

# CIS 419/519 Introduction to Machine Learning

## Assignment 1

Due: September 23, 2015 11:59pm

### Instructions

Read all instructions in this section thoroughly.

**Collaboration:** Make certain that you understand the course collaboration policy, described on the course website. You must complete this assignment **individually**; you are **not** allowed to collaborate with anyone else. You may *discuss* the homework to understand the problems and the mathematics behind the various learning algorithms, but **you are not allowed to share problem solutions or your code with any other students**. You must also not consult code on the internet that is directly related to the programming exercise. We will be using automatic checking software to detect academic dishonesty, so please don't do it. You are also prohibited from posting any part of your solution to the internet, even after the course is complete. Similarly, please don't post this PDF file or the homework skeleton code to the internet.

**Formatting:** This assignment consists of two parts: a problem set and program exercises.

For the problem set, you must write up your solutions electronically and submit it as a single PDF document. We will not accept handwritten or paper copies of the homework. Your problem set solutions must use proper mathematical formatting. For this reason, we **strongly** encourage you to write up your responses using L<sup>A</sup>T<sub>E</sub>X. (Alternative word processing programs, such as MS Word, produce very poorly formatted mathematics.) Your final project report is required to be in L<sup>A</sup>T<sub>E</sub>X, so you might as well learn it now.

Your solutions to the programming exercises must be implemented in python, following the precise instructions included in Part 2. Portions of the programming exercise will be graded automatically, so it is imperative that your code follows the specified API. A few parts of the programming exercise asks you to create plots or describe results; these should be included in the same PDF document that you create for the problem set.

**Homework Template and Files to Get You Started:** The homework zip file contains the skeleton code and data sets that you will require for this assignment. **Please read through the documentation provided in ALL files before starting the assignment.**

**Citing Your Sources:** Any sources of help that you consult while completing this assignment (other students, textbooks, websites, etc.) **\*MUST\*** be noted in the your README file. This includes anyone you briefly discussed the homework with. If you received help from the following sources, you do not need to cite it: course instructor, course teaching assistants, course lecture notes, course textbooks or other readings.

**Submitting Your Solution:** We will post instructions for submitting your solution one week before the assignment is due. Be sure to check Piazza then for details.

**CIS 519 ONLY Problems:** Several problems are marked as “[CIS 519 ONLY]” in this assignment. Only students enrolled in CIS 519 are required to complete these problems. However, we do encourage students in CIS 419 to read through these problems, although you are not required to complete them.

All homeworks will receive a percentage grade, but CIS 519 homeworks will be graded out of a different total number of points than CIS 419 homeworks. Students in CIS 419 choosing to complete CIS 519 ONLY exercises will not receive any credit for answers to these questions (i.e., they will not count as extra credit nor will they compensate for points lost on other problems).

**Acknowledgements:** Parts of the linear regression exercise has been adapted from course materials by Andrew Ng.

# PART I: PROBLEM SET

Your solutions to the problems will be submitted as a single PDF document. Be certain that your problems are well-numbered and that it is clear what your answers are. Additionally, you will be required to duplicate your answers to particular problems in the **README** file that you will submit.

## 1 Decision Tree Learning (15 pts)

The following table gives a data set for deciding whether to play or cancel a ball game, depending on the weather conditions.

| Outlook  | Temp (F) | Humidity (%) | Windy? | Class      |
|----------|----------|--------------|--------|------------|
| sunny    | 75       | 70           | true   | Play       |
| sunny    | 80       | 90           | true   | Don't Play |
| sunny    | 85       | 85           | false  | Don't Play |
| sunny    | 72       | 95           | false  | Don't Play |
| sunny    | 69       | 70           | false  | Play       |
| overcast | 72       | 90           | true   | Play       |
| overcast | 83       | 78           | false  | Play       |
| overcast | 64       | 65           | true   | Play       |
| overcast | 81       | 75           | false  | Play       |
| rain     | 71       | 80           | true   | Don't Play |
| rain     | 65       | 70           | true   | Don't Play |
| rain     | 75       | 80           | false  | Play       |
| rain     | 68       | 80           | false  | Play       |
| rain     | 70       | 96           | false  | Play       |

(a) (5 pts.) At the root node for a decision tree in this domain, what are the information gains associated with the Outlook and Humidity attributes? (Use a threshold of 75 for humidity (i.e., assume a binary split:  $\text{humidity} \leq 75$  /  $\text{humidity} > 75$ ). Be sure to show your computations.

(b) (5 pts.) Again at the root node, what are the gain ratios associated with the Outlook and Humidity attributes (using the same threshold as in (a))? Be sure to show your computations.

(c) (5 pts.) Draw the complete (unpruned) decision tree, showing the class predictions at the leaves.

## 2 [CIS 519 ONLY] Decision Trees & Linear Discriminants (10 pts)

Describe **in detail** how to modify a classic decision tree algorithm (ID3 / C4.5) to obtain oblique splits (i.e, splits that are not parallel to an axis).

## PART II: PROGRAMMING EXERCISES

Before starting these programming exercises, you will need to make certain that you are working on a computer with particular software:

- python 2.7.x (<https://www.python.org/downloads/>)
- numpy (<http://www.numpy.org/>)
- scipy (<http://www.scipy.org/>)
- scikit-learn (<http://scikit-learn.org/stable/>)

This software has already been installed on the college's lab computers. If you choose to work on your own computer, you will need to install this software yourself. The instructions for installing scikit-learn already include the instructions for installing numpy and scipy, so we recommend that you start there. There is a thread on piazza discussing common installation issues, so if you run into any trouble, be sure to consult the thread or post your own problem (and solution, once you have it) there.

To test whether your computer is set up to run these exercises, launch the python interpreter from the command line using the command `python`. Make certain that it says that you're running python version 2.7.x; if not, you may need to change the python executable you're running.

Then, run the following code in the python interpreter (you should just be able to cut & paste the code):

```
from sklearn import tree
X = [[0, 0], [2, 2]]
y = [0.5, 2.5]
clf = tree.DecisionTreeRegressor()
clf = clf.fit(X, y)
clf.predict([[1, 1]])
```

If this code runs without error and gives you the following output:

```
array([ 0.5])
```

then everything should be configured correctly for this homework.

Although you will be able to complete the first programming exercise immediately, you will likely need to wait for subsequent lectures to be able to complete the remainder of the assignment.

### 1 Getting Started with Scikit-learn / Decision Trees (20 pts)

In the first programming exercise, we will get started with using scikit-learn and python by using an existing decision tree implementation provided with scikit-learn. We will be applying decision tree learning to a real-world data set: evaluating cardiac Single Proton Emission Computed Tomography (SPECT) images.

#### Relevant Files in Homework 1 Skeleton<sup>1</sup>

- **example\_dt\_iris.py**: script to load the IRIS dataset and perform decision tree learning on each pair of features, plotting the results
- **data/SPECTF.dat**: the complete SPECTF Heart data set
- **dtree\_eval.py**: script to load the SPECTF data, train the decision tree model, and test its performance. You will modify this file to implement 100 trials of ten-fold cross validation.

---

<sup>1</sup>**Bold text** indicates files that you will need to complete; you should not need to modify any of the other files.

## 1.1 Getting Started

Read through and run the `example_dt_iris.py` script included in the assignment files. This script loads the iris dataset and trains a decision tree classifier on it. The script also plots the decision surface (color-coded by class, of which there are three) for each pair of four possible features. Notice how the decision surfaces are all axis-aligned rectangular in shape. Make sure you understand how this script works, since you will need to build upon it to train the decision tree later.

Read through scikit-learn's pages on the decision tree classifier, available at: <http://scikit-learn.org/stable/modules/tree.html>. You may also find it helpful to look through the tutorials available on the scikit-learn website.

## 1.2 Data Set Description

We will be applying decision tree learning to the evaluation of cardiac Single Proton Emission Computed Tomography (SPECT) images. We will work with a database of 267 SPECT image sets, each of which corresponds to a patient. Each patient's scan was classified as either "normal" or "abnormal" by a physician; your job is to train a classifier to automatically evaluate SPECT image sets based on this training data. Instead of working with raw image sets, each SPECT image set was processed to extract 44 continuous features that summarize the original SPECT images. Each feature is a number between 0 and 100 corresponding to a "region of interest" in the image during stress or at-rest tests. The data is given in `SPECTF.dat`: the first column represents the class label and the remaining columns represent the features. The SPECTF data originally came from <http://archive.ics.uci.edu/ml/datasets/SPECTF+Heart>.

## 1.3 Comparing Decision Trees

To get you started, we have already provided code in the `dtree_eval.py` script to: read in the SPECTF data, randomize the order of the instances in the data set, split the data into training and testing sets, train the built-in SKLearn decision tree classifier, predict labels for the test data, and report the classifier's performance compared to the true labels (as determined by cardiologists). Run the code from the command line via `python dtree_eval.py`. Notice that the script outputs the accuracy for just one particular training/testing split; this is hardly a good measure of how a decision tree could perform on this problem!

Your task is to modify the script to output a good estimate of the classifier's performance, averaged over 100 trials of 10-fold cross-validation over the SPECTF data set. Be certain to follow the experimental procedure we discussed in class. As a reminder, make certain to observe the following details:

- For each trial, split the data randomly into 10 folds, choose one to be the "test" fold and train the decision tree classifier on the remaining nine folds. Then, evaluate the trained model on the held-out "test" fold to obtain its performance.
- Repeat this process until each fold has been the test fold exactly once, then advance to the next trial.
- Be certain to shuffle the data at the start of each trial, but never within a trial. Report the mean and standard deviation of the prediction accuracy over all 100 trials of 10-fold cross validation. In the end, you should be computing statistics for 1,000 accuracy values.

Note: although sklearn provides libraries that implement cross-fold validation, you may not use them for this assignment – you must implement cross-fold validation yourself.

Once it is working for the basic decision tree, modify this script to also evaluate the performance of decision stumps (a 1-level decision tree) and a 3-level decision tree on the same data. Make certain that the three classifiers are trained and tested on exactly the same subsets of the data each trial/fold.

Your implementation should be placed entirely in the `evaluatePerformance()` function, which should output a matrix of statistics as defined in the API specified in the function header in `dtree_eval.py`.

## 1.4 [CIS 519 ONLY] Generating a Learning Curve (10 pts)

Modify the code above to also generate and output a plot showing the learning curve over the training data. The learning curve should plot the mean and standard deviation of the test accuracy for 10%, 20%, ..., 100% of the training data. Note that 100% of the training data corresponds to only 90% of the complete data set, since we're doing 10-fold cross-validation.

As before, the learning curve statistics should be computed over 100 trials of 10-fold cross-validation. Make certain that you compute the learning curve for each subset of the training set for a particular trial/fold combination; in other words, the learning curve should be generated in an **inner loop** inside the trial/fold loops. In addition to the decision stumps, 3-level decision tree, and depth-unlimited decision tree, add a few additional decision trees of varying limited depths to explore the effect of decision tree depth on learning performance. Display the learning curves for all classifiers on the same plot. Add a well-labeled key to your plot.

To display the standard deviations on the plot, see the `fill_between` ([http://matplotlib.org/api/pyplot\\_api.html#matplotlib.pyplot.fill\\_between](http://matplotlib.org/api/pyplot_api.html#matplotlib.pyplot.fill_between)) or `errorbar` ([http://matplotlib.org/examples/statistics/errorbar\\_demo\\_features.html](http://matplotlib.org/examples/statistics/errorbar_demo_features.html)) functions in matplotlib.

Include your figure in your PDF writeup of the assignment, and make certain that it is well-labeled.

## 2 Linear Regression (20 pts)

In this exercise, you will implement multivariate linear regression. This exercise looks very long, but in practice you implement only around 10 lines of code total; it is designed to walk you through working with scikit-learn and implementing machine learning algorithms from scratch in python.

The homework 1 codebase contains a number of files and functions for this exercise:

### Files and API

`test_linreg_univariate.py`: script to test univariate linear regression

- `plotData1D`: produce the scatter plot of the one dimensional data
- `plotRegLine1D`: You will use the `plotData1D` function to make a scatter plot and use the parameters obtained from regression to plot a regressed line on the same plot.
- `visualizeObjective`: Visualize the objective function by plotting surface and contour plots (You do not edit this method)

`test_linreg_multivariate.py`: script to test multivariate linear regression

`linreg.py`: file containing code for linear regression

`LinearRegression` : class for multivariate linear regression

- `__init__`: constructor of the class
- `fit`: method to train the multivariate linear regression model
- `predict`: method to use the trained linear regression model for prediction
- `computeCost`: compute the value of the objective function
- `gradientDescent`: optimizes the parameter values via gradient descent

### Data Sets (located in the data directory)

- `univariateData.dat`: data for the univariate linear regression problem
- `multivariateData.dat`: data for the multivariate linear regression problem

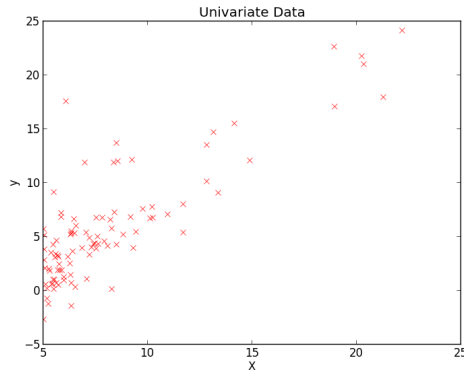


Figure 1: Scatter plot of the 1D data

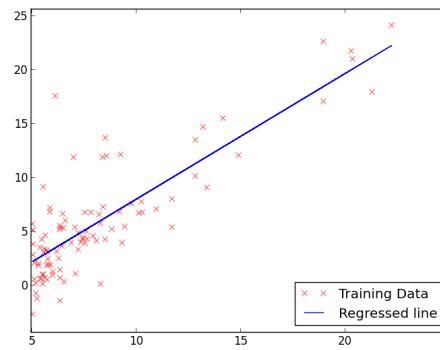


Figure 2: Regressed line of the 1D data

## 2.1 Visualizing the Data

Visualization of data often provides valuable insight into the problem, but is frequently overlooked as part of the machine learning process. We will start by visualizing the univariate data set using a 2D scatter plot of the output vs the input. However, in this class we will typically be dealing with multi-dimensional data sets. Once we go beyond two dimensions, visualization becomes much more difficult. In such cases, we must either visualize each dimension separately, or use dimensionality reduction techniques (such as PCA) to reduce the number of features. Later in the course, we will discuss such techniques.

You can load the univariate data into the matrix variables  $X$  and  $y$  by executing the following commands in the python interpreter from within the `hw1` directory:

```
import numpy as np
filePath = "data/univariateData.dat"
file = open(filePath, 'r')
allData = np.loadtxt(file, delimiter=',')
X = np.matrix(allData[:, :-1])
y = np.matrix((allData[:, -1])).T
# get the number of instances (n) and number of features (d)
n,d = X.shape
```

Then, create a scatterplot of the data using the `plotData1D` method:

```
from test_linreg_univariate import plotData1D
plotData1D(X,y)
```

Your output graph should be similar to Figure 1.

This is a good chance to learn about matplotlib and the plotting tools in python. Matplotlib is a python 2D plotting library (can be easily extended to 3D with other libraries) which creates MATLAB-like plots. For details, see the code for `plotData1D`.

## 2.2 Implementation

Implement multivariate linear regression via gradient descent by completing the `LinearRegression` class skeleton. Be certain not to change the class API. The only places you need to change in the file are marked with "TODO" comment tags.

Linear regression fits the parameter vector  $\theta$  to the dataset. In this exercise, we will use gradient descent to find the optimal solution. Recall that the  $L_2$  linear regression objective function is convex, so gradient descent will find the global optimum. This problem also has a closed-form solution, but more on that later.

Linear regression minimizes the squared loss on the data set to yield the optimal  $\hat{\theta}$ , which is given as

$$\hat{\theta} = \underset{\theta}{\operatorname{argmin}} J(\theta) \quad (1)$$

$$\mathcal{J}(\theta) = \frac{1}{2n} \sum_{i=1}^n \left( h_{\theta}(\mathbf{x}^{(i)}) - y^{(i)} \right)^2, \quad (2)$$

where the function  $h_{\theta}(\mathbf{x})$  maps the input feature space to the output space via

$$h_{\theta}(\mathbf{x}) = \theta^T \mathbf{x}. \quad (3)$$

In the case of univariate regression (where  $\mathbf{x}$  is only one variable),  $h_{\theta}(\mathbf{x}) = \theta^T \mathbf{x} = \theta_0 + \theta_1 x$ . Note that  $\theta_0$  acts as a bias term. Instead of treating this term separately, we can incorporate it into the same format as Equation 3 by adding a new feature containing a one (1) to every instance  $x$ ; this allows us to treat  $\theta_0$  as the coefficient over just another feature  $x_0$  whose value is always 1. For the entire data set  $\mathbf{X}$ , we can add a column of ones to the  $\mathbf{X}$  matrix in order to include this bias term via

```
X = np.c_[np.ones((n,1)), X]
```

Gradient descent searches the space of possible  $\theta$ 's to minimize the cost function  $\mathcal{J}(\theta)$ . The basic loop of gradient descent has already been implemented for you; you will just need to fill in the update equation. Each iteration of the descent should perform the following simultaneous update to the parameters:

$$\theta_j \leftarrow \theta_j - \alpha \frac{1}{n} \sum_{i=1}^n \left( h_{\theta}(\mathbf{x}^{(i)}) - y^{(i)} \right) x_j^{(i)}. \quad (4)$$

When using this update equation, **be certain to update all  $\theta_j$ 's simultaneously**. Each step of gradient descent will bring  $\theta$  closer to the optimal value that minimizes  $\mathcal{J}(\theta)$ . The variable  $\alpha$  is the learning rate, and should be chosen to be small (e.g.,  $\alpha = 0.01$ ).

The initial value for each entry of  $\theta$  is chosen randomly in some small range around 0 (using either a Gaussian with a small variance or drawing uniformly with a small range is fine, so long as the mean is 0).

Finally, in order to make our implementation easy to test, we will implement the cost function  $\mathcal{J}(\theta)$  (Equation 2) as a separate function `computeCost` in the `LinearRegression` class.

## Common Issues

- Make certain that all  $\theta_j$ 's are updated simultaneously; don't update one entry of the vector and then use that partly-updated  $\theta$  to compute  $h_{\theta}(\mathbf{x}^{(i)})$  while updating another entry in the same gradient descent iteration.
- Remember that gradient descent is searching over the space of  $\theta$ 's;  $\mathbf{X}$  and  $\mathbf{y}$  do not change in each iteration.

**Testing Your Implementation** One simple way to test your implementation is to examine the value of  $\mathcal{J}(\theta)$  printed out each iteration of gradient descent. If it is working correctly, you should see the cost monotonically decreasing over time.

Once you are finished with your implementation, train it on the univariate data set and then call the `plotRegLine1D` function in the `test_linreg_univariate.py`.

```
from test_linreg_univariate import plotRegLine1D
from linreg import LinearRegression
X = np.c_[np.ones((n,1)), X] # if you didn't do this step already
lr_model = LinearRegression(alpha = 0.01, n_iter = 1500)
lr_model.fit(X,y)
plotRegLine1D(lr_model, X, y)
```

You should get a plot that looks like Figure 2.

## 2.3 Understanding Gradient Descent

For this part, you do not need to write any code. To better understand our implementation, we will visualize the objective function and the path chosen by gradient descent.

For the univariate data set, we can plot the variation of the objective function over the space of  $\theta_0$  and  $\theta_1$  as a surface plot and a contour plot to show the convex shape and the descent. The blue line in Figure 3 traces the path taken by the gradient descent and the magenta dots show the points at each iteration.

To see this with your own implementation, run the following command from the command prompt (not within the python interpreter):

```
python test_linreg_univariate.py
```

Once you have the plot, move it around to clearly observe the path taken by gradient descent. Explore these results by changing the starting point for gradient descent (i.e., the initial value of  $\theta$ ). Refer to the `visualizeObjective` function in `test_linreg_univariate.py` for more details.

Once your implementation is working on univariate data, test it on multivariate data by running

```
python test_linreg_multivariate.py
```

from the command line.

## 2.4 Accelerating Our Implementation

Although it won't make much difference with the small data sets we're using in this problem, we can often make machine learning implementations much faster and more concise by vectorizing our code. Modify your previous implementation to compute the cost function and gradient updates using only matrix operations (e.g., no loops within the equation).

For example, the objective (cost) function can be written in matrix form as:

$$\mathcal{J}(\theta) = \frac{1}{2n} (\mathbf{X}\theta - \mathbf{y})^\top (\mathbf{X}\theta - \mathbf{y}) \quad , \quad (5)$$

where  $\mathbf{X} \in \mathbb{R}^{n \times d}$  is the matrix of instances, and  $\mathbf{y} \in \mathbb{R}^n$  is the vector of corresponding labels. It is up to you to figure out the matrix form of the gradient descent update step.

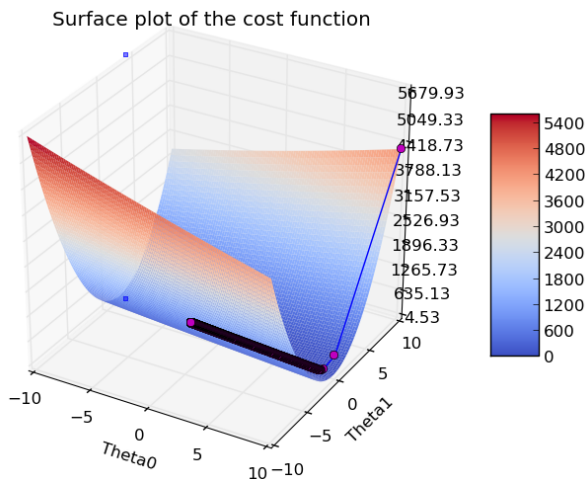


Figure 3: Surface plot of the cost function and gradient descent, starting with  $\theta = [10, 10]$

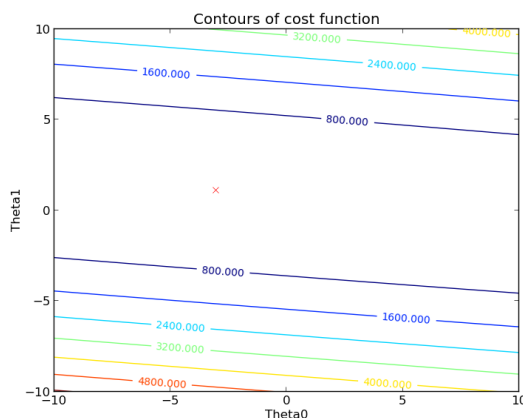


Figure 4: Contour plot of the gradient descent



## 2.5 Closed Form Solution

In this section, you will implement the closed-form solution for linear regression. Recall that the closed-form solution is given by

$$\boldsymbol{\theta} = (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top \mathbf{y} \quad (6)$$

Update the code in the `main` function of `test_linreg_univariate.py` and `test_linreg_multivariate.py` to print out the solution obtained using the closed form formula. For full credit, your computation should only require one line of code. Confirm that your implementation yields nearly the same result as the closed form solution in both the univariate and multivariate cases.

On first glance, using the closed form solution seems a lot easier than going through the trouble of using gradient descent. However, the closed form solution only works in batch learning scenarios with relatively small data sets. Specifically, here are two important cases when using gradient descent to solve linear regression is essential:

1. When the size of  $\mathbf{X}$  becomes large (either in terms of the number of instances or the number of features), the matrix inversion required to obtain the closed form solution becomes extremely computationally expensive. Gradient descent is *much* faster in this case.
2. In the online learning paradigm, training instances arrive in a stream, one after the other. Consequently, we don't have all of the data available (as required for the closed form solution), and so much use alternative techniques, such as stochastic gradient descent, which we will discuss later in the semester. (As a quick primer, stochastic gradient descent uses only one data instance to update the model parameters and is used as a fast approximation for batch gradient descent.)