CPSC 340 Assignment 3 (due 2017-02-19 at 11:59pm)

Linear Regression

Instructions

Rubric: {mechanics:3}

The above points are allocated for following the general homework instructions.

A note on the provided code: we have switched the code style to object-oriented, which is hopefully less painful than things like

```
model["predict"] = predict_equality
```

that plagued us in earlier assignments. Now that you've all used Python for at least a few CPSC 340 assignments, hopefully the object-oriented style isn't too much syntax to absorb.

A note on Python 2: if you are using Python 2.7, please add

```
from __future__ import division
```

to the top of each Python file. You should also grab the Python 2 compatible data files from the "home" repo on GitHub, like you did for Assignment 2.

1 Vectors, Matrices, and Quadratic Functions

The first part of this question makes you review basic operations on vectors and matrices. If you are rusty on basic vector and matrix operations, see the notes on linear algebra on the course webpage. The second part of the question gives you practice taking the gradient of linear and quadratic functions, and the third part gives you practice finding the minimizer of quadratic functions.

1.1 Basic Operations

Rubric: {reasoning:3}

Using the definitions below,

$$\alpha = 5, \quad x = \begin{bmatrix} 2 \\ 3 \end{bmatrix}, \quad y = \begin{bmatrix} 1 \\ 4 \end{bmatrix}, \quad z = \begin{bmatrix} 2 \\ 0 \\ 1 \end{bmatrix}, \quad A = \begin{bmatrix} 1 & 2 \\ 2 & 3 \\ 3 & 2 \end{bmatrix},$$

evaluate the following expressions (show your work, but you may use answers from previous parts to simplify calculations):

1. $x^T x = \begin{bmatrix} 2 & 3 \end{bmatrix} \begin{bmatrix} 2 \\ 3 \end{bmatrix} = 2^2 + 3^2 = 4 + 9 = 13$
2. $\|x\|^2 = x^T x = 13$
3. $x^T(x + \alpha y) = \begin{bmatrix} 2 & 3 \end{bmatrix} (\begin{bmatrix} 2 \\ 3 \end{bmatrix} + 5 \begin{bmatrix} 1 \\ 4 \end{bmatrix}) = \begin{bmatrix} 2 & 3 \end{bmatrix} \begin{bmatrix} 7 \\ 23 \end{bmatrix} = 2(7) + 3(23) = 83$
4. $Ax = \begin{bmatrix} 1 & 2 \\ 2 & 3 \\ 3 & 2 \end{bmatrix} \begin{bmatrix} 2 \\ 3 \end{bmatrix} = \begin{bmatrix} 2+6 \\ 4+9 \\ 6+6 \end{bmatrix} = \begin{bmatrix} 8 \\ 13 \\ 12 \end{bmatrix}$
5. $z^T Ax = \begin{bmatrix} 2 & 0 & 1 \end{bmatrix} \begin{bmatrix} 8 \\ 13 \\ 12 \end{bmatrix} = 2(8) + 0(13) + 1(12) = 28$
6. $A^T A = \begin{bmatrix} 1 & 2 & 3 \\ 2 & 3 & 2 \end{bmatrix} \begin{bmatrix} 1 & 2 \\ 2 & 3 \\ 3 & 2 \end{bmatrix} = \begin{bmatrix} 1+4+9 & 2+6+6 \\ 2+6+6 & 4+9+4 \end{bmatrix} = \begin{bmatrix} 14 & 14 \\ 14 & 17 \end{bmatrix}$

If $\{\alpha, \beta\}$ are scalars, $\{x, y, z\}$ are real-valued column-vectors of length d, and $\{A, B, C\}$ are real-valued $d \times d$ matrices, state whether each of the below statements is true or false in general and give a short explanation.

7. $yy^T y = \|y\|^2 y$
   False, as $\|y\|^2 = y^T y \neq yy^T$
8. $x^T A^T(Ay + Az) = x^T A^T Ay + z^T A^T Ax$
   False, $x^T A^T(Ay + Az) = x^T A^T Ay + x^T A^T Az$
9. $x^T(B + C) = Bx + Cx$
   False, as $x^T A$ is not the same as Ax
10. $(A + BC)^T = A^T + C^T B^T$
    True, as $(A + BC)^T = A^T + (BC)^T = A^T + C^T B^T$
11. $(x - y)^T(x - y) = \|x\|^2 - x^T y + \|y\|^2$
    False, as $(x - y)^T(x - y) = \|x - y\|^2 = \|x\|^2 - 2x^T y + \|y\|^2$
12. $(x - y)^T(x + y) = \|x\|^2 - \|y\|^2$
    True, as $(x_i - y_i)(x_i + y_i) = x_i^2 - y_i^2$
    therefore, $(x - y)^T(x + y) = \|x\|^2 - \|y\|^2$

Hint: check the dimensions of the result, and remember that matrix multiplication is generally not commutative.

1.2 Converting to Matrix/Vector/Norm Notation

Rubric: {reasoning:3}

Using our standard supervised learning notation (X, y, w) express the following functions in terms of vectors, matrices, and norms (there should be no summations or maximums).

1. $\sum_{i=1}^n |w^T x_i - y_i|$.
2. $\max_{i \in \{1,2,\ldots,n\}} |w^T x_i - y_i| + \frac{\lambda}{2} \sum_{j=1}^n w_j^2$.
3. $\sum_{i=1}^n z_i (w^T x_i - y_i)^2 + \lambda \sum_{j=1}^d |w_j|$.

You can use Z to denote a diagonal matrix that has the values z_i along the diagonal.

1. $\sum_{i=1}^{n} |w^T x_i - y_i| = \|Xw - y\|_1$
2. $\max_{i\in\{1,2,\dots,n\}} |w^T x_i - y_i| + \frac{\lambda}{2}\sum_{j=1}^{n} w_j^2 = \|Xw - y\|_\infty + \frac{\lambda}{2}\|w\|_2^2$
3. $\sum_{i=1}^{n} z_i(w^T x_i - y_i)^2 + \lambda\sum_{j=1}^{d} |w_j| = \|\sqrt{Z}(Xw - y)\|_2^2 + \lambda\|w\|_1$

1.3 Minimizing Quadratic Functions as Linear Systems

Rubric: {reasoning:4}

Write finding a minimizer w of the functions below as a system of linear equations (using vector/matrix notation and simplifying as much as possible). Note that all the functions below are convex so finding a w with $\nabla f(w) = 0$ is sufficient to minimize the functions (but show your work in getting to this point).

1. $f(w) = \frac{1}{2}\|w - v\|^2$.
2. $f(w) = \frac{1}{2}\|w\|^2 + w^T X^T y$.
3. $f(w) = \frac{1}{2}\|Xw - y\|^2 + \frac{1}{2}w^T \Lambda w$.
4. $f(w) = \frac{1}{2}\sum_{i=1}^{n} z_i(w^T x_i - y_i)^2$.

1. $f(w) = \frac{1}{2}\|w - v\|^2 = \frac{1}{2}(w - v)^T(w - v) = \frac{1}{2}(w^T - v^T)(w - v) = \frac{1}{2}(w^T w - 2w^T v + v^T v)$

   $\nabla f(w) => \frac{1}{2}(2w - 2v + 0) = 0$
   $\nabla f(w) => w - v = 0 => w = v$

   Therefore $w = v$ is the minimizer for $f(w)$.

2. $f(w) = \frac{1}{2}\|w\|^2 + w^T X^T y = \frac{1}{2}w^T w + w^T X^T y$

   $\nabla f(w) => w + X^T y = 0 => w = -X^T y$

   Therefore $w = -X^T y$ is a minimizer for f(w).

3. $f(w) = \frac{1}{2}\|Xw - y\|^2 + \frac{1}{2}w^T \Lambda w$
   But we know, $\frac{1}{2}\|Xw - y\|^2 = \frac{1}{2}w^T X^T X w - w^T X^T y + \frac{1}{2}y^T y$

   Therefore, $f(w) = \frac{1}{2}w^T X^T X w - w^T X^T y + \frac{1}{2}y^T y + \frac{1}{2}w^T \Lambda w$

   $\nabla f(w) => X^T X w - X^T y + 0 + \Lambda w = 0$
   $\nabla f(w) => (X^T X + \Lambda)w = X^T y => w = (X^T X + \Lambda)^{-1} X^T y$

   Therefore if $X^T X + \Lambda$ is invertible, $w = (X^T X + \Lambda)^{-1} X^T y$ is a minimizer for $f(w)$.

4. $f(w) = \frac{1}{2}\sum_{i=1}^{n} z_i(w^T x_i - y_i)^2 = \frac{1}{2}\sum_{i=1}^{n}(\sqrt{z_i}(w^T x_i - y_i))^2 = \frac{1}{2}\|\sqrt{Z}(Xw - y)\|^2 = \frac{1}{2}\|\sqrt{Z}Xw - \sqrt{Z}y\|^2$

   $f(w) = \frac{1}{2}(\sqrt{Z}Xw - \sqrt{Z}y)^T(\sqrt{Z}Xw - \sqrt{Z}y) = \frac{1}{2}(w^T X^T\sqrt{Z^T} - y^T\sqrt{Z^T})(\sqrt{Z}Xw - \sqrt{Z}y)$

   $f(w) = \frac{1}{2}(w^T X^T\sqrt{Z^T Z}Xw - 2w^T X^T\sqrt{Z^T Z}y + y^T\sqrt{Z^T Z}y)$

   $\nabla f(w) => X^T\sqrt{Z^T Z}Xw - X^T\sqrt{Z^T Z}y + 0 = 0$
   $\nabla f(w) => X^T\sqrt{Z^T Z}Xw = X^T\sqrt{Z^T Z}y => w = (X^T\sqrt{Z^T Z}X)^{-1}X^T\sqrt{Z^T Z}y$

   Therefore if $X^T\sqrt{Z^T Z}X$ is invertible, $w = (X^T\sqrt{Z^T Z}X)^{-1}X^T\sqrt{Z^T Z}y$ is a minimizer for $f(w)$.

Above we assume that v is a $d \times 1$ vector, and Λ is a $d \times d$ diagonal matrix with positive entries along the diagonal.

Hint: Once you convert to vector/matrix notation, you can use the results from class to quickly compute these quantities term-wise. As a sanity check for your derivation, make sure that your results have the right dimensions.

2 Linear Regression and Nonlinear Bases

In class we discuss fitting a linear regression model by minimizing the squared error. This classic model is the simplest version of many of the more complicated models we will discuss in the course. However, it typically performs very poorly in practice. One of the reasons it performs poorly is that it assumes that the target y_i is a linear function of the features x_i with an intercept of zero. This drawback can be addressed by adding a bias variable and using nonlinear bases (although nonlinear bases may increase to over-fitting).

In this question, you will start with a data set where least squares performs poorly. You will then explore how adding a bias variable and using nonlinear (polynomial) bases can drastically improve the performance. You will also explore how the complexity of a basis affects both the training error and the test error. In the final part of the question, it will be up to you to design a basis with better performance than polynomial bases.

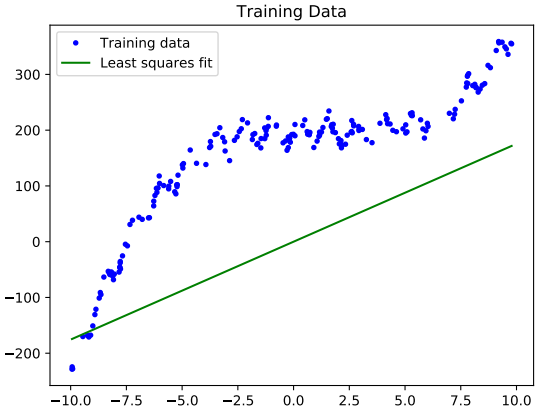
2.1 Adding a Bias Variable

Rubric: {code:3,reasoning:1}

If you run `python main.py -q 2.1`, it will:

1. Load a one-dimensional regression dataset.
2. Fit a least-squares linear regression model.
3. Report the training error.
4. Report the test error (on a dataset not used for training).
5. Draw a figure showing the training data and what the linear model looks like.

Unfortunately, this is an awful model of the data. The average squared training error on the data set is over 28000 (as is the test error), and the figure produced by the demo confirms that the predictions are usually nowhere near the training data:



The y-intercept of this data is clearly not zero (it looks like it's closer to 200), so we should expect to improve performance by adding a *bias* variable, so that our model is

$$y_i = w^T x_i + w_0.$$

instead of

$$y_i = w^T x_i.$$

In file *linear_model.py*, complete the class, *LeastSquaresBias*, that has the same input/model/predict format as the *LeastSquares* class, but that adds a *bias* variable w_0. Hand in your new class, the updated plot, and the updated training/test error.

Hint: recall that adding a bias w_0 is equivalent to adding a column of ones to the matrix X. Don't forget that you need to do the same transformation in the *predict* function.

- linear_model.py
- Updated plot:

- Training error = 3551.346, Testing error = 3393.869

2.2 Polynomial Basis

Rubric: {code:4,reasoning:1}

Adding a bias variable improves the prediction substantially, but the model is still problematic because the target seems to be a *non-linear* function of the input. Complete *LeastSquareBasis* class, that takes a data vector x (i.e., assuming we only have one feature) and the polynomial order p. The function should perform a least squares fit based on a matrix Z where each of its rows contains the values $(x_i)^j$ for $j = 0$ up to p. E.g., *LeastSquaresBasis.fit(x,y)* with $p = 3$ should form the matrix

$$Z = \begin{bmatrix} 1 & x_1 & (x_1)^2 & (x_1)^3 \\ 1 & x_2 & (x_2)^2 & (x_2)^3 \\ \vdots & & & \\ 1 & x_n & (x_n)^2 & (x_N)^3 \end{bmatrix},$$

and fit a least squares model based on it. Hand in the new class, and report the training and test error for $p = 0$ through $p = 10$. Explain the effect of p on the training error and on the test error.

Note: you should write the code yourself; don't use a library like sklearn's `PolynomialFeatures`

- linear_model.py
- $p = 0$, Training error = 15480.52, Testing error = 14390.763.
- $p = 1$, Training error = 3551.346, Testing error = 3393.869.
- $p = 2$, Training error = 2167.992, Testing error = 2480.725.
- $p = 3$, Training error = 252.046, Testing error = 242.805.
- $p = 4$, Training error = 251.462, Testing error = 242.126.
- $p = 5$, Training error = 251.144, Testing error = 239.545.
- $p = 6$, Training error = 248.583, Testing error = 246.005.
- $p = 7$, Training error = 247.011, Testing error = 242.888.
- $p = 8$, Training error = 241.306, Testing error = 245.966.
- $p = 9$, Training error = 235.762, Testing error = 259.296.
- $p = 10$, Training error = 235.074, Testing error = 256.3.
- As p increases, we are able to fit the data more accurately and reduce training and testing error, but at around $p = 5$, the testing error starts increasing due to overfitting on the given validation data.

3 Non-Parametric Bases and Cross-Validation

Unfortunately, in practice we often don't know what basis to use. However, if we have enough data then we can make up for this by using a basis that is flexible enough to model any reasonable function. These may perform poorly if we don't have much data, but can perform almost as well as the optimal basis as the size of the dataset grows. In this question you will explore using Gaussian radial basis functions (RBFs), which have this property. These RBFs depend on a parameter σ, which (like p in the polynomial basis) can

be chosen using a validation set. In this question, you will also see how cross-validation allows you to tune parameters of the model on a larger dataset than a strict training/validation split would allow.

3.1 Proper Training and Validation Sets

Rubric: {reasoning:3}

If you run `python main.py -q 3.1`, it will load a dataset and split the training examples into a "train" and a "validation" set. It will then search for the best value of σ for the RBF basis (it also uses regularization since Z^TZ tends to be very close to singular). Once it has the "best" value of σ, it re-trains on the entire dataset and reports the training error on the full training set as well as the error on the test set.

Unfortunately, there is a problem with the way this is done. Because of this problem, the RBF basis doesn't perform much better than a linear model. What is the problem with this training/validation/testing procedure? How can you fix this problem?

The problem is that the data in X is sorted, and when we split directly without random sampling the training set is all negative values and the validation set is almost all positive values and hence the error comes out so high. This can be fixed by randomly sampling indices from X for training and validation sets.

Hint: The problem is not with the `LeastSquaresRBF` code, it is with the training/validation/testing procedure. You may want to plot the models that are considered during the search for σ

3.2 Cross-Validation

Rubric: {code:3,reasoning:1}

Using the standard solution to the above problem, a strange behaviour appears: if you run the script more than once it might choose different values of σ. It rarely performs too badly, but it's clear that the randomization has an effect on the value of σ that we choose. This variability would be reduced if we had a larger "train" and "validation" set, and one way to simulate this is with *cross-validation*. Modify the training/validation procedure to use 10-fold cross-validation to select σ, and hand in your code. What value of σ does this procedure typically select?

Note: your should implement this yourself. Don't use a library like sklearn's `cross_val_score`.

- linear_model.py
- This method gives the value $\sigma = 1$, with Training error= 39.492, Testing error= 71.168 on full dataset.

3.3 Cost of Non-Parametric Bases

Rubric: {reasoning:3}

When dealing with larger datasets, an important issue is the dependence of the computational cost on the number of training examples n and the number of features d. What is the cost in big-O notation of training the model on n training examples with d features under (a) the linear basis and (b) Gaussian RBFs (for a fixed σ)? What is the cost of classifying t new examples under each of these two bases? When are RBFs cheaper to train? When are RBFs cheaper to test?

Big-O of:

- Training on Linear Basis = $O(nd^2)$
- Predicting on Linear Basis = $O(td)$

- Training on Gaussian RBF $= O(n^2 d)$
- Predicting on Gaussian RBF $= O(ndt)$

Gaussian RBFs are cheaper to train when the data is uniformly distributed, and they are cheaper to test when the testing data is distributed similar to the training data.

3.4 Non-Parametric Bases with Uneven Data

Rubric: {reasoning:1}

There are reasons why this dataset is particularly well-suited to Gaussian RBFs:

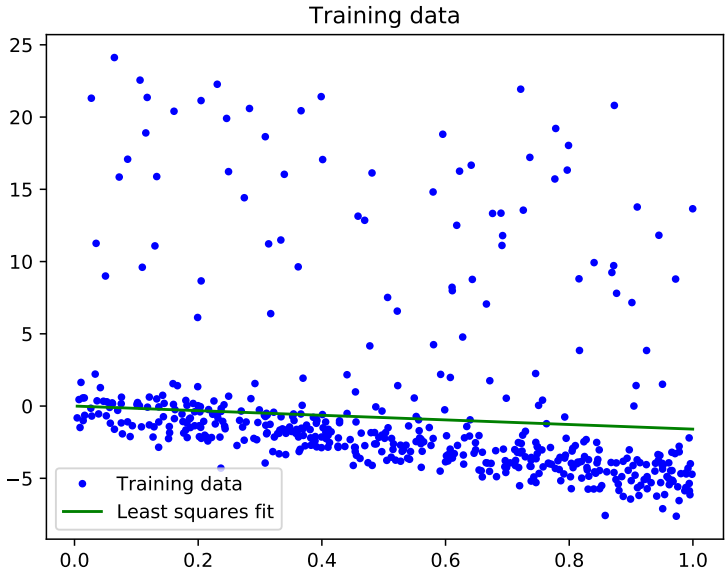
1. The period of the oscillations stays constant.
2. We have evenly sampled the training data across its domain.

If either of these assumptions are violated, the performance with our Gaussian RBFs might be much worse. Consider a scenario where 1 and/or 2 above is violated. What method(s) discussed in lecture might be more appropriate in this scenario?

1. If the period of oscillation is not constant, the best value of σ will vary for the data and hence the result will be less accurate than using polynomial regression.
2. If the training data is unevenly sampled, we will get vast differences in RBF basis values next to each other, which is not ideal for prediction as the testing data will most likely not have similar data and the RBF basis for prediction will be way off.

4 Robust Regression and Gradient Descent

If you run `python main.py -q 4.1`, it will load a one-dimensional regression dataset that has a non-trivial number of 'outlier' data points. These points do not fit the general trend of the rest of the data, and pull the least squares model away from the main downward trend that most data points exhibit:



4.1 Weighted Least Squares in One Dimension

Rubric: {code:3}

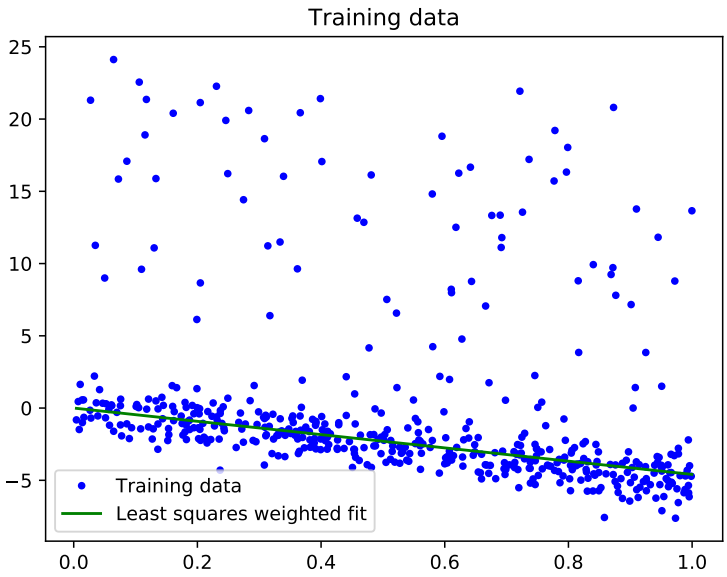
One of the most common variations on least squares is *weighted* least squares. In this formulation, we have a weight z_i for every training example. To fit the model, we minimize the weighted squared error,

$$f(w) = \frac{1}{2}\sum_{i=1}^{n} z_i(w^T x_i - y_i)^2.$$

In this formulation, the model focuses on making the error small for examples i where z_i is high. Similarly, if z_i is low then the model allows a larger error.

Complete the model class, *WeightedLeastSquares*, that implements this model (note that Q1.3.4 asks you to show how this formulation can be solved as a linear system). Apply this model to the data containing outliers, setting $z = 1$ for the first 400 data points and $z = 0.1$ for the last 100 data points (which are the outliers). Hand in your code and the updated plot.

- linear_model.py
- Updated plot:



4.2 Smooth Approximation to the L1-Norm

Rubric: {reasoning:3}

Unfortunately, we typically do not know the identities of the outliers. In situations where we suspect that there are outliers, but we do not know which examples are outliers, it makes sense to use a loss function that is more robust to outliers. In class, we discussed using the sum of absolute values objective,

$$f(w) = \sum_{i=1}^{n} |w^T x_i - y_i|.$$

This is less sensitive to outliers than least squares, but it is non-differentiable and harder to optimize. Nevertheless, there are various smooth approximations to the absolute value function that are easy to optimize. One possible approximation is to use the log-sum-exp approximation

$$|r| \approx \log(\exp(r) + \exp(-r)).$$

Using this approximation, we obtain an objective of the form

$$f(w) = \sum_{i=1}^{n} \log\left(\exp(w^T x_i - y_i) + \exp(y_i - w^T x_i)\right).$$

which is smooth but less sensitive to outliers than the squared error. Derive the gradient ∇f of this function with respect to w. You should show your work but you do not have to express the final result in matrix notation.

$f(w) = \sum_{i=1}^{n} \log\left(\exp(w^T x_i - y_i) + \exp(y_i - w^T x_i)\right)$

$\nabla f(w) = \sum_{i=1}^{n} \frac{\frac{d(\exp(w^T x_i - y_i) + \exp(y_i - w^T x_i))}{dw}}{\exp(w^T x_i - y_i) + \exp(y_i - w^T x_i)} = \frac{X^T \exp(Xw - y) - X^T \exp(y - Xw)}{\exp(Xw - y) + \exp(y - Xw)} = X^T \frac{\exp(Xw - y) - \exp(y - Xw)}{\exp(Xw - y) + \exp(y - Xw)}$

4.3 Robust Regression

Rubric: {code:2,reasoning:1}

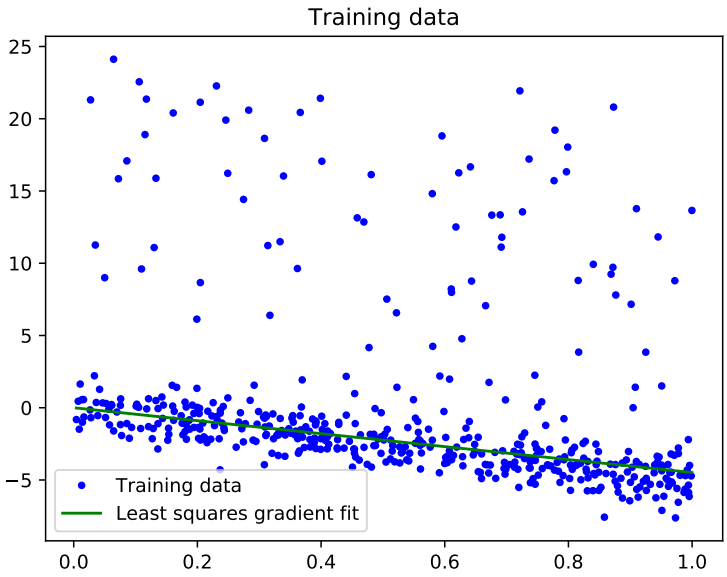
The class *LinearModelGradient* is the same as *LeastSquares*, except that it fits the least squares model using a *gradient descent* method. If you run `python main.py -q 4.3` you'll see it produces the same fit as we obtained using the normal equations.

One advantage of the gradient descent strategy is that it only costs $O(nd)$ for an iteration of the gradient method, which is faster than forming X^TX which costs $O(nd^2)$. Of course, we need to know the *number* of gradient iterations in order to precisely compare these two strategies, but for now we will assume that the number of gradient iterations is typically often reasonable.

The typical input to a gradient method is a function that, given w, returns $f(w)$ and $\nabla f(w)$. See *funObj* in *LinearModelGradient* for an example. Note that the *fit* function of *LinearModelGradient* also has a numerical check that the gradient code is approximately correct, since implementing gradients is often error-prone.[1]

A second advantage of gradient-based strategies is that they are able to solve problems that do not have closed-form solutions, such as the formulation from the previous section. The class *LinearModelGradient* has most of the implementation of a gradient-based strategy for fitting the robust regression model under the log-sum-exp approximation. The only part missing is the function and gradient calculation inside the *funObj* code. Modify *funObj* to implement the objective function and gradient based on the smooth approximation to the absolute value function (from the previous section). Hand in your code, as well as the plot obtained using this robust regression appraoch.

- linear_model.py
- Updated plot:



[1]Sometimes the gradient checker itself can be wrong. See CPSC 303 for a lot more on numerical differentiation.