Problem Set 4: Supervised Learning

**Handed out:** Wednesday, November 18, 2015

**Due:** 11:59pm, Friday, December 4, 2015

The goal of this problem set is to offer you an opportunity to apply real machine learning tools to real data. In particular, we’re going to use the *scikit-learn* implementation of *logistic regression* to predict whether tumors are benign or malignant. We’ll be using the *Wisconsin Diagnostic Breast Cancer* dataset (you can find more info in the file WDBC.txt).

The pset consists of five major sections exploring four core topics in machine learning: data preparation, supervised learning, evaluation, and model improvement.

* In Part 1 you will learn about data pre-processing, implementing functions to partition the dataset into training and evaluation subsets.
* Part 2 deals with the actual training of a model using a *supervised* approach, as well as using the model to make predictions and score labeled evaluation data.
* For Part 3 you will implement various performance metrics and write functions for generating plots.
* Part 4 deals with a particular approach -- referred to as *feature reduction* or *feature elimination* -- for systematic model improvement.
* In Part 5 you will write the high-level control for running experiments with this data and methodology.

Have fun with this problem set! Machine learning, and supervised learning in particular, can be an astonishingly powerful tool. We hope that through this hands-on experience you can appreciate the opportunities -- and limitations -- presented by this paradigm.

NOTE: Start early!! This is a long pset. We’ve included some tests for the some of the earlier problems, and will release a ‘checkpoint’ package about half way between the pset’s release and due dates.

# Part 0 - Installation

You will make use of scikit-learn, a Python library for machine learning algorithms. You can check if you have it in your Anaconda installation by typing import sklearn in the Spyder console. If this fails, try installing the library by typing conda install scikit-learn in the command line, or refer to this link:

<http://scikit-learn.org/stable/install.html>

Note that scikit-learn should be included in your Anaconda installation. If you’re not using Anaconda and have problems installing scikit-learn with another Python distribution, we might not be able to help you.

# Part 1 - Data Prep

An important part of every experiment is pre-processing the data. This can include many different steps. The first step is often just reading the data files into some data structure. We have provided code for doing that in the helper section. You do not need to change this code, but it will be useful for you to understand how it works.

In particular, readData reads data from a file and getTumorData creates Tumor instances of these data. In this part, you may ignore the dontUse argument of getTumorData. You will come back to that in Part 4.

Often one needs to interfere with the data in more sophisticated ways. For example, adding noise is sometimes necessary for robustness purposes. In our case, we chose to corrupt the data by adding some noise to the features. We’ve included the function corruptData in the helper section. You do not need to change anything in this function, but it will be useful for you to understand how it works.

Now that you know a bit about pre-processing, let’s turn to your part - data set partitioning!

An important pre-processing step is partitioning the data into train and test sets. The train set is used for training the model, while the test set is used for testing and evaluating the performance of the trained model. Crucially, the test set must not be used during training. It can only be used in the final evaluation step. In other words, it is held out in all previous steps.

In order to facilitate development of different models without compromising the test set, oftentimes the training set itself is also split into two sets, say, training and *development*. Then all the experimentation takes place by training on the new training set and testing on the development set, obtaining some performance measure, improving the model, training again, testing on the development set again, and so on.

In this part, you will implement splitData, a function for splitting a data set into training and test sets, given a certain holdout fraction. For robustness, this division should be done at random. In other words, you don’t want to just take the first k data points for training and the rest for testing; instead the partitions should be uniformly random, where every data point has an equal probability (holdoutFrac) of being put into the test set.

# Part 2 - Supervised Learning: Logistic Regression

At the heart of any machine learning approach is a model. The model has parameters which are set to fit some dataset, often referred to as *training* the model or *learning* the parameters. Once parameters have been learned, the model can be used to make predictions about new data. In this problem set we’re going to be using the logistic regression model, which is a classic model used for *supervised learning*.

Specifically, we want to classify tumors as either benign or malignant, where a given tumor is represented by a set of 30 features, or attributes, measured by analyzing a sample of its cells. We are going to train a logistic regression model by training it on a set of feature vectors/arrays, one per tumor, paired with their corresponding labels (benign or malignant). The fact that the model is trained with *labeled* data makes this a supervised learning scenario.

## Problem 2.1

Write a function trainModel which takes as argument a list of Tumor instances, generates the data matrix and labels array, and fits the model to the data. The function should return the trained model. We are going to leverage the publicly available and widely used scikit-learn library to do most of the heavy lifting for us. Take a look at the class [sklearn.linear\_model.LogisticRegression](http://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html) and the method [fit(X,y)](http://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html#sklearn.linear_model.LogisticRegression.fit). Note that fit must be called on an *instance* of LogisticRegression. Also look at the getFeatures and getLabel methods of the Tumor class.

By *data matrix* we mean a 2D array where each row corresponds to a tumor and each column to a feature. A *labels array* is an array with an entry for each tumor (number of rows in the data matrix), such that the value of that entry is a 1 if the corresponding tumor is malignant or 0 if it is benign. If X is our data matrix, X[i][j] is the ith tumor’s jth feature value. If y is the corresponding label array, y[i] is the label for the ith tumor.

## Problem 2.2

Write a function predictLabels which takes as argument a trained logistic regression model, a threshold value between 0 and 1, and a list of Tumor instances. For each tumor, the function should use the model to estimate the probability that the tumor should have label 1 (malignant). If the probability is greater than the specified threshold, the label should be predicted 1, otherwise it should be predicted 0. The function should return a list with the labels of all the data points (in the corresponding order).

Be sure to refer to the method [predict\_proba(X)](http://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html#sklearn.linear_model.LogisticRegression.predict_proba) from the LogisticRegression class. This function gives the probability of each label for a given tumor (represented as a feature array). In fact, we can call predict\_proba with an entire collection of tumors such that it tells us the label probabilities for each one. We encode the tumors in a data matrix X, as explained above in problem 2.1. The output of the method is a 2D array, with one row per tumor and two columns; namely, predict\_proba(X)[i][0] and predict\_proba(X)[i][1] give the probabilities that tumor i is benign and malignant, respectively.

# Part 3 - Evaluation

Every machine learning experiment involves some sort of evaluation. In this part, you will implement quantitative evaluation metrics for assessing the quality of your supervised learning algorithms. Our metrics rely on the concept of true/false positive/negative, as defined below. Assuming a binary classification problem into positive and negative classes (in our case 1 and 0, respectively), an automatic system may predict a given item to be positive and be correct or incorrect. In the first case, we count this item as a true positive; in the latter, as a false positive. More formally, we define:

* True positive: the number of positive elements correctly identified as positive.
* False positive: the number of negative elements incorrectly identified as positive.
* True negative: the number of negative elements correctly identified as negative.
* False negative: the number of positive elements incorrectly identified as negative.

## Problem 3.1

Write a function scoreTestSet. It should take as arguments a trained model, a threshold, and a list of Tumor instances. The function should predict the labels for the tumors, and return the counts of true/false positive/negative predictions by comparing the predicted labels to the true labels (those returned by calls to getLabel for each Tumor instance).

## Problem 3.2

Based on the above definitions, you will implement the following evaluation metrics:[[1]](#footnote-0)

1. **Accuracy**: the fraction of true negative and positive out of all elements. This is also the fraction of correctly identified elements.
2. **Sensitivity**: the fraction of correctly identified positive elements out of all positive elements. This is also called *recall* or *true positive rate* in some fields.
3. **Specificity**: the fraction of correctly identified negative elements out of all negative elements.
4. **Positive predictive value**: the fraction of correctly identified positive elements out of all positively identified elements. This is also called *precision* in some fields.

Implement the four evaluation metrics according to these definitions and the instructions in the doc strings. Remember to test your functions with some simple values.

**Note**: some of these metrics require division. If doing so will cause a ZeroDivisionError, you should return a *nan*, a special Python value. You can do so by returning float(‘nan’)

## Problem 3.3

Recall that we had a threshold parameter *p* (also called hyperparameter), controlling the prediction probability above which we classify an example as positive. Varying this parameter might lead to quite different performances on different evaluation metrics. In order to evaluate this variability, one often plots a curve known as the ROC curve.[[2]](#footnote-1) This curve details the true positive rate (on the y axis) vs the false positive rate (on the x axis).

The true positive rate is simply the sensitivity that you have already implemented.

What is the false positive rate? It is simply 1 - specificity. Think why!

The ROC curve is then achieved by varying the value of *p*, applying the trained model at each value, and recording these rates.

To obtain a quantitative measure of the ROC curve, we calculate the area under the curve. In general it’s not entirely clear how to do that. Fortunately, scikit-learn provides a convenient method, which you can call like so:

[sklearn.metrics.auc(x, y, reorder=False)](http://scikit-learn.org/stable/modules/generated/sklearn.metrics.auc.html)

Please use True as the value of the third argument, although this shouldn’t matter too much in practice.

Implement the buildROC function according to these instructions and the doc string.

## Problem 3.4

In Part 1 we mentioned the deliberate corruption of the data as a means of making the model more robust. We want to see how the performance of the model changes as we vary the corruption rate. Implement a function plotPerfVsCorruption that takes as argument the name of the file containing the original, uncorrupted data, a holdout fraction, and a threshold probability. The function should read the clean data, and for each corruption fraction on the interval [0,1], in increments of 0.05, it should generate a corrupted version of the data, train and evaluate a model on the corrupted data, and plot the accuracy (y-axis) of the model for each corruption rate (x-axis).

Make use of the functions readData, corruptData, and getTumorData, as well as the functions you’ve implemented so far.

# Part 4 - Model Improvement: Feature Reduction

Systematic model improvement is an important aspect of data science. Feature reduction is one approach that can result in significant gains and is relatively simple to implement. The idea is that correlated features can often, counterintuitively, mislead a model or cause it to overfit. Even when they’re not detrimental to accuracy, redundant features offer little useful information at the cost of more computation (more features generally means more computation, sometimes a **lot** more).

In light of this, feature reduction attempts to find a subset of features that perform optimally. One efficient way to do this is to iteratively identify the feature that most improves model accuracy when eliminated, remove it, and repeat until accuracy ceases to improve. This *greedy* approach is computationally much cheaper than trying every possible feature subset yet performs quite well.

## Problem 4.1

Implement the function findBestFeatureToEliminate, which takes as arguments the training data as a list of strings (one string per data point), a list of features to omit, and a threshold probability. The function has to identify the remaining feature (excluding those in features\_to\_omit) that most improves the model accuracy when excluded. It should return the name of said feature as well as the best performing model.

If no feature exists that improves the model accuracy, the value None should be returned as the feature name and the returned model should be the one trained with all the features (excluding those in features\_to\_omit).

The training data should be partitioned into a training set and a development set (80% and 20% respectively), such that models are trained on the training set and accuracy is evaluated on the development set.

## Problem 4.2

The part that remains is to greedily eliminate features, starting from the full feature set, until we can’t improve the model. Write a function buildReducedModel that accomplishes that, returning the model trained on the best reduced set of features. The function takes the training data as a list of strings as well as a probability threshold. This function should only be a few lines in length.

# Part 5 - Running the Experiment

You are now ready to run the full experiment. You will need to implement a function that reads data from a file, corrupts it if necessary, trains a model, and test it. You will also do some feature reduction.Your code in this part should mostly just call all the functions you implemented in the previous parts. The following is a list of required steps:

1. Read data from file file\_name.
2. Corrupt data according to corrupt\_frac fraction.
3. Split data into training and test according to holdout\_frac.
4. Train a full model on the training set using all features.
5. Evaluate accuracy of the full trained model on the training and test sets using the specified threshold.
6. Plot the ROC curve of the full model on the test set.
7. Plot performance vs corruption fraction by calling plotPerfVsCorruption with holdout\_frac and threshold.
8. Train a reduced model on the training set using feature reduction.
9. Evaluate accuracy of the reduced trained model on the training and test sets using the specified threshold.

After you implement this part, uncomment the execution line at the bottom of the code file and run the code.

In addition to your implemented code, your **deliverables** in this part are:

1. ROC and performance vs corruption plots of the full trained model on the test set.
2. Evaluation metrics (accuracy, sensitivity, specificity, and positive predictive values) of the full model on the training and test sets. Compare the different metrics on both data sets (training and test), and include a *brief* discussion in your write-up.
3. Evaluation metrics (same as above) of the reduced model on the training and test sets. Include a similar brief discussion in your write-up.

You should submit a short write-up with these points in addition to your code.

## 

## Sanity check

You might wonder if your model does a good job on this classification task. A good sanity check is to think of some simple *baselines*, i.e., simple systems that can be used to get baseline scores to compare with. Perhaps the simplest such systems are majority rule and random. In majority rule, one takes the majority label from the training data and always predicts it on the test data. In our case, this will lead to ~63% accuracy. In random rule, one picks a random label at test time. In a binary classification like ours, this should lead to roughly 50% accuracy.

**Hint**: Your system should beat these baselines by a large margin.

# Submit

* **ps4.py**
* **Problem Set 4 Writeup.pdf** (should include all plots, metrics, and explanations for Part 5)

Upload all your files to Stellar. If there is some error uploading, email the file to 6.0002-staff [at] mit.edu.

You may upload new versions of each file until the 11:59pm deadline, but anything uploaded after that will be ignored, unless you still have enough late days left.

1. If you like, you may refer to the Wikipedia article for more details: <https://en.wikipedia.org/wiki/Sensitivity_and_specificity>. [↑](#footnote-ref-0)
2. The acronym stands for receiver operating characteristic, although the name is not very informative. For more details, refer to <https://en.wikipedia.org/wiki/Receiver_operating_characteristic>. [↑](#footnote-ref-1)