code in `featurize.py`, which will save your features to a `.mat` file.

2. For your decision tree, and for a data point of your choosing from each class (spam and ham), state the splits (i.e., which feature and which value of that feature to split on) your decision tree made to classify it. An example of what this might look like:

   (a) ("viagra") ≥ 2

   (b) ("thanks") < 1

   (c) ("nigeria") ≥ 3

   (d) Therefore this email was spam.

   (a) ("budget") ≥ 2

   (b) ("spreadsheet") ≥ 1

   (c) Therefore this email was ham.

3. Generate a random 80/20 training/validation split. Train decision trees with varying maximum depths (try going from depth = 1 to depth = 40) with all other hyperparameters fixed. For this problem, please include: (1) A plot of your validation accuracies as a function of the depth. (2) A brief write-up describing which depth had the highest validation accuracy and your explanation of the behavior you observe in your plot. If you find that you need to plot more depths, feel free to do so. For reference, our implementation was able to run in 2 minutes. If your method is taking much longer than this, we recommend you attempt to implement some of the run-time suggestions in section 3.1.

4. Train a very 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. You can use any visualization method you want, from simple printing to an external library; the `rcviz` library on github works well.

A Appendix

A.1 Suggested Architecture

This is a complicated coding project. You should put in some thought about how to structure your program so your decision trees don't end up as horrific forest fires of technical debt. Here is a rough, **optional** spec that only covers the barebones decision tree structure. This is only for your benefit—writing clean code will make your life easier, but we won't grade you on it. There are many different ways to implement this.

Your decision trees ideally should have a well-encapsulated interface like this:

```
classifier = DecisionTree(params)
classifier.train(train_data, train_labels)
predictions = classifier.predict(test_data)
```

where `train_data` and `test_data` are 2D matrices (rows are data, columns are features).

A decision tree (or **DecisionTree**) is a binary tree composed of **Nodes**. You first initialize it with the necessary parameters (which depend on what techniques you implement). As you train your tree, your tree should create and configure **Nodes** to use for classification and store these nodes internally. Your **DecisionTree** will store the root node of the resulting tree so you can use it in classification.

Each **Node** has left and right pointers to its children, which are also nodes, though some (like leaf nodes) won't have any children. Each node has a split rule that, during classification, tells you when you should continue traversing to the left or to the right child of the node. Leaf nodes, instead of containing a split rule, should simply contain a label of what class to classify a data point as. Leaf nodes can either be a special configuration of regular **Nodes** or an entirely different class.

Node fields:

- `split_rule`: A length 2 tuple that details what feature to split on at a node, as well as the threshold value at which you should split. The former can be encoded as an integer index into your data point's feature vector.
- `left`: The left child of the current node.
- `right`: The right child of the current node.
- `label`: If this field is set, the **Node** is a leaf node, and the field contains the label with which you should classify a data point as, assuming you reached this node during your classification tree traversal. Typically, the label is the mode of the labels of the training data points arriving at this node.

DecisionTree methods:

- `entropy(labels)`: A method that takes in the labels of data stored at a node and compute the entropy for the distribution of the labels.

- `information_gain(features, labels, threshold)`: A method that takes in some feature of the data, the labels and a threshold, and compute the information gain of a split using the threshold.
- `entropy(label)`: A method that takes in the labels of data stored at a node and compute the entropy (or Gini impurity).
- `purification(features, labels, threshold)`: A method that takes in some feature of the data, the labels and a threshold, and compute the drop in entropy (or Gini impurity) of a split using the threshold.
- `segmenter(data, labels)`: A method that takes in data and labels. When called, it finds the best split rule for a **Node** using the entropy measure and input data. There are many different types of segmenters you might implement, each with a different method of choosing a threshold. The usual method is exhaustively trying lots of different threshold values from the data and choosing the combination of split feature and threshold with the lowest entropy value. The final split rule uses the split feature with the lowest entropy value and the threshold chosen by the segmenter. *Be careful how you implement this method!* Your classifier might train very slowly if you implement this poorly.
- `train(data, labels)`: Grows a decision tree by constructing nodes. Using the entropy and segmenter methods, it attempts to find a configuration of nodes that best splits the input data. This function figures out the split rules that each node should have and figures out when to stop growing the tree and insert a leaf node. There are many ways to implement this, but eventually your DecisionTree should store the root node of the resulting tree so you can use the tree for classification later on. Since the height of your DecisionTree shouldn't be astronomically large (you may want to cap the height—if you do, the max height would be a hyperparameter), this method is best implemented recursively.
- `predict(data)`: Given a data point, traverse the tree to find the best label to classify the data point as. Start at the root node you stored and evaluate split rules at each node as you traverse until you reach a leaf node, then choose that leaf node's label as your output label.

Random forests can be implemented without code duplication by storing groups of decision trees. You will have to train each tree on different subsets of the data (data bagging) and train nodes in each tree on different subsets of features (attribute bagging). Most of this functionality should be handled by a random forest class, except attribute bagging, which may need to be implemented in the decision tree class. Hopefully, the spec above gives you a good jumping-off point as you start to implement your decision trees. Again, it's highly recommended to think through design before coding.

Happy hacking!

B Submission Instructions

Please submit

- a PDF write-up containing your *answers, plots, and code* to Gradescope
- a .zip file of your *code* and a README explaining how to run your code to Gradescope