EECS 189 Introduction to Machine Learning
Fall 2020

HW9

This homework is due **Wednesday, November 4 at 11:59 p.m.**

1 Getting Started

Read through this page carefully. You may typeset your homework in latex or submit neatly handwritten/scanned solutions. Please start each question on a new page. Deliverables:

1. Submit a PDF of your writeup, **with an appendix for your code**, to the appropriate assignment on Gradescope. If there are graphs, include those graphs in the correct sections. Do not simply reference your appendix.
2. If there is code, submit all code needed to reproduce your results.
3. If there is a test set, submit your test set evaluation results.

After you've submitted your homework, watch out for the self-grade form.

(a) Who else did you work with on this homework? In case of course events, just describe the group. How did you work on this homework? Any comments about the homework?

(b) Please copy the following statement and sign next to it. We just want to make it *extra* clear so that no one inadvertently cheats.

I certify that all solutions are entirely in my words and that I have not looked at another student's solutions nor have I looked at any online solutions to any of these problems. I have credited all external sources in this write up.

2 Classification Policy

Suppose we have a classification problem with classes labeled $1, \dots, c$ and an additional "doubt" category labeled $c+1$. Let $f : \mathbb{R}^d \to \{1, \dots, c+1\}$ be a decision rule. Define the loss function

$$L(f(\mathbf{x}), y) = \begin{cases} 0 & \text{if } f(\mathbf{x}) = y \quad f(\mathbf{x}) \in \{1, \dots, c\}, \\ \lambda_c & \text{if } f(\mathbf{x}) \neq y \quad f(\mathbf{x}) \in \{1, \dots, c\}, \\ \lambda_d & \text{if } f(\mathbf{x}) = c+1 \end{cases} \tag{1}$$

where $\lambda_c \geq 0$ is the loss incurred for making a misclassification and $\lambda_d \geq 0$ is the loss incurred for choosing doubt. In words this means the following:

- When you are correct, you should incur no loss.
- When you are incorrect, you should incur some penalty λ_c for making the wrong choice.
- When you are unsure about what to choose, you might want to select a category corresponding to "doubt" and you should incur a penalty λ_d.

We can see that in practice we'd like to have this sort of loss function if we don't want to make a decision if we are unsure about it. This sort of loss function, however, doesn't help you in instances where you have high certainty about a decision, but that decision is wrong.

To understand the expected amount of loss we will incur with decision rule $f(\mathbf{x})$, we look at what is called the risk. (We've also used the word "loss" for this kind of thing.) The risk of classifying a new data point $\mathbf{x}$ as class $f(\mathbf{x}) \in \{1, 2, \ldots, c+1\}$ is

$$R(f(\mathbf{x})|\mathbf{x}) = \sum_{i=1}^{c} L(f(\mathbf{x}), i)\, P(Y = i|\mathbf{x}).$$

(a) **Show that the following policy $f_{opt}(x)$ obtains the minimum risk:**

- (**R1**) Find class i such that $P(Y = i|\mathbf{x}) \geq P(Y = j|\mathbf{x})$ for all j, meaning you pick the class with the highest probability given x.
- (**R2**) Choose class i if $P(Y = i|\mathbf{x}) \geq 1 - \frac{\lambda_d}{\lambda_c}$
- (**R3**) Choose doubt otherwise.

(b) **How would you modify your optimum decision rule if $\lambda_d = 0$? What happens if $\lambda_d > \lambda_c$? Explain why this is or is not consistent with what one would expect intuitively.**

3 Sensors, Objects, and Localization (Part 2)

Let us say there are n objects and m sensors located in a $2d$ plane. The n objects are located at the points $(x_1, y_1), \ldots, (x_n, y_n)$. The m sensors are located at the points $(a_1, b_1), \ldots, (a_m, b_m)$. We have measurements for the distances between the objects and the sensors: D_{ij} is the measured distance from object i to sensor j. The distance measurement has noise in it. Specifically, we model

$$D_{ij} = \|(x_i, y_i) - (a_j, b_j)\| + Z_{ij},$$

where $Z_{ij} \sim N(0, 1)$. The noise is independent across different measurements.

Assume we observe $D_{ij} = d_{ij}$ with $(X_i, Y_i) = (x_i, y_i)$ for $i = 1, \ldots, n$ and $j = 1, \ldots, m$. Here, $m = 7$. **Our goal is to predict $(X_{i'}, Y_{i'})$ from newly observed $D_{i'1}, \ldots, D_{i'7}$.** For a data set with q points, the error is measured by the average distance between the predicted object locations and the true object locations,

$$\frac{1}{q} \sum_{i=1}^{q} \sqrt{(\hat{x}_i - x_i)^2 + (\hat{y}_i - y_i)^2},$$

where $(\hat{x}_i, \hat{y}_i)$ is the location of objects predicted by a model.

We are going to consider seven models in this problem and compare their performance:

- A *Zeros Model*: This model always predicts $(0, 0)$ as the location. It serves as a baseline for how well we might do if we totally ignore the data.
- A *Generative Model*: This is basically from the earlier assignment (HW7 prob. 10): we first estimate sensor locations from the training data by solving the nonlinear least squares problem using the Gauss-Newton algorithm[1], and then use the estimated sensor locations to estimate the new object locations.
- An *Oracle Model*: This is the same as generative model except that we will use the ground truth sensor location rather than the estimated sensor location.
- A *Linear Model*. Using the training set, the linear model attempts to fit (X_i, Y_i) directly from the distance measurements $(D_{i1}, \ldots, D_{i7})$. Then it predicts $(X_{i'}, Y_{i'})$ from $(D_{i'1}, \ldots, D_{i'7})$ using the map that it found during training. (It never tries to explicitly model the underlying sensor locations.)
- A *Second-Order Polynomial Regression Model*. The set-up is similar to the linear model, but including second-order polynomial features.
- A *Third-Order Polynomial Regression Model*. The set-up is similar to the linear model, but including third-order polynomial features.
- A *Neural Network Model*. The Neural Network should have two hidden layers, each with 100 neurons, and use ReLU and/or tanh for the non-linearity. (You are encouraged to explore on your own beyond this however. These parameters were chosen to teach you a hype-deflating lesson.) The neural net approach also follows the principle of finding/learning a direct connection between the distance measurements and the object location.

(a) **Implement the last four models listed above in part A of the jupyter notebook.** Starter code has been provided for data generation and visualization to aid your explorations. We provide you a simple gradient descent framework for you to implement the neural network, but you are also free to use the TensorFlow and PyTorch code from your previous homework and other 3rd party libraries.

(b) In the following parts, we will deal with two types of data, the "regular" data set, and the "shifted" data set. The "regular" data set has the same distribution as the training data set, while the "shifted" data set has a different distribution. **Run part B of the notebook to visualize** the sensor location, the distribution of the "regular" data set, and the distribution of the "shifted" data set. **Attach the plot**.

(c) The starter code generated a set of 7 sensors and the following data sets:

- 15 training sets where n_{train} varies from 10 to 290 in increments of 20.
- A "regular" testing data set where $n_{\text{test}} = 1000$.

[1] This is not covered in the class but you can find the related information in https://www.wikiwand.com/en/Gauss-Newton_algorithm

- A "shifted" testing data set where $n_{\text{test}} = 1000$. You can do this by setting original_dist to `False` in the function `generate_data` in `starter.py`.

Use part C of the notebook to train each of the seven models on each of the fifteen training sets. Use your results to generate three figures (est. run time: 30 min). Each figure should include *all* of the models on the same plot so that you can compare them:

- A plot of *training error* versus n_{train} (the amount of data used to train the model) for all of the models.
- A plot of *testing error* on the "regular" test set versus n_{train} (the amount of data used to train the model) for all of the models.
- A plot of *testing error* on the "shifted" test set versus n_{train} (the amount of data used to train the model) for all of the models.

Briefly describe your observations. What are the strengths and weaknesses of each model? How do they compare to the predict-zero baseline?

(d) We are now going to do some hyper-parameter tuning on our neural network. Fix the number of hidden layers to be two and let ℓ be the number of neurons in each of these two layers. Try values for ℓ between 100 and 500 in increments of 50. Use data sets with $n_{\text{train}} = 200$ and $n_{\text{test}} = 1,000$. **What is the best choice for ℓ (the number of neurons in the hidden layers)? Justify your answer with plots.** The starter code is in part D of the notebook.

(e) We are going to do some more hyper-parameter tuning on our neural network. Let k be the number of hidden layers and let ℓ be the number of neurons in each hidden layer. **Write a formula for the total number of weights in our network in terms of ℓ and k. If we want to keep the total number of weights in the network approximately equal to** 10000**, find a formula for ℓ in terms of k.** Try values of k between 1 and 4 with the appropriate implied choice for ℓ. Use data sets with $n_{\text{train}} = 200$ and $n_{\text{test}} = 1000$. **What is the best choice for k (the number of layers)? Justify your answer with plots.** The starter code is in part E of the notebook.

(f) You might have seen that the neural network performance is disappointing compared to the generative model in the "shifted" data. Try increasing the number of training data and tune the hyper-parameters. Can you get it to generalize to the "shifted" test data? **Attach the "number of training data vs accuracy" plot to justify your conclusion.** What is the intuition how the neural network works on predicting D? The starter kit is provided in part F of the notebook.

4 MLE of Multivariate Gaussian

In lecture, we discussed uses of the multivariate Gaussian distribution. We just assumed that we knew the parameters of the distribution (the mean vector $\boldsymbol{\mu}$ and covariance matrix $\boldsymbol{\Sigma}$). In practice, though, we will often want to estimate μ and Σ from data. (This will come up even beyond regression-type problems: for example, when we want to use Gaussian models for classification problems.) This problem asks you to derive the Maximum Likelihood Estimate for the mean and variance of a multivariate Gaussian distribution.

(a) Let $\mathbf{X}$ have a multivariate Gaussian distribution with mean $\mu \in \mathbb{R}^d$ and covariance matrix $\Sigma \in \mathbb{R}^{d\times d}$. **Write the log likelihood of drawing the n i.i.d. samples $\mathbf{x}_1, \ldots, \mathbf{x}_n \in \mathbb{R}^d$ from X given Σ and μ.**

(b) (*Optional*) **Prove that the maximum likelihood estimates of μ and Σ are the sample mean and covariance.** For taking derivatives with respect to matrices, you may use any formula in "The Matrix Cookbook" without proof. This is a reasonably involved problem part with lots of steps to get to the answer. We recommend students first do the one-dimensional case and then the two-dimensional case to warm up.

Note: Conventions for gradient and derivative in "The Matrix Cookbook" may vary from the conventions we saw in the discussion.

(c) Use the following code to sample from a two-dimensional Multivariate Gaussian and plot the samples:

```
import numpy as np
import matplotlib.pyplot as plt
mu = [15, 5]
sigma = [[20, 0], [0, 10]]
samples = np.random.multivariate_normal(mu, sigma, size=100)
plt.scatter(samples[:, 0], samples[:, 1])
plt.show()
```

Try the following three values of Σ:

$$\Sigma = \begin{bmatrix} 20 & 0 \\ 0 & 10 \end{bmatrix}; \quad \Sigma = \begin{bmatrix} 20 & 14 \\ 14 & 10 \end{bmatrix}; \quad \Sigma = \begin{bmatrix} 20 & -14 \\ -14 & 10 \end{bmatrix}.$$

Calculate the mean and covariance matrix of these distributions from the samples (that is, implement part (b)). Report your results. Include your code in your write-up. Note: you are allowed to use numpy.

5 Regularized and Kernel k-Means

Recall that in k-means clustering we attempt to minimize the objective

$$\min_{C_1, C_2, \ldots, C_k} \sum_{i=1}^{k} \sum_{x_j \in C_i} \|x_j - \mu_i\|_2^2, \text{ where}$$

$$\mu_i = \text{argmin}_{\mu_i \in \mathbb{R}^d} \sum_{x_j \in C_i} \|x_j - \mu_i\|_2^2 = \frac{1}{|C_i|} \sum_{x_j \in C_i} x_j, \quad i = 1, 2, \ldots, k.$$

The samples are $\{x_1, \ldots, x_n\}$, where $x_j \in \mathbb{R}^d$. C_i is the set of sample points assigned to cluster i and $|C_i|$ is its cardinality. Each sample point is assigned to exactly one cluster.

(a) **What is the minimum value of the objective when $k = n$ (the number of clusters equals the number of sample points)?**

(b) (Regularized k-means) Suppose we add a regularization term to the above objective. The objective is now

$$\sum_{i=1}^{k} \left(\lambda \|\mu_i\|_2^2 + \sum_{x_j \in C_i} \|x_j - \mu_i\|_2^2 \right).$$

Show that the optimum of

$$\min_{\mu_i \in \mathbb{R}^d} \lambda \|\mu_i\|_2^2 + \sum_{x_j \in C_i} \|x_j - \mu_i\|_2^2$$

is obtained at $\mu_i = \frac{1}{|C_i|+\lambda} \sum_{x_j \in C_i} x_j$.

(c) Here is a (sadly "dream world") example where we would want to regularize clusters. Suppose there are n students who live in a $\mathbb{R}^2$ Euclidean world and who wish to share rides efficiently to Berkeley for their in-person final exam in EECS189/289A. The university permits k shuttles which may be used for shuttling students to the exam location. The students need to figure out k good locations to meet up. The students will then walk to the closest meet up point and then the shuttles will deliver them to the exam location. Let x_j be the location of student j, and let the exam location be at $(0, 0)$. Assume that we can drive as the crow flies, i.e., by taking the shortest path between two points. **Write down an appropriate objective function to minimize the total distance that the students and vehicles need to travel.** How is this different from the regularized k-means objective above?

(*Hint: your result should be similar to the regularized k-means objective.*)

(d) (Kernel k-means) Suppose we have a dataset $\{x_i\}_{i=1}^n, x_i \in \mathbb{R}^\ell$ that we want to split into k clusters, i.e., finding the best k-means clustering *without the regularization*. Furthermore, suppose we know *a priori* that this data is best clustered in an impractically high-dimensional feature space $\mathbb{R}^m$ with an appropriate metric. Fortunately, instead of having to deal with the (implicit) feature map $\boldsymbol{\phi} : \mathbb{R}^\ell \to \mathbb{R}^m$ and (implicit) distance metric[2], we have a kernel function $\kappa(x_1, x_2) = \langle \boldsymbol{\phi}(x_1), \boldsymbol{\phi}(x_2) \rangle$ that we can compute easily on the raw samples without having to explicitly compute all the features. How should we perform the kernelized counterpart of k-means clustering?

Derive the underlined portion of this algorithm, and show your work in deriving it. The main issue is that although we define the means μ_i in the usual way, we can't ever compute $\boldsymbol{\phi}$ explicitly because it's way too big. Therefore, in the step where we determine which cluster each sample point is assigned to, we must use the kernel function κ to obtain the right result. (Review the lecture on kernels if you don't remember how that's done.)

[2]Just as how the interpretation of kernels in kernelized ridge regression involves an implicit prior/regularizer as well as an implicit feature space, we can think of kernels as generally inducing an implicit distance metric as well. Think of how you would represent the squared distance between two points in terms of pairwise inner products and operations on them.

Algorithm 1 Kernel k-means

Require: Data matrix $X \in \mathbb{R}^{n \times d}$; Number of clusters K; kernel function $\kappa(x_1, x_2)$
Ensure: Cluster class class(j) for each sample x_j.

```
function KERNEL-K-MEANS(X, c)
    Randomly initialize class(j) to be an integer in 1, 2, ..., K for each x_j.
    while not converged do
        for i ← 1 to K do
            Set C_i = {j ∈ {1, 2, ..., n} : class(j) = i}.
        end for
        for j ← 1 to n do
            Set class(j) = argmin_k _____
        end for
    end while
    Return C_i for i = 1, 2, ..., c.
end function
```

(e) The expression you derived may have unnecessary terms or redundant kernel computations. **Explain how to eliminate them**; that is, how to perform the computation quickly without doing irrelevant computations or redoing computations already done.

(f) For this part, we will try to gain more intuition on kernel k-means by running it on real examples and visualizing it. There is no code needed to be written, and you simply have to run the demo to find the "best" kernel and hyperparameters. Of course, you are welcome to add some code if it helps you automate the process. **Please refer to the Jupyter notebook for directions and questions you need to answer.**

6 LDA and QDA on MNIST

All parts of this question are to be completed in the associated Jupyter notebook. The goal of this problem is to make you more familiar with the MNIST dataset as well as to help you understand how to use LDA and QDA on such a standard real-world dataset.

(a) The MNIST dataset consists of 28 by 28 grayscale images of digits, where each pixel can take on integer values from 0 to 255 (inclusive). When trying to learn this dataset using QDA, we need to learn the means μ_i of each class, as well as the covariance matrices Σ_i of each class. Recall that there are 10 classes: 0 through 9. **How many degrees of freedom are there in the QDA model?** (i.e. how many different parameters are there to learn)

For LDA, we learn a single covariance matrix Σ, not one for each class. **How many degrees of freedom are there in our LDA model?**

What does this tell you about the amount of data needed to train a QDA versus an LDA model? In particular, approximately how many images from the MNIST dataset do we need in order to create a QDA classifier? What about an LDA classifier?

(Hint: We don't expect an exact answer, just an order of magnitude with some reasoning.

Think about a linear system with x unknowns. For such a system, think about how many equations we need to solve for all the unknowns uniquely.)

(b) **Implement the `compute_qda_params_single_class` and `compute_qda_params` functions for QDA, and inspect the MLE means of each of the 10 digit classes.**

(c) **Implement the `classify` function to use the learned parameters to classify a vector of test points.**

(Hint: look into using the `slogdet` and `einsum` functions from NumPy to improve performance and avoid floating point errors.)

Then run the subsequent cells and observe your classifier's performance on the training and test sets.

(d) **Implement the `accuracy_vs_n` function** and run the cell to view the training and test accuracy of your classifier as we increase the number of training points made available. **Comment on what you see in the results.**

(e) **Implement the `compute_lda_params` function to implement LDA** wherein you estimate a single covariance matrix for all the classes instead of a covariance matrix per class. Then run the subsequent cells to observe its performance on the training and test sets, as a function of the number of training points made available. **Comment on your results, with reference to your analogous observations for QDA.**

(f) Run the `digit_accuracy_vs_n` function for QDA and LDA to see the per-digit accuracy for both classifiers. Notice how the accuracy is not the same for different digits. **Comment on your observations.**

(g) **Modify the `fuzz` function to treat `eps` as a tunable hyperparameter. Which choice of `eps` is best?** Remember to construct a validation set to evaluate different hyperparameter choices, so the test data is kept hidden until you evaluate the final model.

7 Loss Shaping, Outliers, and Noise

In this problem you will explore how quirks in your training data can cause some of the ML techniques you have already seen, like least-squares regression, to perform poorly. You will then try alternate objective functions and other techniques for shaping your loss landscape to improve your performance. Although this problem is by no means a comprehensive introduction to the many loss shaping techniques used in practice, and doesn't touch on techniques particular to neural networks at all, we hope that you will start to see the kind of analytical thinking that goes into choosing a loss function and modifying basic ML techniques for real problems.

Unless otherwise noted, each of the problem parts will ask you to write some code in the accompanying Jupyter notebook, train a model, and think about the results.

(a) **Classification: MSE**
We begin with a classification problem where our training data has a signficant class imbalance

relative to the test data. This makes it easy to diagnose when our classifier breaks down, but there are many situations where training and test data might have identical class proportions but other considerations, such as prediction performance *per class* or equivalence of outcomes, require changes to the classifier to deal with this imbalance.

To begin **train a linear least-squares classifier, report the test accuracy, and visualize the decision boundary.** Think about why the classifier failed. It may help to consider training a least-squares classifier on the 1D example below, with the decision function being whether the prediction $\hat{f}(x_{test}) > 0$. The two classes are $y \in \{-1, 1\}$.

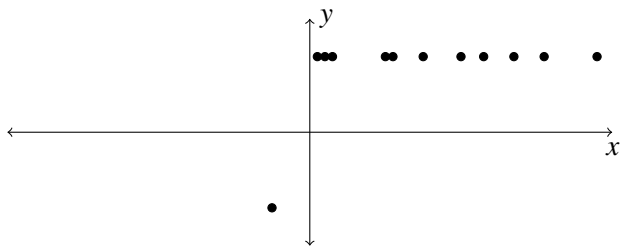

Figure 1: Example 1D classification problem

(b) **Classification: Weighting**

One way to overcome the class imbalance is to add copies of the samples belonging to the lower population class to the training set. **Show that adding c copies of a sample is equivalent to weighting its error by c in the least-squares regression problem. Then create a new training set with enough copies of the underrepresented class samples to equalize the imbalance and retrain your classifier. Report the test accuracy and visualize the decision boundary.**

(c) **Classification: Logistic Regression**

An alternative to modifying the training data is to use *logistic regression*, where we model the probability that a data point corresponds to class 1:

$$h_\theta(\mathbf{x}_i) = P(y_i = 1|\mathbf{x}_i, \theta) = \frac{1}{1 + e^{-\mathbf{x}_i^T\theta}}.$$

This technique is called logistic *regression* because we are learning a regression model with the probability of membership in a class as the target, rather than a $\{1, -1\}$ proxy for class membership. When we need a hard decision we simply choose the class with the highest probability.

θ is typically found using maximum likelihood estimation with the log-likelihood

$$\begin{aligned}\max_\theta \log L(\theta|\mathbf{X}, \mathbf{y}) &= \max_\theta \log\left(\prod_{i=1}^N P(y_i|\mathbf{x}_i, \theta)\right) \\ &= \max_\theta \log\left(\prod_{i=1}^N h_\theta(\mathbf{x}_i)^{y_i}(1 - h_\theta(\mathbf{x}_i))^{(1-y_i)}\right)\end{aligned}$$

$$= \max_{\theta} \sum_{i=1}^{N} \Big(y_i \log h_\theta(\mathbf{x}_i) + (1 - y_i) \log(1 - h_\theta(\mathbf{x}_i)) \Big).$$

Here we use $\mathbf{y} \in \{0, 1\}^N$ rather than $\{-1, 1\}^N$ as with the MSE loss. This loss function may be familiar to some of you as the cross-entropy loss. Here we are actually maximizing the log-likelihood, but we could equivalently minimize the negative log-likelihood.

Use the `LogisticRegression` class from `sklearn` to train a model with the *un*balanced training set, then report the test accuracy and visualize the decision boundary. Then do the same with your copy-balanced training set. Explain which loss function, between logistic loss and MSE loss, is more sensitive to class imbalance and why.

(d) **Regression: Outliers**

Now we will leave classification behind and explore some of the behaviors possible in regression. We will start with a linear model and training data corrupted by outliers from another distribution:

$$\begin{aligned} f(x) &= ax + b \\ y &= \delta(f(x) + \epsilon),\ \epsilon \sim \mathcal{N}(0, \sigma^2) \\ \delta &= \begin{cases} 1 & \text{w.p. } p \\ -5 & \text{w.p. } 1 - p \end{cases} \end{aligned}$$

In the notebook you will fit this data using three loss functions: $L1$, $L2$, and Huber. You should be familiar with the $L2$ loss from least-squares regression. You have seen an $L1$ penalty before in LASSO regression, but it is also possible to train a model with an $L1$ penalty on the errors. The Huber loss is a hybrid of $L1$ and $L2$ loss functions. It acts like the $L2$ loss for 'small' errors less than some threshold, then switches to the $L1$ loss for errors above the threshold. (This should remind you of the relationship between logistic loss and the exponential loss that you saw all the way back in HW0.) This allows the Huber loss to provide a good estimate of the mean while limiting the influence of outliers without totally discounting them. Compare this to our use of OMP from a previous homework, where we used it to remove the influence of outliers entirely by a kind of feature augmentation. Anyway, the Huber loss is

$$L_{Huber}(f(x), y, \delta) = \begin{cases} \frac{1}{2}|f(x) - y|^2, & |f(x) - y| \le \delta \\ \delta|f(x) - y| - \frac{1}{2}\delta^2, & \text{otherwise} \end{cases}$$

Train a model on the data set with outliers using each of the three loss functions. Report the test MSE against the true function, and comment on the effect of the outliers and estimate of the true coefficients in each case.

(e) **Regression: Multiplicative Input Noise**

Now we have an underlying exponential model, but instead of additive output observation noise like you are used to the noise is multiplicative and applied to the input. In other words,

instead of corrupted y values we observe corrupted x values. The ys we observe actually came from different x values than we see in the training data.

$$y = e^{zx},\ z \sim U[1-\epsilon, 1+\epsilon]$$

We will attempt to learn this function with a polynomial model of degree p.

$$\hat{f}(x) = w_0 + \sum_{i=1}^{p} w_i x^i$$

As you may recall, the Taylor expansion of e^x is $1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots$ so we can approximate the true model arbitrarily well with large enough degree p.

Train regressors with $L1$, $L2$, and Huber loss functions and report the test error. Comment on the suitability of each loss function in this situation.

(f) **Regression: Transformed Target**
Since the function we want to learn is exponential, a natural experiment to try is to regress against a log-transform of the y data. This will result in a linear relationship with x *and* reduce the range of noise variances. **Use `sklearn`'s `TransformedTargetRegressor` class to train a ridge regression model on log-transformed y values. Report the test error and comment on the result.**

8 Your Own Question

Write your own question, and provide a thorough solution.

Writing your own problems is a very important way to really learn the material. The famous "Bloom's Taxonomy" that lists the levels of learning is: Remember, Understand, Apply, Analyze, Evaluate, and Create. Using what you know to create is the top-level. We rarely ask you any HW questions about the lowest level of straight-up remembering, expecting you to be able to do that yourself. (e.g. make yourself flashcards) But we don't want the same to be true about the highest level.

As a practical matter, having some practice at trying to create problems helps you study for exams much better than simply counting on solving existing practice problems. This is because thinking about how to create an interesting problem forces you to really look at the material from the perspective of those who are going to create the exams.

Besides, this is fun. If you want to make a boring problem, go ahead. That is your prerogative. But it is more fun to really engage with the material, discover something interesting, and then come up with a problem that walks others down a journey that lets them share your discovery. You don't have to achieve this every week. But unless you try every week, it probably won't happen ever.

Contributors:

- Alvin Wan
- Anant Sahai
- Ashwin Pananjady
- Cathy Wu
- Chawin Sitawarin
- Inigo Incer
- Jianbo Chen
- Jonathan Shewchuk
- Josh Sanz
- Marc Khoury
- Mark Velednitsky
- Michael MacDonald
- Philipp Moritz
- Rahul Arya
- Yichao Zhou