CS 6362 Machine Learning: Homework 1
Learning Foundations
Due: 8/30/2017 11:59PM Central Time
50 Points Total Version 1.0

Make sure to read from start to finish before beginning the assignment.

1 Introduction

The goal of the programming homeworks in this course is to build a machine learning library. In each homework assignment you will expand your learning library by implementing and evaluating new algorithms. Most assignments will build upon previous assignments by comparing algorithms on common data. The purpose of the first assignment will be to introduce the data and build the foundations of the learning library.

1.1 Components of Assignments

Assignments consist of two parts.

1. **Programming:** You will implement learning algorithms and test them on provided data.
2. **Analytical questions:** These questions will ask you to consider questions related to the topics covered by the assignment. You will be able to answer these questions without relying on your programming.

Assignments are worth various points (usually between 50 and 100) and the point totals will be indicated in the assignment.

Each assignment will contain a version number at the top. While we try to ensure every homework is perfect when we release it, small errors do happen. When we correct these, we'll update the version number, post a new PDF and announce the change. Each homework starts at version 1.0 (no beta).

2 Data

The first part of the semester will focus on supervised classification. We consider several real world binary classification datasets taken from a range of applications. Each dataset is in the same format (described below) and contains a train, development and test file. You will train your algorithm on the train file and use the development set to test that your algorithm works. The test file contains unlabeled examples that we will use to test your algorithm. It is **a very good idea** to run on the test data just to make sure your code doesn't crash. You'd be surprised how often this happens.

2.1 Biology

Biological research produces large amounts of data to analyze. Applications of machine learning to biology include finding regions of DNA that encode for proteins, classification of gene expression data and inferring regulatory networks from mRNA and proteomic data.

Our biology task of characterizing splice junction gene sequences comes from molecular biology. Splice junctions are points on a DNA sequence at which "superfluous" DNA is removed during the process of protein creation in higher organisms. Exons are sequences of DNA that are retained after splicing while introns are spliced out. The goal of this prediction task is to recognize DNA sequences that contain boundaries between exons and introns. Sequences contain exon/intron (EI) boundaries, intron/exon (IE) boundaries, or do not contain splice examples.

For a binary task, you will classify sequences as either EI boundaries (label 1) or non-splice sequences (label 0). Each learning instance contains a 60 base pair sequence (ex. ACGT), with some ambiguous slots. Features encode which base pair occurs at each position of the sequence.

2.2 Finance

Finance is a data rich field that employs numerous statistical methods for modeling and prediction, including the modeling of financial systems and portfolios.[1]

Our financial task is to predict which Australian credit card applications should be accepted (label 1) or rejected (label 0). Each example represents a credit card application, where all values and attributes have been anonymized for confidentiality. Features are a mix of continuous and discrete attributes and discrete attributes have been binarized.

2.3 NLP

Natural language processing studies the processing and understanding of human languages. Machine learning is widely used in NLP tasks, including document understanding, information extraction, machine translation and document classification.

Our NLP task is sentiment classification. Each example is a product review taken from Amazon kitchen appliance reviews. The review is either positive (label 1) or negative (label 0) towards the product. Reviews are represented as uni-gram and bi-grams; each one and two word phrase is extracted as a feature.

2.4 Speech

Statistical speech processing has its roots in the 1980s and has been the focus of machine learning research for decades. The area deals with all aspects of processing speech signals, including speech transcription, speaker identification and speech information retrieval.

Our speech task is spoken letter identification. Each example comes from a speaker saying one of the twenty-six letters of English alphabet. Our goal is to predict which letter was spoken. The data was collected by asking 150 subjects to speak each letter of the alphabet twice.

Each spoken utterance is represented as a collection of 617 real valued attributes scaled to be between -1.0 and 1.0. Features include spectral coefficients; contour features,

[1]For an overview of such applications, see the proceedings of the 2005 NIPS workshop on machine learning in finance. http://www.icsi.berkeley.edu/~moody/MLFinance2005.htm

sonorant features, pre-sonorant features, and post-sonorant features. The binary task is to distinguish between the letter M (label 0) and N (label 1).

2.5 Vision

Computer vision processes and analyzes images and videos and it is one of the fundamental areas of robotics. Machine learning applications include identifying objects in images, segmenting video and understanding scenes in film.

Our vision task is image segmentation. In image segmentation, an image is divided into segments are labeled according to content. The images in our data have been divided into 3x3 regions. Each example is a region and features include the centroids of parts of the image, pixels in a region, contrast, intensity, color, saturation and hue. The goal is to identify the primary element in the image as either a brickface, sky, foliage, cement, window, path or grass. In the binary task, you will distinguish segments of foliage (label 0) from grass (label 1).

2.6 Synthetic Data

When developing algorithms it is often helpful to consider data with known properties. We typically create synthetic data for this purpose. To help test your algorithms, we are providing two synthetic datasets. These data are to help development.

2.6.1 Easy

The easy data is labeled using a trivial classification function. Any reasonable learning algorithm should achieve near flawless accuracy. Each example is a 10 dimensional instance drawn from a multi-variate Gaussian distribution with 0 mean and a diagonal identity covariance matrix. Each example is labeled according to the presence one of 6 features; the remaining features are noise.

2.6.2 Hard

Examples in this data are randomly labeled. Since there is no pattern, no learning algorithm should achieve accuracy significantly different from random guessing (50%). Data is generated in an identical manner as *Easy* except there are 94 noisy features.

3 Programming (35 Points)

In this assignment you will build your learning framework by writing some simple binary classification algorithms. We have provided Java code that performs most basic operations for the learning library. You will fill in the details. Search for comments that begin with `TODO`; these sections need to be written. You may change the internal code as you see fit but the behavior for the given command lines cannot be changed. Do not change the name or package of any of the provided code.

3.1 How to Run the Library

The library operates in two modes: train and test. Both stages are in the main method of `Learn`.

The command line for train mode is:

```
java cs362.Learn -mode train -algorithm algorithm -model_file model_file \
                    -data train_file -task classification
```

The `mode` option indicates which mode to run (train or test). The `algorithm` option indicates which training algorithm to use. Each assignment will specify the string argument for an algorithm. The `data` option indicates the data file to load. Finally, the `model_file` option specifies where to save the trained model.

The test mode is run in a similar manner:

```
java cs362.Learn -mode test -model_file model_file -data test_file \
                    -predictions_file predictions_file -task classification
```

The `model_file` is loaded and run on the `data`. Results are saved to the `predictions_file`.

As an example, the following trains an even/odd classifier on the speech training data:

```
java cs362.Learn -mode train -algorithm even_odd -model_file speech.even_odd.model \
                     -data speech.train -task classification
```

To run the trained model on development data:

```
java cs362.Learn -mode test -model_file speech.even_odd.model -data speech.dev \
                     -predictions_file speech.dev.predictions -task classification
```

As we add new algorithms we will also add command line flags to specify algorithmic parameters. These will be specified in each assignment.

To run the code you will need both the provided library and Apache Commons CLI (version 1.0 or above) [2]. Make sure this jar file is on the classpath.

3.2 Data Formats

The data are provided in what is commonly known as SVM-light format. Each line contains a single example:

```
0 1:-0.2970 2:0.2092 5:0.3348 9:0.3892 25:0.7532 78:0.7280
```

The first entry on the line is the label. The label can be an integer (0/1 for binary classification) or a real valued number (for regression.) The classification label of -1 indicates unlabeled. Subsequent entries on the line are features. The entry `25:0.7532` means that feature 25 has value 0.7532. Features are 1-indexed.

Model predictions are saved as one predicted label per line in the same order as the input data. The code that generates these predictions is provided in the library. The script `compute_accuracy.py` can be used to evaluate the accuracy of your predictions for classification:

```
compute_accuracy.py data_file predictions_file
```

We provide this script since it is exactly how we will evaluate your output.

3.3 Components

The foundations of the learning framework have been provided for you in the skeleton library. You will need to complete this library by filling in code where you see a `TODO` comment. You are free to make changes to the code as needed provided you do not change the behavior of the command lines described above.

The classes of interest in the library are:

[2] http://commons.apache.org/downloads/download_cli.cgi

- **FeatureVector**- The data representing an instance are stored as a feature vector. A feature vector is a vector of doubles, where the value of the ith dimension of the feature vector corresponds to the value of the ith feature. A FeatureVector must support operations such as `get(index)`, which returns the value of the feature at `index` and `add(index, value)`, which sets the value of the feature at `index`.

  Since many learning applications encode instances as sparse vectors, `FeatureVector` should be **a sparse vector**. A sparse vector efficiently encodes very high dimensional data by not maintaing values for features with 0 values. Some common implementations of sparse vectors include hash maps and lists of index/value pairs. If you fail to do this correctly, your code will run very slowly. You will need to add a method(s) to iterate over the non-empty positions of the vector. How you chose to do this is up to you.

- **Label**- A label object encodes the label for a learning example. The `Label` class is abstract and you will implement a `ClassificationLabel`, which contains an int to indicate a class (binary prediction will be 0 or 1) and a `RegressionLabel`, which contains a double to indicate the value of the label. `toString()` methods must be written for each label since these are called to write the label to the predictions file.

- **Instance**- An instance represents a single learning example. An instance contains a data object and a label. The data object is a `FeatureVector` and the label will be a `Label` object. For classification, the label will be a `ClassificationLabel` object. When the label is unknown (test data) the label will be null.

- **DataReader**- This class reads in data and creates `Instance` objects.

- **Predictor**- This is an abstract class that will be the parent class for all learning algorithms. Learning algorithms must implement the `train` and `predict` methods. Predictors must be serializable so that they can be saved after training.

- **PredictionsWriter**- Writes the predictions to a file.

- **Learn**- The main method used to run the learning library. This is where `Predictor` objects are created and trained.

For those new to Java, you may find these classes helpful: HashMap, HashSet, ArrayList. Also, if you encounter serialization errors on this or future assignments, simply add `implements Serializable` to the classes that are throwing the error.

3.4 Learning Algorithms

You will test your learning library by writing some simple learning algorithms.

Majority Classifier: A majority classifier labels every object with the most common label in the training data. When two labels are tied for occurring the most often then the majority classifier picks one at random for labeling all test data. This random pick should be saved to use for labeling all of the test data. This classifier should be selected by passing the string `majority` as the argument for `algorithm`.

Even/Odd Classifier: Compute two sums: `even-sum` and `odd-sum`. `even-sum` is the sum of the values for every even numbered feature. `odd-sum` is the sum of the values for every odd numbered feature. If `even-sum` $\geq$ `odd-sum`, predict 1. Otherwise predict 0. We realize that this doesn't make much sense as a prediction rule. The goal is to demonstrate that you can successfully store and access the features. Remember, you should be using sparse feature vectors as described above. This classifier should be selected by passing the string `even_odd` as the argument for `algorithm`.

Note that for this assignment, the algorithms are not expected to do very well on any data, including the *Easy* synthetic data.

3.5 Grading Programming

The programming section of your assignment will be graded using an automated grading program. Your code will be run using the provided command line options, as well as other variations on these options (different parameters, data sets, etc.) The grader will consider the following aspects of your code.

1. **Compilation:** Does your code compile? If not, you'll fail the programming part of the assignment. Make sure your code compiles and that you submit all of your code in the proper Java directory structure (i.e., folders for each package.) Note that you **cannot** use external libraries except those given as part of the assignment.

2. **Exceptions:** Does your code run without crashing?

3. **Output:** Many assignments will ask you to write some data to the console. Make sure you follow the provided output instructions exactly.

4. **Accuracy:** If your code works correctly, then it should achieve a certain accuracy on each data set. While there are small difference that can arise, a correctly working implementation will get the right answer.

5. **Speed/Memory:** Efficiency largely doesn't matter, except where lack of efficiency severely slows your code (think so slow that we assume it is broken) or the lack of efficiency demonstrates a lack of understanding of the algorithm. For example, if your code runs in two minutes and everyone else runs in 2 seconds, you'll lose points. Alternatively, if you require 2 gigs of memory, and everyone else needs 10 MB, you'll lose points. In general, this happens not when you are not optimizing your code, but when you've implemented something incorrectly.

3.6 Submitting Code

All of the code you submit must be in a file called hw1code.zip. We assume that the root of this zip file corresponds to the default package. **Do not place code in a sub directory called src.** To compile your code, we will extract the contents in hw1code.zip and compile everything that ends in .java. The java compiler assumes that directories correspond to packages. Since we require you to submit a file called cs362.Learn then you must have a directory called cs362 that contains a file named Learn.java.

It is probably easiest if all of the code you write is in the package cs362, although you can (and probably should) create sub packages like cs362.classification. You also have the option of putting code in the default package, the root of the .zip file or anywhere else you want. The key requirements are: 1) Your code must compile assuming that the root

of the zip is the default package. 2) You must have a directory called cs362 that contains a file named Learn.java.

3.7 Code Readability and Style

In general, we do not care about code style or that it conforms to Java standards for naming variables, methods, etc. However, your code should be readable, which means comments and clear organization. If your code works perfectly then you will get full credit. However, if it does not we will look at your code to determine how to allocate partial credit. If we cannot read your code or understand it, then it is very difficult to assign partial credit. Therefore, it is in your own interests to make sure that your code is reasonably readable and clear.

3.8 Code Structure

Your code must support the command line options and the example commands listed in the assignment. Aside from this, you are free to change the internal structure of the code, write new classes, change methods, add exception handling, etc. However, do not change the name or packages of any code you have been provided. We suggest you remember the need for clarity in your code organization.

3.9 Knowing Your Code Works

How do you know your code really works? That is a very difficult problem to solve. Here are a few tips:

1. Check results on `easy` and `hard`. They should be close to 100% and 50% respectively. For this assignment though, that won't be the case. However, you can easily check if things are working by hand.

2. While you **cannot** share code with classmates, you can share results. It is acceptable to discuss your results on dev data for your different algorithms. A common result will quickly emerge that you can measure against.

3. Output intermediate steps. Looking at final predictions that are wrong tells you little. Instead, print output as you go and check it to make sure it looks right. This can also be helpful when sharing information on the bulletin board.

4. Debug. If you don't know how to use the Java Debugger, learn. It's incredibly helpful.

4 Analytical (15 Points)

In addition to completing the analytical questions, your assignment for this homework is to learn Latex. All homework writeups must be PDFs compiled from Latex. Why learn latex?

1. It is incredibly useful for writing mathematical expressions.

2. It makes references simple.

3. Many academic papers are written in latex.

The list goes on. Additionally, it makes your assignments much easier to read than if you try to scan them in or complete them in Word.

We realize learning latex can be daunting. Fear not. There are many tutorials on the Web to help you learn. We recommend using pdflatex. It's available for nearly every operating system. Additionally, we have provided you with the tex source for this PDF, which means you can start your writeup by erasing much of the content of this writeup and filling in your answers. You can even copy and paste the few mathematical expressions in this assignment for your convenience. As the semester progresses, you'll no doubt become more familiar with latex, and even begin to appreciate using it.

Be sure to check out this cool latex tool for finding symbols. It uses machine learning! `http://detexify.kirelabs.org/classify.html`

1 (3 points) Explain why you agree or disagree with the following statement.
It is always best to select a hypothesis class that contains the optimal hypothesis.

2 (3 points) True or False: *An infinite hypothesis class always contains the optimal hypothesis.* If true, why? If false, give a counter example.

3 (3 points) Consider the following development scenario. A researcher collects 1000 labeled examples for learning, dividing them into a training set (500 examples) and a test set (500 examples). To develop a classifier, the researcher experiments by adding new features. To guide feature construction, the research tests each set of features by training on the train data and testing on the test data, measuring the resulting change in accuracy and only keeping features that help. When the researcher is finished, he collects 1000 new labeled examples and evaluates the classifier on these new examples.

Do you expect accuracy on these 1000 new examples to be the same, better or worse than the original 500 test examples? Why?

4 (3 points) At the start of the semester, you arrive home to find three packages containing your three new textbooks. Each package was sent via Fedex or UPS with equal probability. Of the three packages, one of the packages is a brown box delivery by UPS. What is the probability that you have one UPS package and two Fedex packages? Why?

5 (3 points) True/False (and why): Suppose you know your hypothesis class contains the optimal hypothesis, and you observe that changing any one small part (e.g., a single weight) of your current hypothesis makes it worse than before. You can safely conclude that the current hypothesis must be optimal.

5 What to Submit

In each assignment you will submit two things.

1. **Code:** Your code as a zip file named `hw1code.zip`. **You must submit source code (.java files)**. We will run your code using the exact command lines described above, so make sure it works ahead of time. Remember to submit all of the source code, including what we have provided to you.

2. **Writeup:** Your writeup, `hw1solutions.pdf`, is a **PDF file** (compiled from latex) containing answers to the analytical questions asked in the assignment.

Make sure you name each of the files exactly as specified (`hw1code.zip` and `hw1solutions.pdf`). Submit the assignment on OAK.