MLPR Assignment 2 – Predicting CT Slice Locations

Due: 12 noon Monday 18 November, 2024

The primary purpose of this assignment is to assess your ability to use a matrix-based computational environment in the context of machine learning methods. The highest marks will go to those who also give excellent *concise* but precise explanations where asked, and do something interesting and well-explained for the last part. Higher marks won't be obtained for answering things that aren't asked for. Keep your answers short and to the point!

Good Scholarly Practice: Please remember the good scholarly practice requirements of the University regarding work for credit. You can find guidance at the School page https://web.inf.ed.ac.uk/infweb/admin/policies/academic-misconduct
This also has links to the relevant University pages. Furthermore, you are required to take reasonable measures to protect your assessed work from unauthorised access. For example, if you put any such work on a public repository then you must set access permissions appropriately (permitting access only to yourself).

You are encouraged to work in pairs on this assignment. Like assignment 1, it should be entirely possible to work by yourself. However, you may work with up to *one* other person. You must let us know if you would like to work by yourself, in a pair of your choice, or would like to be assigned to a random pair on this MS Form by 2pm Tuesday 22 Oct. By asking to work in a pair, you will be committing to submitting an assignment with them, for which you are jointly responsible.

IMPORTANT: Pairs must work jointly on the whole assignment. Like assignment 1, it's a sequence of questions that build on each other, and we hope you'll learn something by doing it all. Splitting up the work, so that parts are done by only one of you, is not acceptable.

Non-sharing policy: You may discuss your high level approach with other pairs or individuals, but you must *never* share code or written materials with other pairs or individuals. Don't post questions on this assignment using the class forum. Email your question to us instead. In response, we might post minor clarifications on the forum, or would email the class list with any major corrections.

Allowable code: You must use Python+NumPy+Matplotlib. You may only use your own code written in the base packages, the support code, and no other libraries. You may not use SciPy, except for the `minimize`, `cho_factor` and `cho_solve` functions imported by the support code and the `scipy.stats.norm.cdf` function.

Make sure to submit your answers as soon as you write them. You can re-submit as many times as you like *until the submission deadline*. Don't leave it until the last minute. Technical difficulties are not a valid reason for late submission.

The deadline and late policy for this assignment are specified on Learn in the "Coursework Planner". Guidance on late submissions is at https://web.inf.ed.ac.uk/node/4533

Queries: If you think there's a problem with the assignment or support code, email Arno rather than asking on Hypothesis. But first, please invest some effort into debugging things yourself, and providing a good bug report[1].

If your query involves code, create a complete but minimal working example of the issue. That is, a few lines of plain text code (no Jupyter notebooks please) that sets everything up and demonstrates the problem. We suggest running the function under question on a tiny toy example, perhaps with input arrays created with `randn`. Then the code will run quickly, and you can look at all the intermediate results at a console or in a debugger.

Background: In this assignment you will perform some preliminary analysis of a dataset from the UCI machine learning repository. Some features have been extracted from slices of CT medical scans. There is a regression task: attempt to guess the location of a slice in the body from features of the scan. A little more information about the data is available online[2]: https://archive.ics.uci.edu/ml/datasets/Relative+location+of+CT+slices+on+axial+axis However, you should use the processed version of the dataset given below.

Download the data here (26 MB).

The patient IDs were removed from this version of the data, leaving 384 input features which were put in each of the "`X_...`" arrays. The corresponding CT scan slice location has been put in the "`y_...`" arrays. We shifted and scaled the "`y_...`" location values for the version of the data that you are using. The shift and scaling was chosen to make the training locations have zero mean and unit variance.

The first 73 patients were put in the `_train` arrays, the next 12 in the `_val` arrays, and the final 12 in the `_test` arrays. Please use this training, validation, test split as given. *Do not shuffle the data further in this assignment*.

The code: We provide support code along with this assignment, which you will need for the questions below. Do not copy this code into the answer boxes; your code should `import` it.

1. **Getting started [15 marks]:**

   As with each question, please report the few lines of code required to perform each requested step. When a question says to report a number, your code should print it, and manually include the resulting number as part of your answer.

   Load the data into Python (your code can assume this step has been done):

   ```
   import numpy as np
   data = np.load('ct_data.npz')
   X_train = data['X_train']; X_val = data['X_val']; X_test =
   data['X_test']
   y_train = data['y_train']; y_val = data['y_val']; y_test =
   data['y_test']
   ```

   a. Verify that (up to numerical rounding errors) the mean of the training positions in `y_train` is zero. The mean of the 5,785 positions in the `y_val` array is not zero. Report its mean with a “standard error”, temporarily assuming that each entry is independent. For comparison, also report the mean with a standard error of the first 5,785 entries in the `y_train`. Explain how your results demonstrate that these standard error bars do not reliably indicate what the average of locations in future CT slice data will be. Why are standard error bars misleading here?

Joint answer shared between s2653520 and s2682783:

Your answer:

```
# Calculate mean and standard error for y_val
y_val_mean = np.mean(y_val)
y_val_std_error = np.std(y_val, ddof=1) / np.sqrt(len(y_val

# Calculate mean and standard error for y_train
y_train_mean = np.mean(y_train)
y_train_std_error = np.std(y_train, ddof=1) / np.sqrt(len(y

# Calculate mean and standard error for the first 5,785 ent
y_train_sample_mean = np.mean(y_train[:5785])
y_train_sample_std_error = np.std(y_train[:5785], ddof=1) /

y_train_mean, y_train_std_error, y_val_mean, y_val_std_erro

#########################
#(-9.13868774539957e-15,#
# 0.0049535309340638205,#
# -0.2160085093241599,  #
# 0.01290449880016868,  #
# -0.44247687859693674, #
# 0.011927303389170828) #
#########################
```

After verification, we got `y_train_mean` is -9.1386877e-15, which can be regarded as 0, and `y_val_mean` is -0.21600851, which is not zero.

We can get the mean with a standard error of `y_val` is **-0.216** and **0.0129**, and the mean with a standard error of the first 5,785 entries in the `y_train` is **-0.442** and **0.0119**

What we expect is that the mean of the first 5785 entries of the `y_train` should be very close to the mean of whole `y_train` (which is zero), since the standard error is pretty tiny, around 0.0049. However, the mean of the first 5785 entries of `y_train` is -0.442, which is not zero.

Explanation: The standard error is calculated under the assumption that each data point is independent. However, in this dataset, the assumption of independence is violated due to the way the data is structured. For example, slice locations from the same patient are likely to be correlated, which means the data points are not independent. In addition, the sampling method was also biased. Specifically, the training, validation, and test sets are split by patients, not randomly shuffled. The first 73 patients are in the training set, the next 12 in validation, and the last 12 in the test set.

Example answer:

```
def errorbar_str(xx):
    xx = np.array(xx)
    mu = xx.mean()
    se = xx.std() / np.sqrt(xx.size)
    print('%.3f +/- %.3f' % (mu, se))
train_pos_mean = np.mean(y_train)
errorbar_str(y_val)
errorbar_str(y_train[:5785])
```

Output:

```
-0.216 +/- 0.013
-0.442 +/- 0.012
```

A number with an error bar can be reported like this: -0.216 ± 0.013. Only quote the standard error to one or two significant figures. Quote the mean to the same number of decimal places as the standard error.

"`-2.16e-01`" is harder to read than "-0.216". If scientific notation is appropriate, documents should format the number as -2.16×10^{-1}. Regardless, don't quote numbers to many significant figures when they're not helpful.

If your numbers weren't correct, review the notes and try again.

An example explanation for this part:

"The mean of the first 5,785 training positions is > 30 estimated standard errors away from zero, and so is misleading about the mean of data later in the training set, let alone totally new data. If the points are positively correlated (nearby points tend to be similar) the variance of the estimator will be larger than estimated:

$$\begin{aligned}\text{var}[\bar{x}] &= \text{var}\left[\frac{1}{N}\sum_{n=1}^{N} x_n\right]\\ &> \frac{1}{N^2}\sum_{n=1}^{N}\text{var}\,[x_n], \quad \text{(positive correlations)}\\ &> \sigma^2/N.\end{aligned}$$

Then even if we have enough data to estimate the population variance σ^2, error bars equal to $\sqrt{\sigma^2/N}$ are too small."

Some good answers plotted the `y_train` data and/or discussed where the data came from to explain why the datapoints are not independent. A good answer could also be based on the "effective number" of datapoints being less than the size of the training set because the data is based on a small number of patients.

Many answers made statements that are generically true about error bars: that they're just estimates. But the *large* mismatch in this question is due to the lack of independence. Poor answers simply asserted that there wasn't enough data. Central limit behaviour happens *rapidly* when the data have a limited range. You can get a feeling for the central limit theorem by working through the suggested exercises in the notes.

Feedback: Well done! Report numbers to three significant digits for better readability (this didn't affect your mark).

Reflection, was your answer good? Different? (not shared with partner):
Add any extra notes

(For the remainder of this assignment you don’t need to report error bars on your results. If you were working on this application in the context of a research problem, you should work on estimating error bars. But given the time available for this assignment, we’re just going to acknowledge that the issue is slightly tricky here, and move on.)

b. Some of the input features are constants: they take on the same value for every training example. Identify these features, and remove them from the input matrices in the training, validation, and testing sets.

   Moreover, some of the input features are duplicates: some of the columns in the training set are identical. For each training set column, discard any later columns that are identical. Discard the same columns from the validation and testing sets.

   Use these modified input arrays for the rest of the assignment. Keep the names of the arrays the same (`X_train`, etc.), so we know what they’re called. You should not duplicate the code from this part in future questions. We will assume it has been run, and that the modified data are available.

   Warning: As in the real world, mistakes at this stage would invalidate all of your results. We strongly recommend checking your code, for example on small test examples where you can see what it's doing.

Report which columns of the `X_...` arrays you remove at each of the two stages. Report these as 0-based indexes. (For the second stage, you might report indexes in the original array, or after you did the first stage. It doesn't matter, as long as your code is clear and correct.)

Joint answer shared between s2653520 and s2682783:

Your answer:

```
constant_columns = [i for i in range(X_train.shape[1]) if n
X_train = np.delete(X_train, constant_columns, axis=1)
X_val = np.delete(X_val, constant_columns, axis=1)
X_test = np.delete(X_test, constant_columns, axis=1)

_, unique_indices = np.unique(X_train, axis=1, return_index
duplicate_columns = [i for i in range(X_train.shape[1]) if
X_train = np.delete(X_train, duplicate_columns, axis=1)
X_val = np.delete(X_val, duplicate_columns, axis=1)
X_test = np.delete(X_test, duplicate_columns, axis=1)

print("Constant columns removed:", constant_columns)
print("Duplicate columns removed:", duplicate_columns)
```

From this code snippet, we removed constant columns `[59, 69, 179, 189, 351]`, and duplicate columns `[76, 77, 185, 195, 283, 354]`

Example answer:

```
mask1 = np.std(X_train, 0) > 0
print('constant features: ', (mask1 != True).nonzero()[0])
# remove [ 59 69 179 189 351]
#mask = mask1
#X_train = X_train[:, mask]
#X_val = X_val[:, mask]
#X_test = X_test[:, mask]

# Take a random combination of the rows. If matches to 32 b
# are probably the same. Yes, I could have been more carefu
rc = np.array(np.dot(np.random.randn(X_train.shape[0]), X_t
        dtype=np.float32)
_, idx = np.unique(rc, return_index=True)
mask2 = np.tile(False, X_train.shape[1])
mask2[idx] = True
print('duplicate features: ', (mask2 != True).nonzero()[0])
print('of which new to remove: ', ((mask2 != True) & mask1)
# remove [ 69  78  79 179 188 189 199 287 351 359]

mask = mask1 & mask2
print('all features removing: ', (mask != True).nonzero()[0
#mask = mask2
X_train = X_train[:, mask]
X_val = X_val[:, mask]
X_test = X_test[:, mask]
D = X_train.shape[1]
```

Output:

```
constant features:  [ 59  69 179 189 351]
duplicate features:  [ 69  78  79 179 188 189 199 287 351 3
of which new to remove:  [ 78  79 188 199 287 359]
all features removing:  [ 59  69  78  79 179 188 189 199 28
```

Most of you had no problems with this part. A small number didn't read the instructions carefully about which duplicates to remove, or based decisions on the validation/test sets, which is poor practice.

Feedback: Well done!

Reflection, was your answer good? Different? (not shared with partner):

Add any extra notes

2. **Linear regression baseline [15 marks]:**

Using `numpy.linalg.lstsq`, write a short function “`fit_linreg(X, yy, alpha)`” that fits the linear regression model

$$f(\mathbf{x}; \mathbf{w}, b) = \mathbf{w}^\top \mathbf{x} + b, \tag{1}$$

by minimizing the cost function:

$$E(\mathbf{w}, b) = \alpha \mathbf{w}^\top \mathbf{w} + \sum_n (f(\mathbf{x}^{(n)}; \mathbf{w}, b) - y^{(n)})^2, \tag{2}$$

with regularization constant α. As discussed in the lecture materials, fitting a bias parameter b and incorporating the regularization constant can both be achieved by augmenting the original data arrays. Use a data augmentation approach that maintains the numerical stability of the underlying `lstsq` solver, rather than a ‘normal equations’ approach. You should only regularize the weights $\mathbf{w}$ and not the bias b.

(In the lecture materials we used λ for the regularization constant, matching Murphy and others. However, `lambda` is a reserved word in Python, so we swapped to `alpha` for our code.)

Use your function to fit weights and a bias to `X_train` and `y_train`. Use $\alpha = 30$.

We can fit the same model with a gradient-based optimizer. The support code has a function `fit_linreg_gradopt`, which you should look at and try.

Report the root-mean-square errors (RMSE) on the training and validation sets for the parameters fitted using both your `fit_linreg` and the provided `fit_linreg_gradopt`. Do you get exactly the same results? Why or why not?

Joint answer shared between s2653520 and s2682783:

Your answer: The root-mean-square errors (RMSE) on the training and validation sets for the parameters fitted using `fit_linreg` is 0.3567565397204054 and 0.4230521968394695, and using `fit_linreg_gradopt` is 0.3567556103401202 and 0.42305510586203865. These results are extremely close but not exactly the same.

Explanation: `fit_linreg` uses a direct least-squares solver, while `fit_linreg_gradopt` likely uses an iterative optimization method like gradient descent. This can lead to minor differences in convergence, especially if the gradient-based optimizer doesn't reach the exact minimum. In addition, iterative methods can introduce small numerical differences depending on convergence criteria, which can cause slight variations in the final values of $\mathbf{w}$ and b.

```
def fit_linreg(X, yy, alpha):
    # Augment X with a column of ones for the bias term
    X_aug = np.hstack((X, np.ones((X.shape[0], 1))))

    # Create an augmented matrix for regularization, leaving bias
    reg_matrix = np.sqrt(alpha) * np.eye(X_aug.shape[1])
    reg_matrix[-1, -1] = 0  # No regularization on the bias term

    # Augment yy to account for the regularization terms
    X_aug_reg = np.vstack((X_aug, reg_matrix))
    yy_reg = np.hstack((yy, np.zeros(X_aug.shape[1])))

    # Use lstsq to solve for weights and bias
    w_aug, _, _, _ = np.linalg.lstsq(X_aug_reg, yy_reg, rcond=Non
    return w_aug[:-1], w_aug[-1]  # Split weights and bias

def rmse(y_true, y_pred):
    return np.sqrt(np.mean((y_true - y_pred) ** 2))

# report fit_linreg
alpha = 30
w, b = fit_linreg(X_train, y_train, alpha)

y_train_pred = X_train @ w + b
y_val_pred = X_val @ w + b

# Calculate RMSE
train_rmse_lstsq = rmse(y_train, y_train_pred)
val_rmse_lstsq = rmse(y_val, y_val_pred)

print("Training RMSE (lstsq):", train_rmse_lstsq)
print("Validation RMSE (lstsq):", val_rmse_lstsq)
```

```
w_gradopt, b_gradopt = fit_linreg_gradopt(X_train, y_train, alpha

# report rmse
y_train_pred = X_train @ w_gradopt + b_gradopt
y_val_pred = X_val @ w_gradopt + b_gradopt

# Calculate RMSE
train_rmse_lstsq = rmse(y_train, y_train_pred)
val_rmse_lstsq = rmse(y_val, y_val_pred)

print("Training RMSE (lstsq) in fit_linreg_gradopt:", train_rmse_
print("Validation RMSE (lstsq) in fit_linreg_gradopt:", val_rmse_
```

Example answer:

```
# Function to report root-mean-square-error (RMSE) to use later:
def rmse(y1,y2):
    return np.sqrt(np.mean((y1-y2)**2))

# Using gradient-based optimizer
alpha = 30
ww1, bb1 = fit_linreg_gradopt(X_train, y_train, alpha)
pred_lr_train = np.dot(X_train, ww1) + bb1
pred_lr_val = np.dot(X_val, ww1) + bb1
print('train_err = %g' % rmse(pred_lr_train, y_train))
print('val_err = %g' % rmse(pred_lr_val, y_val))

# Using least squares solver:
def fit_linreg(X, yy, alpha):
    # Least squares problem, regularizing weights but not bias:
    N, D = X.shape
    X_bias = np.hstack([np.ones((N,1)), X])
    X_reg = np.vstack([X_bias,
            np.hstack([np.zeros((D,1)), np.sqrt(alpha)*np.eye(D)]
    yy_reg = np.hstack([yy, np.zeros(D)])
    params = np.linalg.lstsq(X_reg, yy_reg, rcond=None)[0]
    ww = params[1:]
    bb = params[0]
    return ww, bb
ww2, bb2 = fit_linreg(X_train, y_train, alpha)
pred_lr_train = np.dot(X_train, ww2) + bb2
pred_lr_val = np.dot(X_val, ww2) + bb2
print('train_err = %g' % rmse(pred_lr_train, y_train))
print('val_err = %g' % rmse(pred_lr_val, y_val))
```

Output:

```
train_err = 0.356758
val_err = 0.423057
train_err = 0.356757
val_err = 0.423052
```

You were asked to write a short function "fit_linreg". Many of you

You were asked to write a short function " fit_linreg ". Many of you sensibly used the same interface as the " fit_linreg_gradopt " function that was provided. Then you could call the functions in the same way. Less neat answers returned a single weight vector and then had to repeatedly extract a bias or make augmented input arrays to use it. As an aside, it makes sense to document what value(s) are returned from functions you write, and (if it's not obvious) how they should be used.

Many answers contained small mistakes, for example accidentally regularizing the bias. You could have checked your code on a small random example:

```
N = 5
D = 3
alpha = 10
X = np.random.randn(N, D);
w_true = np.random.randn(D)
b_true = np.random.randn()
yy = X@w_true + b_true + 0.1*np.random.randn(N)
w1, b1 = fit_linreg(X, yy, alpha)
w2, b2 = fit_linreg_gradopt(X, yy, alpha)
print(np.max(np.abs(w1-w2)))
print(np.abs(b1-b2)) # should both be very small
```

In Python, the code gives about 12 significant figures if you add `tol=0` in the call to `minimize` and change the number of iterations to be large.

The standard deviation of the training set was 1.0, so a predictor that guesses zero for every input would get a training RMSE of 1.0. Some answers reported RMSE values bigger than one, and so were "obviously" wrong. There isn't much point working on different methods if every answer is going to be wrong... It's a good idea to write a single routine to report results, and test it carefully.

Generic advice for the future: run small tests before fighting with experiments that take a long time to run, and might generate meaningless results.

An example explanation for this part is:

"Both functions attempt to optimize the same convex cost function, so should obtain nearly the same training cost. `fit_linreg` uses a specialized routine intended to accurately solve the problem. The gradient solver returns weights with a larger cost because it hasn't converged to numerical precision. Optimizers often terminate before fully converging if they reach a maximum number of iterations, or have converged to some target numerical tolerance."

Some answers referred to `lstsq` as a closed-form solution, but it is still a numerical routine, that can have accuracy problems in some cases. Some answers described the gradient-based optimizer as if it was a (stochastic) steepest gradient descent method, perhaps with a fixed step size, which it wasn't.

A few of you derived or looked up the analytic "normal equations" minimum of the regularized cost. However, you were asked to implement the method by augmenting the data. A reference was given in the lecture notes which goes into the precision that can be obtained with the different linear algebra approaches. The "normal equations" approach is often used because it's faster, but it can be less accurate. We want you to be able to work through ways to solve problems that you haven't seen before, not just use a solution that's easily found in textbooks.

Feedback: Good answer. Some details are not quite right. A few decimal places would have been helpful for comparison. The least squares solver is a numerical method, only be accurate to numerical precision too. The cost function is strictly convex (assuming the data is full rank or alpha != 0) so the gradient optimiser should reach the same solution as the least squares solution on convergence.

Reflection, was your answer good? Different? (not shared with partner):

Add any extra notes

3. **Invented classification tasks [20 marks]:**

It is often easier to work with binary data than real-valued data: we don’t have to think so hard about how the values might be distributed, and how we might process them. We will invent some binary classification tasks, and fit these.

We will pick 20 positions within the range of training positions, and use each of these to define a classification task:

```
K = 20 # number of thresholded classification problems to fit
mx = np.max(y_train); mn = np.min(y_train); hh = (mx-mn)/(K+1)
thresholds = np.linspace(mn+hh, mx-hh, num=K, endpoint=True)
for kk in range(K):
    labels = y_train > thresholds[kk]
    # ... fit logistic regression to these labels
```

The logistic regression cost function and gradients are provided with the assignment in the function `logreg_cost`. It is analogous to the `linreg_cost` function for least-squares regression, which is used by the `fit_linreg_gradopt` function that you used earlier.

Fit logistic regression to each of the 20 classification tasks above with $\alpha = 30$.

Given a feature vector, we can now obtain 20 different probabilities, the predictions of the 20 logistic regression models. Transform both the training and validation input matrices into new matrices with 20 columns, containing the probabilities from the 20 logistic regression models. You don't need to loop over the rows of `X_train` or `X_val`, you can use array-based operations to make the logistic regression predictions for every datapoint at once.

Fit a regularized linear regression model ($\alpha = 30$) to your transformed 20-dimensional training set. Report the training and validation root mean square errors (RMSE) of this model.

Joint answer shared between s2653520 and s2682783:

Your answer: Training root mean square errors (RMSE) of this model is 0.15441 and validation RMSE of this model is 0.25425

```
def fit_logreg_gradopt(X, yy, alpha):
    """
    Fit a regularized logistic regression model using gradient op
    """
    D = X.shape[1]
    args = (X, yy, alpha)
    init = (np.zeros(D), np.array(0))
    w_logreg, b_logreg = minimize_list(logreg_cost, init, args)
    return w_logreg, b_logreg

def logreg_k(X, yy, K, alpha=30):
    """
    Trains K binary logistic regression models on data using diff
    """
    mx = np.max(yy); mn = np.min(yy); hh = (mx-mn)/(K+1)
    thresholds = np.linspace(mn+hh, mx-hh, num=K, endpoint=True)

    # weights and biases for the K logistic regression models
    w_logreg_k = np.zeros((K, X_train.shape[1]))
    b_logreg_k = np.zeros((K))

    for kk in range(K):
        # get binary training labels based on thresholds[kk]
        labels = yy > thresholds[kk]

        w_logreg, b_logreg = fit_logreg_gradopt(X, labels, alpha)

        w_logreg_k[kk, :] = w_logreg
        b_logreg_k[kk] = b_logreg

    return w_logreg_k, b_logreg_k

def sigma(a):
    return 1 / (1 + np.exp(-a))

K = 20 # number of thresholded classification problems to fit

# Transform both the training and validation input matrices into
w_logreg_k, b_logreg_k = logreg_k(X_train, y_train, K)
X_train_new = sigma(X_train @ w_logreg_k.T + b_logreg_k)
X_val_new = sigma(X_val @ w_logreg_k.T + b_logreg_k)


w_linreg, b_linreg = fit_linreg_gradopt(X_train_new, y_train, alp

pred_train = X_train_new @ w_linreg + b_linreg
pred_val = X_val_new @ w_linreg + b_linreg
```

```
rmse_train_gd = rmse(y_train, pred_train)
rmse_val_gd = rmse(y_val, pred_val)

print('Training RMSE:', rmse_train_gd)
print('Validation RMSE:', rmse_val_gd)
```

Example answer:

```
# Fitting classifiers
alpha = 30
K = 20 # number of thresholded classification problems to fit
mx = np.max(y_train); mn = np.min(y_train); hh = (mx-mn)/(K+1)
thresholds = np.linspace(mn+hh, mx-hh, num=K, endpoint=True)
V_lr = np.zeros((K, D))
bb_lr = np.zeros(K)
for kk in range(K):
    print('Fitting classifier %d / %d' % (kk+1, K))
    labels = y_train > thresholds[kk]
    args = (X_train, labels, alpha)
    V_lr[kk,:], bb_lr[kk] = minimize_list(
            logreg_cost, (np.zeros(D), np.array(0)), args)

# Fitting dataset transformed using classifiers
P_fn = lambda X: 1/(1 + np.exp(-(np.dot(X,V_lr.T) + bb_lr[None,:]
P_train = P_fn(X_train)
P_val = P_fn(X_val)
w_p, b_p = fit_linreg(P_train, y_train, alpha)
pred_p_train = np.dot(P_train, w_p) + b_p
pred_p_val = np.dot(P_val, w_p) + b_p
print('train_p_err = %g' % rmse(pred_p_train, y_train))
print('val_p_err = %g' % rmse(pred_p_val, y_val))
```

Output:

```
train_p_err = 0.154412
val_p_err = 0.254248
```

Most people had no trouble with the code for this part. If you had trouble, have another go. You can't claim to be a machine learning expert if you can't fit logistic regression and do something with the results.

Feedback: Well done! The code is well structured and easy to read. RMSE values are reported correctly.

Reflection, was your answer good? Different? (not shared with partner):

Add any extra notes

4. **Small neural network [10 marks]:** In Question 3 you fitted a small neural network. The logistic regression classifiers are sigmoidal hidden units, and a linear output unit predicts the outputs. However, you didn't fit the parameters jointly to the obvious least squares cost function. A least squares cost function and gradients for this neural network are implemented in the `nn_cost` function provided.

   Try fitting the neural network model to the training set, with a) a sensible random initialization of the parameters; b) the parameters initialized using the fits made in Q3.

   Does one initialization strategy work better than the other? Does fitting the neural network jointly work better than the procedure in Q3? Your explanation should include any numbers that your answer is based on.

Joint answer shared between s2653520 and s2682783:

Your answer: 1. Does one initialization strategy work better than the other?

Yes, a sensible random initialization of the parameters（here we use the Glorot Initialization) performs slightly better than Q3 Initialization. While both achieve similar training RMSEs (0.1383 for Glorot vs. 0.1391 for Q3), Glorot Initialization yields a lower validation RMSE (0.2677 vs. 0.2691), indicating better generalization.

Glorot Initialization starts with randomly distributed weights and biases scaled to balance the input and output dimensions. This flexibility allows the model to explore a broader parameter space during optimization, leading to a better fit to the validation data. On the other hand, Q3 Initialization leverages pre-trained parameters from the logistic and linear regression models, which provide a more informed starting point. While this can help optimization converge faster, it may limit the model’s ability to find a globally optimal solution. As a result, Glorot Initialization is more effective in this case.

2. Does fitting the neural network jointly work better than the procedure in Q3?

No, fitting the neural network jointly in Q4 does not work better than the procedure in Q3. While the neural network achieves a lower training RMSE (0.1383 in neural network vs. 0.1544 in Q3), its validation RMSE is higher (0.2677 in Q4 vs. 0.2542 in Q3), indicating worse generalization performance.

The two-step procedure in Q3 separates feature generation (via logistic regression) and final prediction (via linear regression), effectively limiting the model's flexibility and reducing overfitting. Each logistic regression classifier independently generates features that are regularized, and the linear regression model applies additional regularization to the transformed features, ensuring a robust fit. This modular structure balances simplicity and performance.

In contrast, Q4’s joint optimization increases flexibility by fitting all parameters simultaneously, allowing the network to better minimize training error. However, this flexibility also increases the risk of overfitting, as the model becomes more tailored to the training data, at the expense of generalization. While neural networks are theoretically more powerful, the increased capacity needs careful regularization and sufficient data to avoid overfitting, which was less effectively controlled in Q4.

```
np.random.seed(42)
```

```
K = 20
D = X_train.shape[1]
alpha = 30


def fit_nn_gradopt(X, yy, alpha, init):
    """
    Fit a regularized logistic regression model using gradient op
    """
    args = (X, yy, alpha)
    ww, bb, V, bk = minimize_list(nn_cost, init, args)
    return ww, bb, V, bk

# Glorot Initialization
def glorot_init(n_in, n_out):
    limit = np.sqrt(6 / (n_in + n_out))
    return np.random.uniform(-limit, limit, size=(n_out, n_in))

def init_nn_parameter_glorot_uniform(D, K):
    ww_init = glorot_init(K, 1).flatten()
    bb_init = np.array(0)
    V_init = glorot_init(D, K)
    bk_init = np.zeros(K)

    init = (ww_init, bb_init, V_init, bk_init)
    return init


# the parameters initialized using the fits made in Q3.
init = (w_linreg, b_linreg, w_logreg_k, b_logreg_k)
params_opt = fit_nn_gradopt(X_train, y_train, alpha, init)

F_train = nn_cost(params_opt, X_train)
F_val = nn_cost(params_opt, X_val)
rmse_train = rmse(F_train, y_train)
rmse_val = rmse(F_val, y_val)
print("Q3 Initialization:")
print('Training RMSE:', rmse_train)
print('Validation RMSE:', rmse_val)


# a sensible random initialization of the parameters: glorot unif
init = init_nn_parameter_glorot_uniform(D, K)
params_opt_glorot = fit_nn_gradopt(X_train, y_train, alpha, init)

F_train = nn_cost(params_opt_glorot, X_train)
F_val = nn_cost(params_opt_glorot, X_val)
rmse_train_glorot = rmse(F_train, y_train)
rmse_val_glorot = rmse(F_val, y_val)
print("Glorot Initialization:")
print("Training RMSE:", rmse_train_glorot)
print("Validation RMSE:", rmse_val_glorot)
```

Example answer:

```
# Neural network, random init
K = 20 # number of hidden units
ww = 0.01 * np.random.randn(K) / np.sqrt(K)
bb = np.array((0))
V = 0.01 * np.random.randn(K, D) / np.sqrt(D)
bk = np.zeros(K)
init = (ww, bb, V, bk)
args = (X_train, y_train, alpha)
params = minimize_list(nn_cost, init, args)
pred_nn_train = nn_cost(params, X_train)
pred_nn_val = nn_cost(params, X_val)
print('train_nn_err = %g' % rmse(pred_nn_train, y_train))
print('val_nn_err = %g' % rmse(pred_nn_val, y_val))

# Neural network, init with Q4
ww = w_p
bb = b_p
V = V_lr
bk = bb_lr
init = (ww, bb, V, bk)
params2 = minimize_list(nn_cost, init, args)
pred_nn_train = nn_cost(params2, X_train)
pred_nn_val = nn_cost(params2, X_val)
print('train_nn2_err = %g' % rmse(pred_nn_train, y_train))
print('val_nn2_err = %g' % rmse(pred_nn_val, y_val))
```

Output:

```
train_nn_err = 0.145658
val_nn_err = 0.27602
train_nn2_err = 0.139663
val_nn2_err = 0.267812
```

Most of you worked out how to use the parameters from Q3 to initialize the neural network. Some of you only used a subset of the parameters you'd fitted in Q3, and set the rest of the neural net parameters to arbitrary values.

We accepted most random initializations of the parameters that set the main weights to small random values, and that didn't set any of the parameters large. Using `randn` or `rand` with no scaling isn't in general a good initialization strategy. Although for this fairly small net it could work ok. However, just unfixing this short-coming wasn't seen as an "innovation" in Q6.

If using uniform random numbers (e.g., with `rand`) rather than Gaussian (e.g., with `randn`) they should usually be shifted to be zero mean. Some of you trying to use ReLUs in Q6 had more problems than usual after initializing all weights to be positive.

Many of you set a weight going into one of the K hidden units using some multiple of `randn()/sqrt(K)`. If the number of inputs D grew, the hidden units would saturate. We would suggest to use a multiple of `randn()/sqrt(D)`. Or use $\sqrt{D+K}$ as advocated by Glorot and Bengio (2010). Some of you divided by D or K, rather than their square-root, which is hard to justify. Some of you divided by something related to $D \times K$; again hard to justify.

Some of you made emphatic statements about differences between the procedures based on a single experiment, with no reliable errorbars. To us, the procedures seemed to be roughly the same. Several excellent students tried the random initialization more than once before saying anything too concrete about it.

Similarly we didn't get large differences in performance by fitting the neural network jointly. Some random initializations led to better validation performance, but others don't. It's possible joint fitting is better if done carefully. But a final test set comparison is probably required if trying multiple fits on the validation set.

Most of you read the support code and realized that you were given code to make predictions with the neural network. Most of you used that code for this question, but your own prediction code in Q3. It would have been sensible to check that these two implementations gave the same results. Several of you would have caught mistakes with such a check. In your future research you should be looking out for checks that you can do.

Feedback: Good answer. Report numbers to three significant digits for better readability. (Just advice, you didn't lose any marks for this.) Your numbers look reasonable, but you don't provide much in terms of an explanation for your results. Training error should not be used to compare methods.

Reflection, was your answer good? Different? (not shared with partner):

Add any extra notes

5. **Bayesian optimisation [20 marks]:**

A popular application area of Gaussian processes is Bayesian optimisation, where the uncertainty in the probabilistic model is used to guide the optimisation of a function. Here we will use Bayesian optimisation with Gaussian processes for choosing the regularisation parameter α. (We would normally use Bayesian optimisation when optimizing more than one parameter.)

Gaussian processes are used to represent our belief about an unknown function. In this case, the function we are interested in is the neural network's validation log root mean square error (log RMSE) as a function of the regularisation paramter α. In Bayesian optimisation, it is common to maximise the unknown function, so we will maximise the negative log RMSE.

We start with a Gaussian process prior over this function. As we observe the actual log RMSEs for particular α's we update our belief about the function by calculating the Gaussian process posterior.

Besides the Gaussian process framework that you're already familiar with, Bayesian optimisation involves a so-called acquisition function. Given our Gaussian process posterior model, we use this function to decide which parameter to query next. The acquisition function describes how useful we think it will be to try a given α, while considering the uncertainty that is represented in our posterior belief.

There are many popular acquisition functions in Bayesian optimisation. One example is the *probability of improvement*. Suppose we have observed $y^{(1)}$ to $y^{(N)}$ (here negative log RMSE at locations $\alpha^{(1)}$ to $\alpha^{(N)}$). Then the function takes the following form:

$$PI(\alpha) = \Phi\left(\frac{\mu(\alpha) - \max(y^{(1)}, \ldots, y^{N})}{\sigma(\alpha)}\right),$$

where $\mu(\alpha)$ is the Gaussian process posterior mean at location α, $\sigma(\alpha)$ is the posterior standard deviation at location α, and Φ denotes the cumulative density function of the Gaussian with mean 0 and variance 1.

We pick the next regularization constant $\alpha^{(N+1)}$ by maximizing the acquisition function:

$$\alpha^{(N+1)} = \arg\max_{\alpha} PI(\alpha).$$

We then evaluate our model for this regularization parameter and update our posterior about the unknown function that maps α to negative log RMSE. We repeat the procedure multiple times and then pick the parameter that yielded the best performance y.

Write a function `train_nn_reg` that trains the neural network from Q4 for a given α parameter and returns the validation RMSE.

Consider α on the range `np.arange(0, 50, 0.02)`. Pick three values from this set for α as training locations and use `train_nn_reg` on these locations. Use the remaining locations as values that you will consider with the acquisition function.

Take the performance of the neural network that you trained in Q4 a) as a baseline. Subtract your α-observed log RMSEs from the log of this baseline and take the resulting values as $y^{(1)}$ to $y^{(3)}$. Then calculate the Gaussian process posterior for these observations. To do so, use `gp_post_par` from the support code. Use the default parameters for `sigma_y`, `ell` and `sigma_f`.

Implement the probability of improvement acquisition function. You can use `scipy.stats.norm.cdf` to calculate Φ. With this acquisition function, iteratively pick new α's, re-train and evaluate each new model with `train_nn_reg` and update your posterior with `gp_post_par`. Do five of these iterations.

Report the maximum probability of improvement together with its α for each of the five iterations. Also report the best α that this procedure found, and its validation and test RMSEs. Have we improved the model?

In this question, the function we are optimizing is the neural network's validation log RMSE as a function of the regularisation parameter α. Where is the observation noise coming from?

Joint answer shared between s2653520 and s2682783:

Your answer: The maximum probability of improvement together with its α for each of the five iterations is shown as following:

Iteration 1: max PI = 0.3456598992062415, alpha = 1.4000000000000001, rmse = 0.24236906530412483

Iteration 2: max PI = 0.3289169724466733, alpha = 3.0, rmse = 0.2390091620873626

Iteration 3: max PI = 0.3119210281373339, alpha = 4.2, rmse = 0.2691166609767137

Iteration 4: max PI = 0.2836414373162618, alpha = 0.0, rmse = 0.26697369255713543

Iteration 5: max PI = 0.13054690444082412, alpha = 38.2, rmse = 0.26764074374958785

Additionally, Best alpha found: 3.0, **Validation RMSE** with best alpha: 0.2390091620873626, **Test RMSE** with best alpha: 0.2682501995281824

Compared to the validation rmse from Q4, we did improve the model.

The observation noise probably comes from the following sources. Firstly is the randomness during training. Training of neural networks usually involves random weight initialisation (e.g., Glorot initialisation), which can affect the performance performance of the final model, resulting in slightly different RMSE values from one training to another. Secondly is about Gradient descent fluctuations. During gradient descent, especially in SGD, there may be some fluctuations in the training path, leading to unstable final model performance.

```
def train_nn_reg(alpha, X_train, y_train, X_val, y_val):
    K = 20
    D = X_train.shape[1]

    # Initialize parameters
    V_init_glorot = glorot_init(D, K)
    ww_init_glorot = glorot_init(K, 1).flatten()
    bk_init_glorot = np.zeros(K)
    bb_init_glorot = 0.0

    params_init_glorot = [ww_init_glorot, bb_init_glorot, V_init_

    # Set optimization arguments
    args = (X_train, y_train, alpha)

    # Train the neural network
    params_opt_glorot = minimize_list(nn_cost, params_init_glorot
    y_val_pred = nn_cost(params_opt_glorot, X_val)

    rmse_val_glorot = rmse(y_val, y_val_pred)

    return rmse_val_glorot

from scipy.stats import norm

def acquisition_function_PI(mu, sigma, y_max):
    with np.errstate(divide='warn'):
        Z = (mu - y_max) / sigma
        PI_values = norm.cdf(Z)
    return PI_values

baseline_rmse = rmse(F_val, y_val) # Calculated in Q4
baseline_log_rmse = np.log(baseline_rmse)

# initial observation negative log RMSE
np.random.seed(42)
initial_alphas = np.random.choice(np.arange(0, 50, 0.2), size=3,
initial_log_rmses = [np.log(train_nn_reg(alpha, X_train, y_train,
initial_y = baseline_log_rmse - np.array(initial_log_rmses)

print("Initial alphas and observed log RMSE differences (y values
for alpha, y in zip(initial_alphas, initial_y):
    print(f"alpha = {alpha}, y = {y}")

remaining_alphas = np.setdiff1d(np.arange(0, 50, 0.2), initial_al
rest_mu, rest_cov = gp_post_par(remaining_alphas, initial_alphas,

best_alpha = None
best_rmse = float('inf')
observed_alphas = list(initial_alphas)
observed_ys = list(initial_y)

for iteration in range(5):
    # get std from posterior
    rest_std = np.sqrt(np.diag(rest_cov))

    y_max = max(observed_ys)

    PI_values = acquisition_function_PI(rest_mu, rest_std, y_max)

    # select alpha with max PI
    max_PI_index = np.argmax(PI_values)
    next_alpha = remaining_alphas[max_PI_index]
    max_PI = PI_values[max_PI_index]

    # train_nn_reg on next alpha
    next_rmse = train_nn_reg(next_alpha, X_train, y_train, X_val,
    next_log_rmse = np.log(next_rmse)
    next_y = baseline_log_rmse - next_log_rmse
    # update observed alphas and ys
    observed_alphas.append(next_alpha)
    observed_ys.append(next_y)

    # transform to numpy arrays
    observed_alphas_np = np.array(observed_alphas)
    observed_ys_np = np.array(observed_ys)

    # update GP posterior
    rest_mu, rest_cov = gp_post_par(remaining_alphas, observed_al

    print(f"Iteration {iteration + 1}: max PI = {max_PI}, alpha =

    # update best alpha
    if next_rmse < best_rmse:
        best_rmse = next_rmse
        best_alpha = next_alpha

print("\nBest alpha found:", best_alpha)
print("Validation RMSE with best alpha:", best_rmse)

rmse_test = train_nn_reg(best_alpha, X_train, y_train, X_test, y_
print("Test RMSE with best alpha:", rmse_test)
```

Example answer:

```
# Baseline for preprocessing the GP observations
bl = rmse(nn_cost(params, X_val), y_val)

def train_nn_reg(alpha):
    ww = 0.01 * np.random.randn(K) / np.sqrt(K)
    bb = np.array((0))
    V = 0.01 * np.random.randn(K, D) / np.sqrt(D)
    bk = np.zeros(K)
    init = (ww, bb, V, bk)
    args = (X_train, y_train, alpha)
    params = minimize_list(nn_cost, init, args)
    pred_nn_train = nn_cost(params, X_train)
    pred_nn_val = nn_cost(params, X_val)
    train_nn_err = rmse(pred_nn_train, y_train)
    val_nn_err = rmse(pred_nn_val, y_val)

    return params, val_nn_err

Alpha_grid = np.arange(0, 50, 0.02)
N_grid = Alpha_grid.size

# Pick three initial grid points as training locations
idx = np.round(N_grid * np.array([0.25, 0.5, 0.75])).astype(int)
Alpha_obs = Alpha_grid[idx]
N_obs = idx.size
yy = np.zeros(N_obs)
```

```
params_obs = []
for nn in range(N_obs):
    params, val_nn_err = train_nn_reg(Alpha_obs[nn])
    yy[nn] = np.log(bl) - np.log(val_nn_err)
    params_obs.append(params)
Alpha_rest = np.delete(Alpha_grid, idx)

N_opt = 5
for nn in range(N_opt):
    print('Fitting network %d / %d' % (nn+1, N_opt))

    rest_cond_mu, rest_cond_cov = gp_post_par(Alpha_rest, Alpha_o
    rest_cond_std = np.sqrt(np.diag(rest_cond_cov))

    # Probability of improvement acquisition function
    acq_pi = norm.cdf((rest_cond_mu - np.max(yy)) / rest_cond_std

    idx_next = np.argmax(acq_pi)
    alpha_next = Alpha_rest[idx_next]
    print('Best pi: %g at alpha=%g' % (acq_pi[idx_next], alpha_ne

    params, val_nn_err = train_nn_reg(alpha_next)
    Alpha_obs = np.append(Alpha_obs, alpha_next)
    yy = np.append(yy, np.log(bl) - np.log(val_nn_err))
    params_obs.append(params)
    Alpha_rest = np.delete(Alpha_rest, idx_next)

# Select best location
idx_opt = np.argmax(yy)
Alpha_opt = Alpha_obs[idx_opt]
params_opt = params_obs[idx_opt]

print('Alpha_opt = %g' % Alpha_opt)

print('train_nn3_err = %g' % rmse(nn_cost(params_opt, X_train), y
print('val_nn3_err = %g' % rmse(nn_cost(params_opt, X_val), y_val

# Final evaluation on test set
print('test_nn3_err = %g' % rmse(nn_cost(params_opt, X_test), y_t
print('test_p_err = %g' % rmse(np.dot(P_fn(X_test), w_p) + b_p, y
```

Output:

```
Fitting network 1 / 5
Best pi: 0.265913 at alpha=14.1
Fitting network 2 / 5
Best pi: 0.635535 at alpha=9.64
Fitting network 3 / 5
Best pi: 0.428255 at alpha=7.94
Fitting network 4 / 5
Best pi: 0.354475 at alpha=7.82
Fitting network 5 / 5
Best pi: 0.211338 at alpha=7.7
Alpha_opt = 7.82
```

```
train_nn3_err = 0.0878738
val_nn3_err = 0.248163
test_nn3_err = 0.270842
test_p_err = 0.285135
```

This question required more code than the previous questions and took longer to run. So it was important to devise sensible checks while writing the answer. The code needed to keep track of the observed points and of the remaining points.

Most of you got the main parts right with minor shortcomings.

Some of you didn't report the validation and test RMSEs as requested in the question. Always make sure to answer everything that is asked for.

Some used the Gaussian process on the training error instead of the validation error. α is a hyperparameter which should be selected on the validation set.

The Gaussian process assumes zero mean. Subtracting a baseline was crucial for having a function scale in line with the support code. The baseline had to be subtracted not just from the initial three observations but from all observations.

Some of you realized that we don't need the `norm.cdf` function. If we dropped it from the equation and code, we've just monotonically stretched the acquisition function: we'd still pick the same points to explore.

Regarding the question about the source of the observation noise, we are modelling the neural network's validation log RMSE as a function of the regularisation parameter α. For this function, the only source of randomness is the neural network weights' random initialization, which is where the observation noise is coming from in this case. If we would fix the initial weights, we would always get the same observations for a given α and thus have zero observation noise.

Feedback: Well done on implementing the Bayesian optimisation procedure! Test errors not compared between questions. To assess whether the Q5 method is the best method, compare the test RMSE values between the Q5 method and the test RMSE score of the previous best performer on the validation dataset (e.g Test RMSE of Q3). Your answer regarding the source of the observation noise is partially correct, but can you elaborate further on why random weight initialization causes this? Gradient descent does not introduce randomness here as we are using a fixed max_iter value. We are not guaranteed to find an optimal solution, but we would get consistent results with fixed initializations.

Reflection, was your answer good? Different? (not shared with partner):

Add any extra notes

6. **What next? [20 marks]** Try to improve regression performance beyond the methods we have tried so far. Explain what you tried, why you thought it might work better, how you evaluated your idea, and what you found.

   Do not write more than 300 words for this part, not including your code (which you do need to include). The bulk of the marks available for this part are for motivating and trying something sensible, which can be simple, and evaluating it sensibly.

The constraints on allowable code given at the start of the assignment still apply. This question is about motivating simple ideas that you can implement from scratch.

Your method does not have to work better to get some marks! However, you need to actually run something and report its results to get some marks. Also, your motivation should be sensible: don't pick an idea that your existing results already indicate are unlikely to work.

We expect that most students will get less than 10 marks on this question: some of the tasks in this assignment are straightforward and the overall mark on the common marking scheme should reserve marks in the 80s and 90s for truly outstanding work. However, we advise you not to spend a huge amount of time on this final part. If you are taking 120 credits of courses, this part is worth 0.8% of your mark average. Lots of extra effort might nudge your mark average by 0.2% or less.

Joint answer shared between s2653520 and s2682783:

Your answer: To improve regression performance, we extended the neural network by adding a second hidden layer(the first experiment) and used Bayesian Optimization (BO) to to tune three hyperparameters(the second experiment). We hypothesized that the increased depth would enable the network to learn more complex, hierarchical patterns, while BO would efficiently identify optimal hyperparameters.

We expected using a two-hidden-layer network better because it can capture both low-level and high-level feature interactions, leading to more accurate predictions. Additionally, BO optimizes parameters such as the regularization strength alpha and layer sizes K1, K2, which are critical for balancing model complexity and generalization.

Therefore, we trained the two-hidden-layer neural network on the training set and computed the RMSE on the validation and test sets. We compared these results to those from the previous single-hidden-layer network. We found that the two-hidden-layer network achieved a lower RMSE on both the validation and test sets:

Validation RMSE: Reduced from 0.23935 (single layer) to 0.22696.

Test RMSE: Reduced from 0.27282 (single layer) to 0.25061. These results suggest that the additional hidden layer improved the model's ability to generalize.

```
# 2-layers neural network
def nn2_cost(params, X, yy=None, alpha=None):
    """NN2_COST neural network with two hidden layers cost functi

       E, params_bar = nn2_cost([ww, bb, V1, bk1, V2, bk2], X, yy
                pred = nn2_cost([ww, bb, V1, bk1, V2, bk2], X)

      Inputs:
              params (ww, bb, V1, bk1, V2, bk2), where:
                     ------------------------------------
                         ww K2,  hidden-output weights
                         bb      scalar output bias
                         V1 K1,D first hidden-input weights
                         bk1 K1, first hidden biases
                         V2 K2,K1 second hidden-hidden weights
                         bk2 K2, second hidden biases
                     ------------------------------------
                   X N,D input design matrix
                  yy N,  regression targets
               alpha     scalar regularization for weights

      Outputs:
```

```
 outputs:
                 E  sum of squares error
        params_bar  gradients wrt params, same format as para
 OR
           pred N,  predictions if only params and X are give
"""
# Unpack parameters from list
ww, bb, V1, bk1, V2, bk2 = params

# Forward computation
# First hidden layer
A1 = np.dot(X, V1.T) + bk1[None, :]  # N x K1
P1 = 1 / (1 + np.exp(-A1))  # N x K1

# Second hidden layer
A2 = np.dot(P1, V2.T) + bk2[None, :]  # N x K2
P2 = 1 / (1 + np.exp(-A2))  # N x K2

# Output layer
F = np.dot(P2, ww) + bb  # N,

if yy is None:
    return F

# Compute cost
res = F - yy  # N,
E = np.dot(res, res) + alpha * (np.sum(V1 * V1) + np.sum(V2 *

# Reverse computation of gradients
F_bar = 2 * res  # N,

# Gradients for output layer
ww_bar = np.dot(P2.T, F_bar) + 2 * alpha * ww  # K2,
bb_bar = np.sum(F_bar)  # Scalar

# Gradients for second hidden layer
P2_bar = np.outer(F_bar, ww)  # N x K2
A2_bar = P2_bar * P2 * (1 - P2)  # N x K2

V2_bar = np.dot(A2_bar.T, P1) + 2 * alpha * V2  # K2 x K1
bk2_bar = np.sum(A2_bar, axis=0)  # K2

# Gradients for first hidden layer
P1_bar = np.dot(A2_bar, V2)  # N x K1
A1_bar = P1_bar * P1 * (1 - P1)  # N x K1

V1_bar = np.dot(A1_bar.T, X) + 2 * alpha * V1  # K1 x D
bk1_bar = np.sum(A1_bar, axis=0)  # K1

# Pack gradients into a tuple matching params
params_bar = (ww_bar, bb_bar, V1_bar, bk1_bar, V2_bar, bk2_ba

return E, params_bar
```

```
def fit_nn2_gradopt(X, yy, alpha, init):
    args = (X, yy, alpha)
    params = minimize_list(nn2_cost, init, args)
    return params

# Training the two-hidden-layer neural network
H1 = 20  # Number of units in the first hidden layer
H2 = 10  # Number of units in the second hidden layer
alpha = best_alpha  # Using the best alpha found earlier

# Initialize parameters
def init_nn2_parameters(D, H1, H2):
    V1 = glorot_init(D, H1)  # K1 x D
    bk1 = np.zeros(H1)
    V2 = glorot_init(H1, H2)  # K2 x K1
    bk2 = np.zeros(H2)
    ww = glorot_init(H2, 1).flatten()  # K2
    bb = 0.0
    return [ww, bb, V1, bk1, V2, bk2]

init_params = init_nn2_parameters(X_train.shape[1], H1, H2)

# Train the network
params_opt = fit_nn2_gradopt(X_train, y_train, alpha, init_params

# Evaluate on validation set
F_val = nn2_cost(params_opt, X_val)
rmse_val = rmse(y_val, F_val)
print("Validation RMSE with two hidden layers:", rmse_val)

# Evaluate on test set
F_test = nn2_cost(params_opt, X_test)
rmse_test = rmse(y_test, F_test)
print("Test RMSE with two hidden layers:", rmse_test)
```

Next, we applied BO to simultaneously optimize alpha, $K1$, and $K2$, using a coarser grid due to computational constraints: alpha in $[0, 50)$, step size 5, $K1$ in $[50, 100)$, step size 5, $K2$ in $[5, 25)$, step size 5.

Due to computational resource limitations, we used a relatively coarse parameter grid. The best parameters $alpha = 15$, $K1 = 25$, $K2 = 15$ achieved a Validation RMSE of 0.25159 and a Test RMSE of 0.28427. While slightly worse than the single-parameter optimization, this reflects the limitations of coarse-grained tuning. Finer granularity and additional resources could yield better results.

```
from scipy.linalg import cho_factor, cho_solve

def gp_post_par(X_rest, X_obs, yy, sigma_y=0.05, ell=5.0, sigma_f
    K_rest = gauss_kernel_fn(X_rest, X_rest, ell, sigma_f)
    K_rest_obs = gauss_kernel_fn(X_rest, X_obs, ell, sigma_f)
    K_obs = gauss_kernel_fn(X_obs, X_obs, ell, sigma_f)
    M = K_obs + sigma_y**2 * np.eye(len(yy))

    # Cholesky decomposition
    M_cho, M_low = cho_factor(M)

    # Posterior mean
    rest_cond_mu = np.dot(K_rest_obs, cho_solve((M_cho, M_low), y

    # Posterior covariance
    rest_cond_cov = K_rest - np.dot(K_rest_obs, cho_solve((M_cho,

    return rest_cond_mu, rest_cond_cov


def train_nn_reg(alpha, K1, K2, X_train, y_train, X_val, y_val):
    D = X_train.shape[1]
    params_init = init_nn_parameters_two_layers(D, K1, K2)
    params_opt_glorot = minimize_list(nn_cost_two_layers, params_

    y_val_pred = nn_cost_two_layers(params_opt_glorot, X_val)

    rmse_val_glorot = rmse(y_val, y_val_pred)

    return rmse_val_glorot

# Initial observations: Randomly select 3 parameter combinations
np.random.seed(42)
D = X_train.shape[1]

initial_params = np.array([
    [10, 60, 10],  # (alpha, K1, K2)
    [15, 25, 15],
    [20, 30, 20]
])
initial_ys = np.array([
    train_nn_reg(alpha, K1, K2, X_train, y_train, X_val, y_val)
    for alpha, K1, K2 in initial_params
])

# Remaining parameter combinations to evaluate
param_grid = np.array([[a, k1, k2] for a in np.arange(0, 50, 5)
                                   for k1 in np.arange(50, 100,
                                   for k2 in np.arange(5, 25, 5
remaining_params = np.array([p for p in param_grid if list(p) not

# Initialize GP
observed_params = initial_params
observed_ys = initial_ys

for iteration in range(5):
    # GP posterior
    mu, cov = gp_post_par(remaining_params, observed_params, obse
    # Compute standard deviation
    sigma = np.sqrt(np.diag(cov))

    # Current best observed value
    y_max = np.max(observed_ys)

    # Acquisition function: Probability of Improvement (PI)
    PI_values = acquisition_function_PI(mu, sigma, y_max)

    # Select next parameters to evaluate (maximizing PI)
    next_index = np.argmax(PI_values)
    next_params = remaining_params[next_index]

    print(f"Iteration {iteration + 1}: Testing parameters {next_p

    # Evaluate model
    next_rmse = train_nn_reg(next_params[0], next_params[1], next
                             X_train, y_train, X_val, y_val)

    # Update observations
    observed_params = np.vstack([observed_params, next_params])
    observed_ys = np.append(observed_ys, next_rmse)

    # Remove selected point from remaining parameters
    remaining_params = np.delete(remaining_params, next_index, ax

# Best parameters
best_index = np.argmin(observed_ys)
best_params = observed_params[best_index]
print("\nBest parameters found:")
print(f"alpha = {best_params[0]}, K1 = {best_params[1]}, K2 = {be
print("Validation RMSE:", observed_ys[best_index])

# Final test evaluation
rmse_test = train_nn_reg(best_params[0], best_params[1], best_par
                         X_train, y_train, X_test, y_test)
print("Test RMSE with best parameters:", rmse_test)
```

In conclusion, the two-hidden-layer model improved generalization compared to the single-layer network. BO-based multi-parameter tuning showed promise but was constrained by grid granularity.

Example answer: There were many things you could have done in this part. Some marks were given to those who tried something simple but sensible, with a clear motivation and interpretation, and without any major issues with machine learning practice. Using the training/validation/test sets appropriately was essential for a middling mark. Higher marks went to those who wrote clear answers, followed the word limit and code restrictions, carefully thought about what they were going to do, picked something interesting, and executed it well. As the single open-ended part of the assignment, it was much harder to get anything like full marks on this one small part. If you got more than half marks, we were impressed with at least some of your answer.

Trying the augmented inputs with the neural networks was an obvious simple thing to try. Some people also augmented the inputs with other basis functions. If trying to improve regression performance, it would make sense to try inputs augmented with basis functions in the neural network, rather than just with linear regression.

The choice of regularization α and/or the number of hidden units K could be tuned. The best settings should be chosen on a validation set.

Some people got better results with Gaussian processes, but had to subsample their data so they didn't run out of memory. Predictions from multiple Gaussian processes (or any models) could be averaged.

Some people tried different non-linearities, deeper networks, and/or different regularizers. Very little code needs changing in `nn_cost` to try some of these things, and you should know how. While in the real world you'd use a proper neural network toolbox, you were asked not to. Knowing how to make small changes to provided code is a necessary skill if you ever want to try something non-standard.

Weaker answers just tried large networks, reported they didn't work better (or much better) and stopped. Some good answers diagnosed that other settings might have to change, for example how long to run the optimizer for. Some of you found that altering the code to support early stopping helped. Some of you realized that some larger models required running the optimizer for longer.

A few people wrote their own stochastic gradient optimizer. Some of these worked much worse than the support code. Others seemed to fit models that generalized significantly better.

A few people departed from neural networks entirely, some successfully and some disastrously. For those trying ideas that didn't work, using their features in the neural networks might have worked, or combining the approaches in some other way.

The results on the validation set were often quite variable, making it difficult to make firm conclusions (although some answers made firm conclusions anyway!). Some answers realized the problem, and made choices based on multiple runs. Some reduced variability (and usually improved performance) by averaging predictions over multiple model fits.

If you tried a few things, a final comparison on the test set would be sensible. If you picked anything from a grid of settings on the validation set, you should

If you picked anything from a grid of settings on the validation set, you should *definitely* have done a test set comparison. Reporting the test set score of just your best model is hard to interpret; the test set might turn out to be harder or easier than the validation set. While you don't want to evaluate too many things on the test set, do report some baselines' test performance for comparison.

Feedback: Reasonable idea. Good work on implementing and evaluating your ideas. Why do you think you need a more powerful NN here? In Q4, you thought that it already might be overfitting, so making it more powerful would just make that problem worse. But in that case, it's interesting why you still got lower validation and test RMSE.

Reflection, was your answer good? Different? (not shared with partner):

Add any extra notes

---

1. Simon Tatham (1999) gives good guidance: https://www.chiark.greenend.org.uk/~sgtatham/bugs.html↩

2. Some of the description on the UCI webpage doesn't seem to match the dataset. For example there are 97 unique patient identifiers in the dataset, although apparently there were only 74 different patients.↩

Notes by Iain Murray and Arno Onken