

## SAMPLE FROM A3

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

### 1 Finding Similar Items

For this question we'll use the Amazon product data set<sup>1</sup> from <http://jmcauley.ucsd.edu/data/amazon/>. We will focus on the "Patio, Lawn, and Garden" section. You should start by downloading the ratings at

<https://stanford.io/2Q7QTvu> and place the file in your data directory with the original filename. Once you do that, running `python main.py -q 1` should do the following steps:

- Load the raw ratings data set into a Pandas dataframe.

- Construct the user-product matrix as a sparse matrix (to be precise, a `scipy.sparse` matrix).
- Create bi-directional mappings from the user ID (e.g. "A2VNYW6HJ1A3R1") to the integer index of the row in `X`.

- Create bi-directional mappings from the item ID (e.g. "0981850006") to the integer index of the column in `X`.

#### 1.1 Exploratory data analysis

##### 1.1.1 Most popular item

Find the item with the most total stars. [Submit the product name and the number of stars.](#)

Note: once you find the ID of the item, you can look up the name by going to the url

[https://www.amazon.com/dp/ITEM\\_ID](https://www.amazon.com/dp/ITEM_ID), where ITEM\_ID is the ID of the item. For example, the URL for item ID "B00CFM0P7Y" is <https://www.amazon.com/dp/B00CFM0P7Y>.

##### 1.1.2 User with most reviews

Find the user who has rated the most items, and the number of items they rated.

##### 1.1.3 Histograms

Make the following histograms:

1. The number of ratings per user
2. The number of ratings per item
3. The ratings themselves

Note: for the first two, use `plt.yscale('log', nonposy='clip')`

to put the histograms on a log-scale. Also, you can use `X.getnnz` to get the total number of nonzero elements along a specific axis.

### 1.2 Finding similar items with nearest neighbours

We'll use scikit-learn's `neighbors.NearestNeighbors` object to find the items most similar to the example item above, namely the

## 3 Robust Regression and Gradient Descent

If you run `python main.py -q 3`, 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:

Note: we are fitting the regression without an intercept here, just for simplicity of the homework question. In reality one would rarely do this. But here it's OK because the "true" line passes through the origin (by design). In Q?? we'll address this explicitly.

### 3.1 Weighted Least Squares in One Dimension

One of the most common variations on least squares is *weighted least squares*. ~~One possible formulation~~ we have a weight  $v_i$  for every training example  $i$ . The fitting procedure minimizes the weighted squared error,

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

In this formulation, the model focuses on making the error small for examples  $i$  where  $v_i$  is high. Similarly, if  $v_i$  is low then the model allows a larger error. Note: these weights  $v_i$  (one per training example) are completely different from the model parameters  $w_j$  (one per feature), which, confusingly, we sometimes also call "weights".

Complete the model class, `WeightedLeastSquares`, that implements this model (note that Q2.2.3 asks you to show how a few similar formulation can be solved as a linear system). Apply this model to the data containing outliers, setting  $v = 1$  for the first

400 data points and  $v = 0.1$  for the last 100 data points (which are the outliers). [Hand in your code and the updated plot.](#)

### 3.2 Smooth Approximation to the L1-Norm

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

Brass Grill Brush 18 Inch Heavy Duty and Extra Strong, Solid Oak Handle, at URL <https://www.amazon.com/dp/B00CFM0P7Y>.

Find the 5 most similar items to the Grill Brush using the following metrics:

1. Euclidean distance (the `NearestNeighbors` default)
2. Normalized Euclidean distance (you'll need to do the normalization)

3. Cosine similarity (by setting `metric='cosine'`)

Some notes/hints...

- If you run `python main.py -q 1.2`, it will grab the row of `X` associated with the grill brush. The `neighbors` method of `NearestNeighbors` is for taking neighbors of `w` and `Neof(w)` is the number of neighbors of `w`. See `funObj` in `LinearModelGradient` for the complete code. Note that the fifth line of `LinearModelGradient` also has a numerical check that the gradient code is approximately correct. Implementing the gradient is often a sparse matrix, it's a bit more of a mess. Therefore, use `sklearn.preprocessing.normalize` to help you with the gradient-based strategies part 2.

Did normalized Euclidean distance and cosine similarity yields the same similar items, as expected?

### 1.3 Total popularity

For both Euclidean distance and cosine similarity, find the number of reviews for each of the 5 recommended items and report it. Do the results make sense given what we discussed in class about Euclidean distance vs. cosine similarity and popular items?

Note: in `main.py` you are welcome to combine this code with your code from the previous part, so that you don't have to copy/paste

all that code in another section of `main.py`.

## 2 Matrix Notation and Minimizing Quadratics

### 2.1 Converting to Matrix/Vector/Norm Notation

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

Note that in part 2 we give a *weight*  $v_i$  to each training example, where  $v_i$  is a scalar. In the least squares linear regression model, we have a  $n \times d$  matrix  $X$  that has the values of the features for each training example, and  $y$  as a vector that has the target values for each training example. We also have a vector  $w$  that has the weights for each feature. The function `train` takes the training data and what the linear model looks like.

log-sum-exp approximation of the max function<sup>2</sup>:

$|r| = \max\{r, -r\} \approx \log(\exp(r) + \exp(-r))$ .

Using this approximation, we obtain an objective of the form

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

which is smooth but less sensitive to outliers than the squared error. Derive the gradient  $\nabla f$  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.

### 3.3 Robust Regression

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 3.3` you'll see it produces the same results as the `LeastSquares` method. The `funObj` method is a function that, given  $w$ , returns the value of the objective function. The `funObj` method in `LinearModelGradient` also has a numerical check that the gradient code is approximately correct. Implementing the gradient is often a sparse matrix, it's a bit more of a mess. Therefore, use `sklearn.preprocessing.normalize` to help you with the gradient-based strategies part 2.

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 approach.

## 4 Linear Regression and Nonlinear Bases

In class we discussed fitting a linear regression model by minimizing the squared error. 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.

### 4.1 Adding a Bias Variable

If you run `python main.py -q 4`, it will:

## 2.2 Minimizing Quadratic Functions as Linear Systems

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.  $f(w) = \frac{1}{2}\|w - v\|^2$  (projection of  $v$  onto real space).
2.  $f(w) = \frac{1}{2}\|Xw - y\|^2 + \frac{1}{2}w^T\Lambda w$  (least squares with weighted regularization).
3.  $f(w) = \frac{1}{2}\sum_{i=1}^n v_i(w^T x_i - y_i)^2 + \frac{\lambda}{2}\|w - w^0\|^2$  (weighted least squares shrunk towards non-zero  $w^0$ ).

Above we assume that  $v$  and  $w^0$  are  $d$  by 1 vectors, and  $\Lambda$  is a  $d$  by  $d$  diagonal matrix (with positive entries along the diagonal). You can use  $V$  as a diagonal matrix containing the  $v_i$  values 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.

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* (a.k.a. intercept) variable, so that our model is

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

instead of

$$y_i = w^0 x_i.$$

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 (also called an intercept)  $w_0$  (also called  $\beta$  in lecture). 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.

### 4.2 Polynomial Basis

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 `LeastSquarePoly` 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., `LeastSquaresPoly.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`.

1 The author of the data set has asked for the following citations: (1) Ups and downs: Modeling the visual evolution of fashion trends with one-class collaborative filtering. R. He, J. McAuley. WWW, 2016, and (2) Image-based recommendations on styles and substitutes. J. McAuley, C. Targett, J. Shi, A. van den Hengel. SIGIR, 2015.

2 Other possibilities are the Huber loss, or  $|r| \approx \sqrt{r^2 + \epsilon}$  for some small  $\epsilon$ .

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