Anthony Poerio
adp59@pitt.edu
University of Pittsburgh
CS1571 – Artificial Intelligence
Homework #04
Linear Regression – Report

Project Overview

For our fourth assignment in Dr. Hwa's AI class, our goal was to create a program capable of performing both univariate and multivariate linear regression, using a custom gradient descent algorithm of our own implementation.

The project is divided into two parts. 1) Univariate Regression; 2) Multivariate Regression. All code is located in the same file, **lin_regr.py**.

In this document, I will outline implementation decisions, difficulties, and final outcomes from each part of the project.

Part I – Univariate Regression

Part I of the project entails creating a gradient descent algorithm to fit a linear regression model against our set of input data → then using that regression model to predict outcomes for a set of test-data, and determining whether these predicted results are reasonable. The chief decisions, difficulties and outcomes are discussed below.

Learning Rate (alpha)

For my univariate regression a gradient descent algorithm, the most difficult aspect was determining a learning rate (or "alpha" value) that achieved the desired results quickly and efficiently.

In the end, I chose to set **alpha=0.01**, and I determined this value simply by using trial and error. At first, I was using 0.8 (which seemed like a reasonably place to start, just based on gut-feeling), but instead of converging, my gradient descent algorithm was ***diverging***, and giving me an overflow error, because my variables quickly became numbers higher than 32-bits could hold. This was obviously a very big problem.

In response to this, I made my learning rate (alpha) progressively smaller, until I found that 0.01 worked well. I also decided to slowly reduce alpha each iteration, by **subtracting 0.001 each time through the loop**, hoping that this would help us find global minima.

Convergence Criterion
My "convergence criterion" for this algorithm was this: stop the algorithm when ***none of our weighting variables have change by more than 0.00001*** since the last iteration.

This makes sense because if we are at the point where the weighting variables are changing by less than one-ten-thousandth each iteration, we can safely say that we've reached a convergence point (or global minimum). This is highly stable, and if we are no longer changing by more that 1/10,000 each time through the loop, I suggest that we are highly likely to have found the global minimum, or some value very close to it.

Outcome
My outcome, using the example data was as follows:

Parameters of trained model are: **w0=2.93833915257, w1=-3.18675772211**
---------TESTING--------
USING LINEAR REGRESSION
The AVERAGE Squared Error over the Entire Testing Data Set = **12.0965249944**

Part II – Multivariate Regression

Part II, was the more difficult section of this project. The goal of Part II is to take the gradient descent algorithm developed for Part II, and generalize it—so that it can be used for fitting an arbitrary number of weighting variables in a multiple regression model.

Implementation Notes
I am **randomly dividing** the input data into two sets → 80% training and 20% testing. I also use an assertion to ensure that we aren't too far from an 80/20 division. And given that I am using some randomization, **you may get an assertion error,** *if data sets aren't divided evenly* (and **expected to happen on occasion**, given the nature of pseudo-randomness in python). **If it does, please re-run.**

Learning Rate and Convergence Criterion
I began my experiment by using the same learning rate and convergence criterion as outlined in Part I. That is, specifically:

- **Alpha=0.01 for learning rate** (and reduce it by 0.001, every iteration)
- **Convergence criterion is achieved when** NO VARIABLES have changed by more than 1/10,000 of a decimal point since the previous iteration.

Practically, this gave me very good results immediately—so I chose not to fiddle with these values. I kept these the same in BOTH gradient descent algorithms.

Final Feature Set → Only Use Academic Features
In the end, the feature set I decided upon is this labeled "**Test_Academic_Features_Only**" in my code-base set. That is, I decided to strip all features which are not explicitly focused on inherent academic ability, or academic achievement. My thoughts are that these are the most relevant predictors for future academic success. And while it did NOT give ***large*** improvement over the regular feature vector, it does consistently perform the best out of all features I tested, though **by a small margin**.

Here is the source code for the vector transformation, as an explicit explanation of how the feature vector is transformed:

```
def academic_features_only(feature_vector_list):
    # Only factor in academic features
    academic_vectors = []

    for pair in feature_vector_list:

        vector = pair[0]
        y = pair[1]

        academic_vector = [0 for x in range(0,5)]
        academic_vector[0] = vector[4]  # weekly study time
        academic_vector[1] = vector[5]  # number of past class failures
        academic_vector[2] = vector[12] # number of absences
        academic_vector[3] = vector[1]  # mother's education
        academic_vector[4] = vector[2]  # father's education

        academic_pair = (academic_vector, y)
        academic_vectors.append(academic_pair)

    return academic_vectors
```

In my most recent example run, this feature vector yielded the following results:

RESULTS:

- Parameters of trained model are: w0=0.513794112569, w1=0.00173145394059, w2=0.0, w3=0.000921015746973, w4=0.00699293535483, w5=0.00684385881733
- The **AVERAGE Squared Error** over the Entire Testing Data Set = **12.632237686**
- The AVERAGE Overall Error For __any given prediction__ = 3.55418593858

Average squared error of 12.632 is really not bad at all. That means the predictions are only 3.55pts away from the true Y-value on average, using a 20-point scale.

Comparison Against Control
I also used two "control vectors" to compare this performance against random guessing. The results of each control vector are:

CONTROL SET: RANDOM GUESSING PARAMETERS IN RANGE [-1,1]
For this set, I randomly guessed 13 weighted parameters, setting value (Wn) to some floating-point number on the range [-1,1]. I chose this range because, practically, that is the range in which all of my Wn values were falling using the the gradient descent algorithm, after normalize each vector's value.

- The **AVERAGE Squared Error** over the Entire Testing Data Set = **2470.35109152**
- The AVERAGE Overall Error For __any given prediction__ = 49.7026266058

CONTROL SET: RANDOM GUESSING BETWEEN 0 and 20
For another control, I also simply randomly guessed Y-Value between 0 and 20, to see how our feature vector would compare against ***random guessing directly***, not even using a linear regression model.

- The **AVERAGE Squared Error** over the Entire Testing Data Set = **128.813382353**
- The AVERAGE Overall Error For __any given prediction__ = 11.3495983344

Clearly, **an average squared error of 12.632** hugely outperforms both control vectors which use random guessing of two (2) different types.

From this, **we can conclude** that using our Academic Feature Vector is and performing gradient descent on it to find weighting values, and fitting it against a linear regression is **very effective compared to control (random guessing)**.

Other Vectors Tried

Here is a list of other vectors that I tried, and a short rationale for each. To see how each feature vector was assembled in more detail, you can view the source code in **lin_regr.py**, starting at **line 452.**

If you run a multivariate test, you will see output for each of these vectors and the steps the model takes to achieve its final weights.

Original Feature Vector Unchanged → (Use all available data)
My first test was to just fit the a linear multivariate regression model to the feature vector that is passed, without changing it at all. Practically, this gave reasonable results (not all that dissimilar from the final vector I chose), but it still underperformed by final feature vector slightly. I used this as a "**baseline**" of sorts, to see if removing variables actually led to any improvement. Generally, it did → but

using ***all available data*** didn't really hurt very much. This leads me to believe it would be a reasonable solution.

Personal Features Only → (Only data related to personal/family life)
I also tried using data **ONLY** related to personal/family life for each student. (I.e. – Free time after school, how much they drink, quality of relationships, going out with friends). This, however, did not yield a large improvement over simply using all data points.

Parental Features Only → (Only data related to parents' education-level and student age)
The idea for this feature set is that the other data points are all silently affected by only three 'true' variables: 1) Mother's Education Level; 2) Father's Education Level; 3) Student's Current age. The thinking is that parents who are high academic achievers will control their children's lives (especially when they are younger) and the other variables are really just indicators of how the environment that the parents create at home.

This was the most interesting vector to me, and actually you should see in your own testing that the results of using only these three data points do not differ very much at all from using the WHOLE data set. So, in some respects, I think this hypothesis is correct. But it does not IMPROVE or outperform the whole data set significantly. Still thought, it interesting in its own right.

Health Features Only → (Only Data Relate to Student's Health)
Next, I tried to isolate vectors related to each student's Health. Again, the idea is that healthy students outperform unhealthy students, and that really the biggest indicator here is whether students are mentally/physically healthy enough to focus on their studies.

In some cases, this does outperform the whole-vector data set. But it wasn't consistent enough in my testing to say that it is definitively the best option.

Academics, Age, and Time → (Use all vectors related to academics, age, and travel time)
Finally, I tried to improve upon the academics vector by factoring in the student's age and travel time to school. These all seem like important variables, given that age often indicates maturity level. And the more time one spends traveling to school, the more tired s/he will be, and the less free time/energy they will have to study.

Again, in some cases this outperforms the whole-vector data set, but the overall results were inconsistent. And the regular academic vector seems just as good, and it has fewer variables; it is more simple. That is why, in the end, I decided to go with regular academics vector, as outlined above**. Overall, I am happy with these results, which can be confirmed by running the program.**