HMC CS 158, Fall 2017
Problem Set 7 Programming: Multiclass Classification, Bagging

Goals:

- To open up the "black-box" by implementing multiclass classification and bagging.
- To investigate how output codes and loss functions affect multiclass classifier performance.
- To investigate how bagging compares to random forests and how random forests can be used for feature selection.
- To investigate whether boosted classifiers overfit.

For this assignment, you can work individually though you are encouraged to work with a partner. You should sign up for partners on Canvas (People → PS7 Groups). If you are looking for a partner, try Piazza. If, after trying Piazza, you are having trouble finding a partner, e-mail Jessica.

Submission

You should submit any answers to the exercises in a single file `writeup.pdf`. This writeup should include your name and the assignment number at the top of the first page, and it should clearly label all problems. Additionally, cite any collaborators and sources of help you received (excluding course staff), and if you are using slip days, please also indicate this at the top of your document.

Your code should be commented appropriately (though you do not need to go overboard). The most important things:

- Your name and the assignment number should be at the top of each file.
- Each class and method should have an appropriate doctsring.
- If anything is complicated, it should include some comments.

There are many possible ways to approach the programming portion of this assignment, which makes code style and comments very important so that staff can understand what you did. For this reason, you will lose points for poorly commented or poorly organized code.

When you are ready to submit, make sure that your code compiles and remove any debugging statements. Then rename the top-level directory as `<username1>_<username2>_ps7`, with the usernames in alphabetical order (e.g. `hadas_yjw_ps7`). This directory should include the electronic version of your writeup and main code, any files necessary to run your code (including any code and data provided by staff), and follow the same structure as the assignment directory provided by staff. So, for this assignment, your directory should have the following structure:

- `<username1>_<username2>_ps7/`
  - `data/`
    - `soybean_train.csv`
    - `soybean_test.csv`
    - `R1.csv`
    - `R2.csv`
    - `cancer_train.csv` (extra credit)
    - `cancer_test.csv` (extra credit)
  - `source/`
    - `soybean.py` (with your modifications)

Parts of this assignment are adapted from course material by Jenna Wiens (UMich) and Tommi Jaakola (MIT).

    * `digits.py` (with your modifications)
    * `boosting.py` (extra credit; with your modifications)
    * `util.py`
  - `writeup.pdf`
  - `soybean.pdf` (pdf printout of `soybean.py`)
  - `digits.pdf` (pdf printout of `digits.py`)
  - `boosting.pdf` (extra credit; pdf printout of `boosting.py`)

Package this directory as a single `<username1>_<username2>_ps7.zip` file, and submit the archive. Additionally, to aid the staff in grading, submit your `pdf`'s as separate files.

1 Soybean Multi-class Classification [12 pts]

In this problem, we are going to apply SVMs to the multi-class classification problem of predicting soybean diseases. We are given the "soybean dataset"[1], where inputs $\boldsymbol{x}$ contain 35 attributes, and the labels y can be one of 15 classes. After removing the examples with missing attributes, the dataset has 262 training examples and 296 test examples.

---

code and data

- code : `soybean.py`
- data : `soybean_train.csv`, `soybean_test.csv`
- data (random output codes) : `R1.csv`, `R2.csv`

documentation

- SVC (SupportVectorClassifier) :
  http://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html

---

Given an output code R and a set of discriminant functions $h_{\boldsymbol{\theta}}(\boldsymbol{x}) = [h_1(\boldsymbol{x}), ..., h_\ell(\boldsymbol{x})]$, the output label of the classification can be computed by the decision rule

$$\hat{y} = \arg\min_y \sum_{i=1}^{\ell} \text{Loss}(R(y,i)h_i(\boldsymbol{x})).$$

$\text{Loss}(R(y,i)h_i(\boldsymbol{x}))$ is usually a monotonically decreasing function $g(z)$, where $z = R(y,i)h_i(\boldsymbol{x})$. The choice of the output code and the loss function may potentially affect the performance of the multi-class classification.

In this problem, we will investigate three types of output codes:

- the "one-versus-all" output code, where all columns have exactly one $+1$ and the rest -1,
- the "one-versus-one" output code, where each column has exactly one $+1$ and one -1 (the rest is 0),
- the "random" output code, where each element in the code has a one-half probability of being $+1$ or -1 (subject to some additional constraints).

[1] http://archive.ics.uci.edu/ml/datasets/Soybean+(Large)

In addition, we will investigate three types of loss functions:

- the Hamming loss function with $g(z) = \frac{1-\text{sign}(z)}{2}$,
- the sigmoid loss function with $g(z) = \frac{1}{1+\exp(\alpha z)}$,
- the logistic loss function with $g(z) = \log(1 + \exp(-\alpha z))$,

where α is a positive scaling constant.

As we want the comparisons to focus on the choice of output code and loss function, we will train all the SVMs using a 4^{th}-degree polynomial kernel [$K(\boldsymbol{u}, \boldsymbol{v}) = (1+\boldsymbol{u}\cdot\boldsymbol{v})^4$] and slack penalty $C = 10$.

We have provided the function `generate_output_codes(...)` to construct the "one-versus-all" and "one-versus-one" output codes for you. In addition, we have provided two random output codes that you can load through `load_code(...)`.[2]

Look over the `MulticlassSVM` class, which utilizes `scikit-learn`'s `SVC` class to enable multiclass classification.[3] In particular, `MulticlassSVM` keeps track of the output code `R`, the individual (binary) classifiers `svms`, and the multiple classes `classes`. We have already implemented `MulticlassSVM.__init__(...)`, which sets up the multiple classifiers (but does not learn them).

(a) **(1 pts)** On a single graph, sketch or plot the three loss functions $g(z)$ for $z \in [-2, 2]$, with $\alpha = 1, 2$ for the sigmoid and logistic loss functions (so, 5 functions in total). Include this plot in your writeup.

(b) **(4 pts)** Implement `MulticlassSVM.fit(...)`, which learns the component binary classifiers given the training data, and `MulticlassSVM.predict(...)`, which makes predictions on the data using the decision rule with the associated loss function.

(c) **(5 pts)** Now compare the different output codes and loss functions:

Number of Errors on Test Data

		loss function		
		Hamming	sigmoid	logistic
output code	one-versus-all			
	one-versus-one			
	random (R1)			
	random (R2)			

i. **(2 pts)** Train a multiclass classifier using the "one-versus-all" output code. Then evaluate its performance on the test data using each of the three different loss functions, and report your results.

Why might Hamming loss *not* be a good loss function for the "one-versus-all" output code? *Hint*: For each error, look at $\sum_{i=1}^{l} \text{Loss}(R(y, i)h_i(\boldsymbol{x}))$ for each class y.

[2]For those interested in the details, for the random codes, we impose the constraint that there should be no identical rows and columns in R and that each column of R should have at least one $+1$ and one -1. To find a good random code, we first generate a large set of random codes $R_t, t = 1, \ldots, T$, then pick the codes with largest minimum code word distance $\rho_t = \min_{i,j} d_H(R_t(i), R_t(j))$, where $d_H(R_t(i), R_t(j))$ denotes the Hamming distance between row vectors $R_t(i)$ and $R_t(j)$ of the code matrix. We have provided two random codes by picking the top 50 codes out of 10k.

[3]You may notice that `SVC` already provides for multiclass classification. However, users are limited to one-versus-one and cannot specify the loss function.

ii. **(2 pts)** Train a multiclass classifier using the "one-versus-one" output code. Compare the performance on the test data with your results above, for all three loss functions. Which output code performs better for this problem? Why?

iii. **(1 pts)** Load the two "random" output codes, then train multiclass classifiers for each of the random output codes, and compare the performance on the test data for all three loss functions. How do the different loss functions affect the classification results, and why? (You may want to refer back to your plots of the various loss functions.)

(d) **(2 pts)** What kind of output code do you think is most suitable for this problem? Why do you think this is the case?

2 Digit Recognition using Bagging [8 pts]

In this problem, we will study bagging and random forest on a handwritten digit dataset. This dataset contains 1797 samples (each having 64 features) of hand-written digits, classified into 10 classes.[4] However, for this problem, we will only use 720 examples from 4 classes.

---

code and data

- code : `digits.py`
- data : `digits` (provided by `scikit-learn`)

documentation

- dataset : http://scikit-learn.org/stable/modules/generated/sklearn.datasets.load_digits.html
- DecisionTreeClassifier : http://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html

---

(a) **(2 pts)** Implement `bagging_ensemble(...)`, which generates a bagging ensemble classifier, where each base classifier is trained independently on a bootstrap sample of the training data.

Note: If you have trouble implementing `bagging_ensemble(...)`, you may use `sklearn.ensemble.BaggingClassifier`. You will not receive any credit for this question, but you can receive full credit for the analysis below.

(b) **(1 pts)** Implement `random_forest(...)`. Random forest is similar to bagging except that for each tree, we select a random subset of the features for training.

Hint: `scikit-learn`'s `DecisionTreeClassifier` class allows you to specify `max_features`, which restricts the number of features to consider at each split (to a random subset). Therefore, you can implement `random_forest(...)` in one-line by modifying `bagging_ensemble(...)` to take in an additional parameter `max_features` equal to the number of features to consider.

Note: If you have trouble implementing `random_forest(...)`, you may use `sklearn.ensemble.RandomForestClassifier`. You will not receive any credit for this question, but you can receive full credit for the analysis below.

[4] If you are interested in learning more about this dataset, you can always look up the documentation for `load_digits(...)`. Here, you will find that each example consists of an 8×8 grayscale image (yielding the 64 features), and you can learn how to visualize the examples.

(c) **(3 pts)** Run `main(...)` to compare the performance of these ensemble classifiers using 100 random splits of the digits dataset into training and test sets, where the test set contains roughly 20% of the data. This generates two plots :

- We vary the number of features considered for the random forest from 1 to 65 (in step sizes of 2). Then, on a single plot, we plot the accuracy of the bagging classifier against that of the random forest with varying number of features. (Note: This code may take several minutes to run.)
- We run the random forest with 8 features considered per split. Then, on a single plot, we plot the accuracies of the two ensemble classifiers as histograms.

Include these plots in your writeup. How does performance vary with the number of features considered per split? Which classifier performs better, and why?

(d) **(2 pts)** One useful application of bagging ensembles is to assess the relative importance of features. In your own words, describe how you think feature importance might be calculated when using Decision Trees as your base classifier.

Now, using `scikit-learn`'s examples[5] as a guideline, generate a representation of the relative importance of each individual pixel for digit recognition (include this plot in your writeup). For this problem, you are allowed to use `sklearn.ensemble.RandomForestClassifier`.

What do you observe? Is this surprising?

3 Boosting [+2 pts]

It seems that the boosting algorithm might seriously overfit. Let us see if this is true in practice.

---

code and data

- code : `boosting.py`
- data : `cancer_train.csv`, `cancer_test.csv`

documentation

- GradientBoostingClassifier:
  http://scikit-learn.org/stable/modules/generated/sklearn.ensemble.GradientBoostingClassifier.html

---

We have provided a small dataset of 38 training examples and 34 test examples, where each example has two features and a label for whether the patient has cancer or not. In this problem, we will use scikit-learn's `GradientBoostingClassifier` with `loss = 'exponential'` to train an AdaBoost ensemble with decision stumps as the base learners.

(a) **(+1 pts)** Complete `boosting.py` based on the specification given in the starter code. In brief, the code should train an ensemble model, compute the number training and test errors (y-axis) with respect to the number of ensembles m (x-axis), and finally plot the number of training errors (blue line) and the number of test errors (red line) as a function of the number of ensembles. What do you observe about this plot? Does the model overfit?

[5] http://scikit-learn.org/stable/modules/ensemble.html#feature-importance-evaluation

(b) **(+1 pts)** Now let us try to better understand what the boosting algorithm is doing. By definition, example i has a margin error at level ρ if $y^{(i)}h_m(\boldsymbol{x}^{(i)}) - \rho \leq 0$. Using the first m decision stumps in the ensemble, vary $\rho \in [0, 1]$ (x-axis), and compute the number of training examples with a margin error of ρ (y-axis). How do the margin errors change as we increase m? Use these plots explain your observations in the previous question. *Note*: Take a close look at $\rho = 0.1$ on the plots.