# COMP9417 - Machine Learning
# Homework 2: Logistic Regression & Optimization

**Introduction** In this homework, we will first explore some aspects of Logistic Regression and performing inference for model parameters. We then turn our attention to gradient based optimisation, the workhorse of modern machine learning methods.

**Points Allocation** There are a total of 25 marks. The available marks are:

- Question 1 a): 2 marks
- Question 1 b): 3 marks
- Question 1 c): 2 marks
- Question 1 d): 3 marks
- Question 1 e): 1 mark
- Question 2 a): 2 marks
- Question 2 b): 3 marks
- Question 2 c): 1 mark
- Question 2 d): 0 marks
- Question 2 e): 3 marks
- Question 2 f): 3 marks
- Question 2 g): 2 marks

**What to Submit**

- A single PDF file which contains solutions to each question. For each question, provide your solution in the form of text and requested plots. For some questions you will be requested to provide screen shots of code used to generate your answer — only include these when they are explicitly asked for.
- **.py file(s) containing all code you used for the project, which should be provided in a separate .zip file.** This code must match the code provided in the report.
- You may be deducted points for not following these instructions.
- You may be deducted points for poorly presented/formatted work. Please be neat and make your solutions clear. Start each question on a new page if necessary.

- You **cannot** submit a Jupyter notebook; this will receive a mark of zero. This does not stop you from developing your code in a notebook and then copying it into a .py file though, or using a tool such as **nbconvert** or similar.
- We will set up a Moodle forum for questions on this homework. Please read the existing questions before posting new questions. Please do some basic research online before posting questions. Please only post clarification questions. Any questions deemed to be *fishing* for answers will be ignored and/or deleted.
- Please check the Moodle forum for updates to this spec. It is your responsibility to check for announcements about the spec.
- Please complete your homework on your own, do not discuss your solution with other people in the course. General discussion of the problems is fine, but you must write out your own solution and acknowledge if you discussed any of the problems in your submission (including their name and zID).
- As usual, we monitor all online forums such as Chegg, StackExchange, etc. Posting homework questions on these site is equivalent to plagiarism and will result in a case of academic misconduct.

**When and Where to Submit**

- Due date: Week 7, Sunday **July 18th**, 2021 by **11:55pm**. Please note that the forum will not be actively monitored on weekends.
- Late submissions will incur a penalty of 20% per day (from the ceiling, i.e., total marks available for the homework) for the first 5 days. For example, if you submit 2 days late, the maximum possible mark is 60% of the available 25 marks.
- Submission must be done through Moodle, no exceptions.

**Question 1. Regularized Logistic Regression & the Bootstrap**

In this problem we will consider the dataset provided in `Q1.csv`, with binary response variable $Y$, and 45 continuous features $X_1, \ldots, X_{45}$. Recall that Regularized Logistic Regression is a regression model used when the response variable is binary valued. Instead of using mean squared error loss as in standard regression problems, we instead minimize the log-loss, also referred to as the cross entropy loss. For a parameter vector $\beta = (\beta_1, \ldots, \beta_p) \in \mathbb{R}^p$, $y_i \in \{0, 1\}$, $x_i \in \mathbb{R}^p$ for $i = 1, \ldots, n$, the log-loss is

$$L(\beta, \beta_0) = \sum_{i=1}^{n} y_i \ln\left(\frac{1}{s(\beta_0 + \beta^T x_i)}\right) + (1 - y_i) \ln\left(\frac{1}{1 - s(\beta_0 + \beta^T x_i)}\right),$$

where $s(z) = (1 + e^{-z})^{-1}$ is the logistic sigmoid (see Homework 0 for a refresher.) In practice, we will usually add a penalty term, and consider the optimisation:

$$(\hat{\beta}_0, \hat{\beta}) = \arg\min_{\beta_0, \beta} \{CL(\beta_0, \beta) + \text{penalty}(\beta)\} \tag{1}$$

where the penalty is usually not applied to the bias term $\beta_0$, and $C$ is a hyper-parameter. For example, in the $\ell_1$ regularisation case, we take penalty$(\beta) = \|\beta\|_1$ (a LASSO for logistic regression).

(a) Consider the `sklearn` logistic regression implementation (section 1.1.11), which claims to minimize the following objective:

$$\hat{w}, \hat{c} = \arg\min_{w,c} \left\{ \|w\|_1 + C \sum_{i=1}^{n} \log(1 + \exp(-\tilde{y}_i(w^T x_i + c))) \right\}. \tag{2}$$

It turns out that this objective is identical to our objective above, but only after re-coding the binary variables to be in $\{-1, 1\}$ instead of binary values $\{0, 1\}$. That is, $\tilde{y}_i \in \{-1, 1\}$, whereas $y_i \in \{0, 1\}$. Argue rigorously that the two objectives (1) and (2) are identical, in that they give us the same solutions ($\hat{\beta}_0 = \hat{c}$ and $\hat{\beta} = \hat{w}$). Further, describe the role of $C$ in the objectives, how does it compare to the standard LASSO parameter $\lambda$? *What to submit: some commentary/your working.*

**Solution:**

We can focus on a single $i$ for simplicity, note that the $i$-th summand in 1 is equivalent to

$$\begin{aligned}
&y_i \ln(1 + \exp(-\beta^T x_i - \beta_0)) + (1 - y_i) \ln\left(1 + \exp(\beta^T x_i + \beta_0)\right) \\
&= \begin{cases} \ln\left(1 + \exp(-1 \times (\beta^T x_i + \beta_0))\right) & \text{if} \quad y = 1 \\ \ln\left(1 + \exp(1 \times (\beta^T x_i + \beta_0))\right) & \text{if} \quad y = 0 \end{cases} \\
&= \begin{cases} \ln\left(1 + \exp(-\tilde{y}_i \times (\beta^T x_i + \beta_0))\right) & \text{if} \quad \tilde{y}_i = 1 \\ \ln\left(1 + \exp(-\tilde{y} \times (\beta^T x_i + \beta_0))\right) & \text{if} \quad \tilde{y} = -1 \end{cases} \\
&= \ln\left(1 + \exp(-\tilde{y}(\beta^T x_i + \beta_0))\right).
\end{aligned}$$

$C$ attaches higher importance to the 'fit'term, so as $C$ increases, we care more about fitting than the penalty, and so $C$ plays an inverse role to that of $\lambda$ in standard LASSO, i.e. $C \propto 1/\lambda$. This is a standard trick used to rewrite loss functions, see for example the SVM objective from lectures.

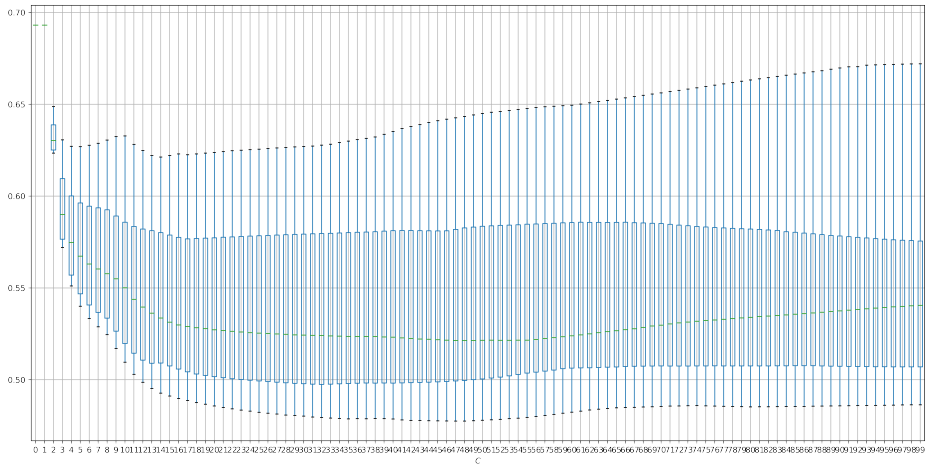
(b) Take the **first** 500 observations to be your training set, and the rest as the test set. In this part, we will perform cross validation over the choice of $C$ from scratch (**Do not use existing cross validation implementations here, doing so will result in a mark of zero.**)

Create a grid of 100 $C$ values ranging from $C = 0.0001$ to $C = 0.6$ in equally sized increments, inclusive. For each value of $C$ in your grid, perform 10-fold cross validation (i.e. split the data into 10 folds, fit logistic regression (using the `LogisticRegression` class in `sklearn`) with the choice of $C$ on 9 of those folds, and record the log-loss on the 10th, repeating the process 10 times.) For this question, we will take the first fold to be the first 50 rows of the training data, the second fold to be the next 50 rows, etc. Be sure to use $\ell_1$ regularisation, and the `liblinear` solver when fitting your models.

To display the results, we will produce a plot: the x-axis should reflect the choice of $C$ values, and for each $C$, plot a box-plot over the 10 CV scores. Report the value of $C$ that gives you the best CV performance. Re-fit the model with this chosen $C$, and report both train and test *accuracy* using this model. Note that we do **not** need to use the $\widetilde{y}$ coding here (the `sklearn` implementation is able to handle different coding schemes automatically) so no transformations are needed before applying logistic regression to the provided data. *What to submit: a single plot, train and test accuracy of your final model, a screen shot of your code for this section, a copy of your python code in solutions.py*

**Solution:**

The optimal choice of $C$ is $\hat{C} = 0.1879$. The train and test accuracy are $0.752$ and $0.74$ respectively. The plot produced is below:




```
            import pandas as pd
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import log_loss, accuracy_score

df = pd.read_csv("Q1.csv", index_col=0)

X = df.iloc[:, :45].to_numpy()
Y = df.Y.to_numpy()

```

```
ii = 500
Xtrain = X[:ii]
Xtest = X[ii:]
Ytrain = Y[:ii]
Ytest = Y[ii:]

N = Xtrain.shape[0]
Cs = np.linspace(0.0001, 0.6, 100)
nC = Cs.shape[0]
K = 10
n_fold = N//K
idxs = np.zeros(shape=(K, n_fold))
for i in range(K):
    idxs[i] = np.arange(i*n_fold, (i+1)*n_fold)
scores = np.zeros(shape=(nC, K))

for nc in range(nC):
    for k in range(K):

        # get idxs for train / test
        cur_test_idx = np.int32(idxs[k])
        cur_train_idx = np.int32(np.delete(idxs, k, axis=0).reshape(-1))

        # fit model
        mod = LogisticRegression(penalty='l1', solver='liblinear', C=Cs[nc])
        mod.fit(Xtrain[cur_train_idx], Ytrain[cur_train_idx].ravel())

        # predictions
        pred_prob = mod.predict_proba(Xtrain[cur_test_idx])
        ll = log_loss(Ytrain[cur_test_idx], pred_prob)
        #acc = accuracy_score(Ytrain[cur_test_idx], preds)

        # scores
        scores[nc, k] = ll

colnames = [f's{i}' for i in range(1, K+1)]
df = pd.DataFrame(scores, columns = colnames)
df['n'] = np.arange(0, Cs.shape[0])
tdf = df.set_index('n').T
tdf.boxplot(figsize=(20,10))
plt.xlabel('$C$')
plt.savefig('figures/cv_scores.png', dpi=500)
plt.show()

Cmin = scores.mean(axis=1).argmin()
Cmin = Cs[Cmin]
Cmin

final_mod = LogisticRegression(
    penalty='l1', solver='liblinear', C=Cmin).fit(Xtrain, Ytrain)

print("train acc: ", accuracy_score(Ytrain, final_mod.predict(Xtrain)))
print("test acc: ", accuracy_score(Ytest, final_mod.predict(Xtest)))

```

(c) In this part we will compare our results in the previous section to the `sklearn` implementation of gridsearch, namely, the `GridSearchCV` class. My initial code for this section looked like:

```
    grid_lr = GridSearchCV(estimator=
                           LogisticRegression(penalty='l1',
                                              solver='liblinear'),
                           cv=10,
                           param_grid=param_grid)
    grid_lr.fit(Xtrain, Ytrain)

```

However, this gave me a very different answer to the result in (b). Provide two reasons for why this is the case, and then, if it is possible, re-run the code with some changes to give consistent results to those in (b), and if not, explain why. It may help to read through the documentation. *What to submit: some commentary, a screen shot of your code for this section, a copy of your python code in solutions.py*

**Solution:**
There are two main resons for the discrepancy:

1. The obvious one is that `GridSearchCV` will choose a different way of splitting the data into folds, this can be remedied through the use of the `KFold` class in `sklearn` and passing that to the `cv` argument.
2. The default scoring used by `GridSearchCV` seems to focus on accuracy, and not maximising the negative log-loss.

The improved code which returns consistent results to part (b) is:

```
        from sklearn.model_selection import KFold
        from sklearn.model_selection import GridSearchCV
        param_grid = {'C': Cs}
        grid_lr = GridSearchCV(estimator=LogisticRegression(penalty='l1',
                                                            solver='liblinear'),
                               scoring='neg_log_loss',
                               cv=KFold(n_splits=K),
                               param_grid=param_grid)
        grid_lr.fit(Xtrain, Ytrain.ravel())
        print(grid_lr.best_params_['C'])

```

We next explore the idea of inference. To motivate the difference between prediction and inference, see some of the answers to this stats.stachexchange post. Needless to say, inference is a much more difficult problem than prediction in general. In the next parts, we will study some ways of quantifying the uncertainty in our estimates of the logistic regression parameters. **Assume for the remainder of this question that $C = 1$, and work only with the training data set ($n = 500$ observations) constructed earlier**.

(d) In this part, we will consider the nonparametric bootstrap for building confidence intervals for each of the parameters $\beta_1, \ldots, \beta_p$. (**Do not use existing Bootstrap implementations here, doing so will result in a mark of zero.**) To describe this method, let's first focus on the case of $\hat{\beta}_1$. The idea behind the nonparametric bootstrap is as follows:

1. Generate $B$ bootstrap samples from the original dataset. Each bootstrap sample consists of $n$ points sampled **with replacement** from the original dataset, where $n$ is the size of the original dataset.

2. On each of the $B$ bootstrap samples, compute an estimate of $\beta_1$, giving us a total of $B$ estimates which we denote $\tilde{\beta}_1^{(1)}, \ldots, \tilde{\beta}_1^{(B)}$.
3. Define the bootstrap mean and standard error respectively:

$$\tilde{\beta}_1 = \frac{1}{B}\sum_{b=1}^{B} \tilde{\beta}_1^{(b)} \qquad \widetilde{\text{s.e.}}(\tilde{\beta}_1) = \sqrt{\frac{1}{B-1}\sum_{b=1}^{B}(\tilde{\beta}_1^{(b)} - \tilde{\beta}_1)^2}.$$

4. A 90% bootstrap confidence interval for $\beta_1$ is then given by the interval:

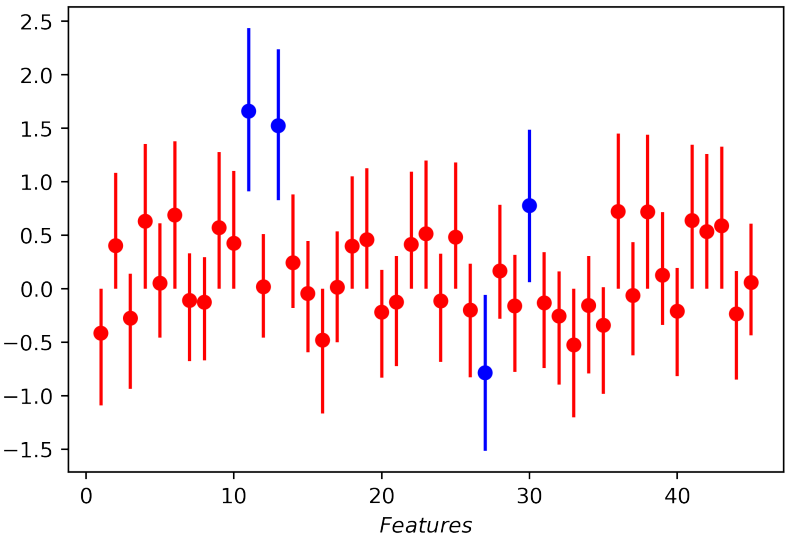
$$((\tilde{\beta}_1)_L, (\tilde{\beta}_1)_U) = (\text{5th quantile of the bootstrap estimates}, \text{95th quantile of the bootstrap estimates})$$

The idea behind a 90% confidence interval is that it gives us a range of values for which we believe with 90% probability the true parameter lives in that interval. If the computed 90% interval contains the value of zero, then this provides us evidence that $\beta_1 = 0$, which means that the first feature should not be included in our model.

Take $B = 10000$ and set a random seed of 12 (i.e. `np.random.seed(12)`). Generate a plot where the $x$-axis represents the different parameters $\beta_1, \ldots, \beta_p$, and plot a vertical bar that runs from $(\tilde{\beta}_p)_L$ to $(\tilde{\beta}_p)_U$. For those intervals that contain 0, draw the bar in red, otherwise draw it in blue. Also indicate on each bar the bootstrap mean. Remember to use $C = 1.0$.

*What to submit: a single plot, a screen shot of your code for this section, a copy of your python code in solutions.py*

**Solution:**



```
            np.random.seed(12)
            B=10000

        for b in range(B):
            b_sample = np.random.choice(np.arange(Xtrain.shape[0]),
                                        size=Xtrain.shape[0])
            Xtrain_b = Xtrain[b_sample]
            Ytrain_b = Ytrain[b_sample]
            mod = LogisticRegression(penalty='l1',
                            solver='liblinear',
                            C=1).fit(Xtrain_b, Ytrain_b)
            coef_mat[b,:] = mod.coef_

        means = np.mean(coef_mat, axis=0)
        lower = np.quantile(coef_mat, 0.05, axis=0)
        upper = np.quantile(coef_mat, 0.95, axis=0)
        colors = ['red' if lower[i] <= 0 and upper[i] >= 0 else 'blue' for i in range(
    p)]

        plt.vlines(x=np.arange(1,p+1), ymin=lower, ymax=upper, colors=colors)
        plt.scatter(x=np.arange(1,p+1), y=means, color=colors)
        plt.xlabel("$Features$")
        plt.savefig("figures/NPBootstrap.png", dpi=400)
        plt.show()
```

(e) Comment on your results in the previous section, what do the confidence intervals tell you about the underlying data generating distribution? How does this relate to the choice of $C$ when running regularized logistic regression on this data? Is regularization necessary?

**Solution:**
The intervals constructed tell us that a large number (most) of the coefficients in the model are plausibly zero. It is therefore a very good idea to regularize the logistic regression fit and we would likely need a value of $C \ll 1$ to get a model that is sparse and doesn't include many of the noisy features.

**Question 2. Gradient Based Optimization**

In this question we will explore some algorithms for *gradient* based optimization. These algorithms have been crucial to the development of machine learning in the last few decades. The most famous example is the backpropagation algorithm used in deep learning, which is in fact just an application of a simple algorithm known as (stochastic) gradient descent. The general framework for a gradient method for finding a minimizer of a function $f : \mathbb{R}^n \to \mathbb{R}$ is defined by

$$x^{(k+1)} = x^{(k)} - \alpha_k \nabla f(x_k), \qquad k = 0, 1, 2, \ldots, \tag{3}$$

where $\alpha_k > 0$ is known as the step size, or learning rate. Consider the following simple example of minimizing $g(x) = 2\sqrt{x^3 + 1}$. We first note that $g'(x) = 3x^2(x^3 + 1)^{-1/2}$. We then need to choose a starting value of $x$, say $x^{(0)} = 1$. Let's also take the step size to be constant, $\alpha_k = \alpha = 0.1$. Then we have

the following iterations:

$$x^{(1)} = x^{(0)} - 0.1 \times 3(x^{(0)})^2((x^{(0)})^3 + 1)^{-1/2} = 0.7878679656440357$$
$$x^{(2)} = x^{(1)} - 0.1 \times 3(x^{(1)})^2((x^{(1)})^3 + 1)^{-1/2} = 0.6352617090300827$$
$$x^{(3)} = 0.5272505146487477$$
$$\vdots$$

and this continues until we terminate the algorithm (as a quick exercise for your own benefit, code this up and compare it to the true minimum of the function which is $x_* = -1$). This idea works for functions that have vector valued inputs, which is often the case in machine learning. For example, when we minimize a loss function we do so with respect to a weight vector, $\beta$. When we take the step-size to be constant at each iteration, this algorithm is called gradient descent. For the entirety of this question, **do not use any existing implementations of gradient methods, doing so will result in an automatic mark of zero for the entire question.**

(a) Consider the following optimisation problem:

$$\min_{x \in \mathbb{R}^n} f(x),$$

where

$$f(x) = \frac{1}{2}\|Ax - b\|_2^2,$$

and where $A \in \mathbb{R}^{m \times n}$, $b \in \mathbb{R}^m$ are defined as

$$A = \begin{bmatrix} 1 & 0 & 1 & -1 \\ -1 & 1 & 0 & 2 \\ 0 & -1 & -2 & 1 \end{bmatrix}, \qquad b = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}.$$

Run gradient descent on $f$ using a step size of $\alpha = 0.1$ and starting point of $x^{(0)} = (1, 1, 1, 1)$. You will need to terminate the algorithm when the following condition is met: $\|\nabla f(x^{(k)})\|_2 < 0.001$. In your answer, clearly write down the version of the gradient steps (3) for this problem. Also, print out the first 5 and last 5 values of $x^{(k)}$, clearly indicating the value of $k$, in the form:

$$k = 0, \qquad x^{(k)} = [1, 1, 1, 1]$$
$$k = 1, \qquad x^{(k)} = \cdots$$
$$k = 2, \qquad x^{(k)} = \cdots$$
$$\vdots$$

*What to submit: an equation outlining the explicit gradient update, a print out of the first 5 and last 5 rows of your iterations, a screen shot of any code used for this section and a copy of your python code in solutions.py.*

**Solution:**

The gradient is $\nabla f(x) = A^T Ax - A^T b$, so the steps are

$$x^{(k+1)} = x^{(k)} - \alpha_k (A^T Ax - A^T b), \qquad k = 0, 1, 2, \ldots.$$

The first 5 steps are:

$$\begin{aligned}
k &= 0, \quad & x_k &= [1, 1, 1, 1] \\
k &= 1, \quad & x_k &= [1., 0.5, 0., 1.5] \\
k &= 2, \quad & x_k &= [1.2, 0.25, -0.25, 1.45] \\
k &= 3, \quad & x_k &= [1.345, 0.125, -0.36, 1.44] \\
k &= 4, \quad & x_k &= [1.4565, 0.0625, -0.4075, 1.459] \\
k &= 5, \quad & x_k &= [1.5499, 0.03125, -0.4242, 1.49205].
\end{aligned}$$

The last 5 steps are:

$$\begin{aligned}
k &= 217, \quad & x_k &= [3.99690261, 6.00384406 \times 10^{-16}, -5.79471060 \times 10^{-4}, 2.99806155] \\
k &= 218, \quad & x_k &= [3.99699850, 6.00384406 \times 10^{-16}, -5.61531549 \times 10^{-4}, 2.99812156] \\
k &= 219, \quad & x_k &= [3.99709142, 6.00384406 \times 10^{-16}, -5.44147417 \times 10^{-4}, 2.99817971] \\
k &= 220, \quad & x_k &= [3.99718146, 6.00384406 \times 10^{-16}, -5.27301471 \times 10^{-4}, 2.99823607] \\
k &= 221, \quad & x_k &= [3.99726872, 6.00384406 \times 10^{-16}, -5.10977048 \times 10^{-4}, 2.99829068].
\end{aligned}$$

So the final value is $[4, 0, 0, 3]$. The code used is:

```
A = np.array([[1,0,1,-1], [-1,1,0,2], [0,-1,-2,1]])
b = np.array([1,2,3])

def f(x):
    return 0.5 * np.linalg.norm(A@x - b, ord=2)**2

def grad_f(x):
    return A.T @ (A@x - b)

alpha = 0.1
tol = 10**-3

xk = np.array([1,1,1,1])
k = 0
while np.linalg.norm(grad_f(xk)) > tol:
    print(f'k = {k}, x_k= {xk}')
    xk = xk - alpha * grad_f(xk)
    k += 1

```

(b) Note that using a constant step-size is sub-optimal. Ideally we would ideally like to take large steps at the beginning (when we are far away from the optimum), then take smaller steps as we move closer towards the minimum. There are many proposals in the literature for how best to choose the step size, here we will explore just one of them called the method of steepest descent.

This is almost identical to gradient descent, except at each iteration $k$, we choose

$$\alpha_k = \arg\min_{\alpha \geq 0} f(x^{(k)} - \alpha \nabla f(x^{(k)})).$$

In words, the step size is chosen to minimize an objective at each iteration of the gradient method, the objective is different at each step since it depends on the current $x$-value. In this part, we will run steepest descent to find the minimizer in (a). First, derive an explicit solution for $\alpha_k$ (mathematically, please show your working). Then run steepest descent with the same $x^{(0)}$ as in (a), and $\alpha_0 = 0.1$. Use the same termination condition. Provide the first and last 5 values of $x^{(k)}$, as well as a plot of $\alpha_k$ over all iterations. *What to submit: a derivation of $\alpha_k$, a print out of the first 5 and last 5 rows of your iterations, a single plot, a screen shot of any code used for this section, a copy of your python code in solutions.py.*

**Solution:**
First, for ease of notation we drop the $k$ superscripts. We note that

$$\begin{aligned} f(x - \alpha \nabla f(x)) &= \frac{1}{2}\|Ax - \alpha A \nabla f(x) - b\|^2 \\ &\overset{\alpha}{\propto} \frac{1}{2}\alpha^2 \|A\nabla f(x)\|_2^2 - \alpha x^T A^T A \nabla f(x) + \alpha b^T A \nabla f(x), \end{aligned}$$

so this is just a quadratic in $\alpha$. Note that the proportionality symbol here means that I am throwing away any terms in the function that do not include $\alpha$. The reason this is okay is because when I differentiate with respect to $\alpha$, these terms will all be zero anyway. So using proportionality makes the maths easier, though it is not necessary to solving the problem. Differentiating and setting to zero yields

$$\alpha = \frac{x^T A^T A \nabla f(x) - b^T A \nabla f(x)}{\|A \nabla f(x)\|_2^2}.$$

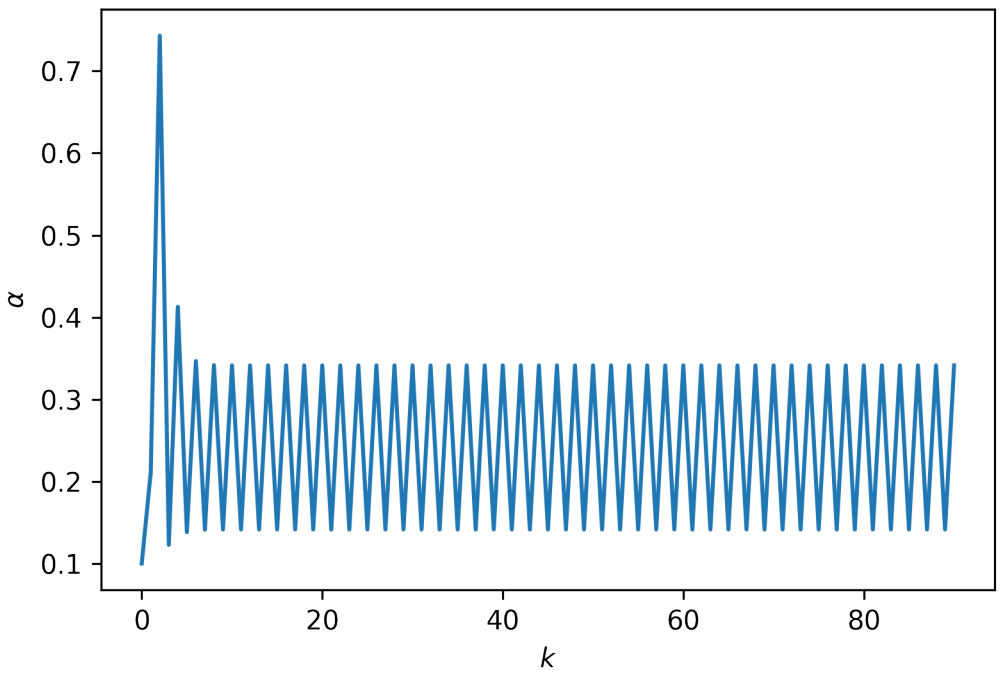
The first 5 steps are:

$$\begin{aligned} k = 0, &\quad x_k = [1, 1, 1, 1] \\ k = 1, &\quad x_k = [1., 0.5, 0., 1.5] \\ k = 2, &\quad x_k = [1.42271293, -0.02839117, -0.52839117, 1.39432177] \\ k = 3, &\quad x_k = [2.04509976, 0.07709813, -0.18730912, 1.65101238] \\ k = 4, &\quad x_k = [2.06071834, 0.02978136, -0.34806892, 1.84620026] \\ k = 5, &\quad x_k = [2.38888907, -0.03168527, -0.28257453, 1.85898232]. \end{aligned}$$

The last 5 steps are:

$$\begin{aligned} k = 85, &\quad x_k = [3.99688335, -1.40691763 \times 10^{-16}, -4.87786042 \times 10^{-4}, 2.99785893] \\ k = 86, &\quad x_k = [3.99692548, -1.40691763 \times 10^{-16}, -6.10498584 \times 10^{-4}, 2.99814648] \\ k = 87, &\quad x_k = [3.99733476, 1.09033223 \times 10^{-17}, -4.17136083 \times 10^{-4}, 2.99816903] \\ k = 88, &\quad x_k = [3.99737079, -5.19049161 \times 10^{-17}, -5.22075184 \times 10^{-4}, 2.99841494] \\ k = 89, &\quad x_k = [3.99772079, -5.19049161 \times 10^{-17}, -3.56718923 \times 10^{-4}, 2.99843423]. \end{aligned}$$

The plot of the step sizes:




The code used:

```
    def alpha_k(xk):
        gfx = grad_f(xk)
        num1 = (xk).T @ A.T @ A @gfx
        num2 = b.T @ A @ gfx
        num = num1 - num2
        den = np.linalg.norm(A @ gfx)**2
        return num/den

    alphas = [0.1]
    tol = 10**-3

    xk = np.array([1,1,1,1])
    k = 0

    while np.linalg.norm(grad_f(xk)) > tol:
        print(f'k = {k}, x_k= {xk}')
        xk = xk - alphas[k] * grad_f(xk)
        alphas.append(alpha_k(xk))
        k += 1

    alphas = np.array(alphas)

```

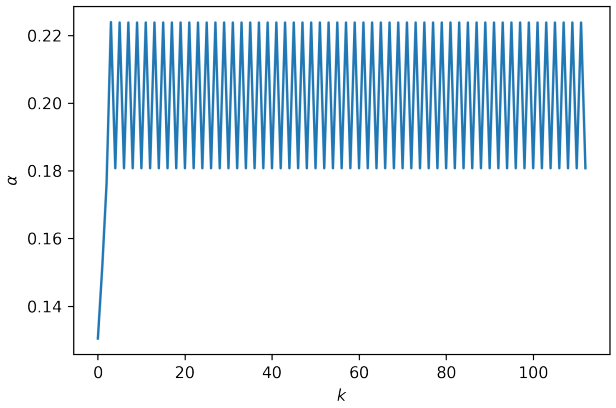
```
    plt.plot(np.arange(k+1), alphas)
    plt.ylabel(r"$\alpha$")
    plt.xlabel(r"$k$")
    plt.savefig('figures/alpha_plot_b.png', dpi=300)
    plt.show()

```

note: the instruction here of taking $\alpha_0 = 0.1$ was unnecessary, as one could solve for $\alpha_0$ via the steepest descent optimisation $\alpha_0 = \arg\min_{\alpha \geq 0} f(x^{(0)} - \alpha \nabla f(x^{(0)}))$. If you do it this way, we get the following results:

$$k = 0, x_k = [1, 1, 1, 1]$$
$$k = 1, x_k = [1., 0.34782609, -0.30434783, 1.65217391]$$
$$k = 2, x_k = [1.39494706, 0.08452804, -0.33726008, 1.32305136]$$
$$k = 3, x_k = [1.50092477, 0.01000748, -0.49745676, 1.52586073]$$
$$k = 4, x_k = [1.74348592 - 1.19743517 \times 10^{-3}, -3.75654918 \times 10^{-1} 1.49120345]$$
$$k = 5, x_k = [1.80875926, -1.15166279 \times 10^{-4}, -4.46009569 \times 10^{-1} 1.70043290]$$

and last 5 steps are

$$k = 108, x_k = [3.99730003, -4.49472497 \times 10^{-16}, -4.47928133 \times 10^{-4}, 2.99819589]$$
$$k = 109, x_k = [3.99737876, -4.49472497 \times 10^{-16}, -5.33380212 \times 10^{-4}, 2.99844552]$$
$$k = 110, x_k = [3.99762774, -4.49472497 \times 10^{-16}, -3.93561204 \times 10^{-4}, 2.99841486]$$
$$k = 111, x_k = [3.99769691, -5.29748502 \times 10^{-16}, -4.68641603 \times 10^{-4}, 2.99863420]$$
$$k = 112, x_k = [3.99791567, -5.29748502 \times 10^{-16}, -3.45793018 \times 10^{-4}, 2.99860726]$$




(c) Comment on the differences you observed, why would we prefer steepest descent over gradient descent? Why would you prefer gradient descent over steepest descent? Finally, explain why this is a reasonable condition to terminate use to terminate the algorithm.

**Solution:**

Steepest descent requires fewer steps as expected, but in general, computing the extra gradient at each iteration might be expensive, especially when we do not have an easy $f$ as in this case. The condition is reasonable since we would expect the gradient of the function to be zero at the minimum, so checking for the norm of the gradient to be close to zero is just checking if we have come close to a good value of $x$.

In the next few parts, we will use the gradient methods explored above to solve a real machine learning problem. Consider the data provided in `Q2.csv`. It contains 414 real estate records, each of which contains the following features:

- transactiondate: date of transaction
- age: age of property
- nearestMRT: distance of property to nearest supermarket
- nConvenience: number of convenience stores in nearby locations
- latitude
- longitude

The target variable is the property price. The goal is to learn to predict property prices as a function of a subset of the above features.

(d) We need to preprocess the data. First remove any rows with missing values. Then, delete all features except for age, nearestMRT and nConvenience. Then use the `sklearn minmaxscaler` to normalize the features. Finally, create a training set from the first half of the resulting dataset, and a test set from the remaining half. Your end result should look like:

- first row X_train: $[0.73059361, 0.00951267, 1.]$
- last row X_train: $[0.87899543, 0.09926012, 0.3]$
- first row X_test: $[0.26255708, 0.20677973, 0.1]$
- last row X_test: $[0.14840183, 0.0103754, 0.9]$
- first row Y_train: $37.9$
- last row Y_train: $34.2$
- first row Y_test: $26.2$
- last row Y_test: $63.9$

*What to submit: a copy of your python code in solutions.py*

**Solution:**

```
        df = pd.read_csv("Q2.csv")
        df = df.dropna()
        X = df[["age", "nearestMRT", "nConvenience"]].values
        from sklearn.preprocessing import MinMaxScaler
        scaler = MinMaxScaler()
        X = scaler.fit_transform(X)
        Y = df['price'].values

        X = np.concatenate((np.ones([len(X), 1]), X), axis=1)
        split_point = X.shape[0] // 2
        X_train = X[:split_point]
```

```
        X_test = X[split_point:]
        Y_train = Y[:split_point]
        Y_test = Y[split_point:]

```

(e) Consider the loss function

$$\mathcal{L}(x, y) = \sqrt{\frac{1}{4}(x - y)^2 + 1} - 1,$$

and consider the linear model

$$\hat{y}_i = w_0 + w_1 x_{i1} + w_2 x_{i2} + w_3 x_{i3}, \qquad i = 1, \ldots, n.$$

We can write this more succinctly by letting $w = (w_0, w_1, w_2, w_3)^T$ and $x_i = (1, x_{i1}, x_{i2}, x_{i3})^T$, so that $\hat{y}_i = w^T x_i$. The mean loss achieved by our model ($w$) on a given dataset of $n$ observations is then

$$\mathcal{L}(w) = \frac{1}{n} \sum_{i=1}^{n} \left[ \sqrt{\frac{1}{4}(y_i - w^T x_i)^2 + 1} - 1 \right].$$

We will run gradient descent to compute the optimal weight vector $w$. The iterations will look like

$$w^{(k+1)} = w^{(k)} - \alpha_k \nabla \mathcal{L}(w).$$

Instead of computing the gradient directly though, we will rely on an automatic differentiation library called JAX. Read the first section of the documentation to get an idea of the syntax. Implement gradient descent from scratch and use the JAX library to compute the gradient of the loss function at each step. You will only need the following import statements:

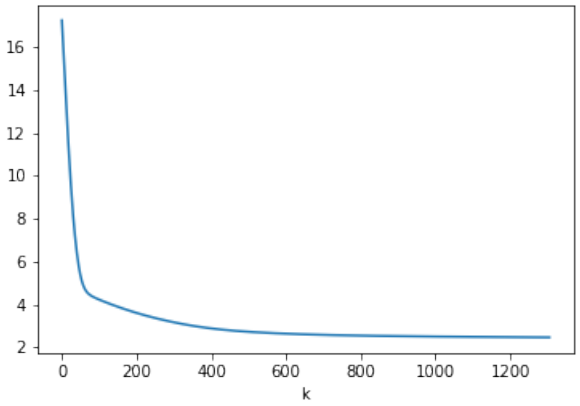
```
    # pip install --upgrade pip
    # pip install jax
    # you may/may not also need to run:  pip install jaxlib
    # pip install --upgrade jax[cpu]

    import jax.numpy as jnp
    from jax import grad

```

Use $w^{(0)} = [1, 1, 1, 1]^T$, and a step size of 1. Terminate your algorithm when the absolute value of the loss from one iteration to the other is less than 0.0001. Report the number of iterations taken, and the final weight vector. Further, report the train and test losses achieved by your final model, and produce a plot of the training loss at each step of the algorithm. *What to submit: a single plot, the final weight vector, the train and test loss of your final model, a screen shot of your code for this section, a copy of your python code in solutions.py*

**Solution:**
This took a total of 1304 steps, the final weight vector was

$$[37.048077, -12.680338, -22.374311, 22.20594]$$

and it achieved a train loss of 2.4738414 and a test loss of 2.6958387. An acceptable range of iterations for full marks for this question is anything between 1280 and 1330. Train loss should be between 2.45 and 2.55, and test loss should be between 2.65 and 2.75.
The plot of losses looks like




The code used here is

```
    def loss(W, X, Y):
        preds = jnp.dot(X, W)
        t1 = jnp.power(Y - preds, 2)
        t2 = jnp.sqrt(0.25 * t1 + 1) - 1
        return jnp.mean(t2)

    Wk = jnp.array([1,1,1,1], dtype=jnp.float32)
    alpha = 0.5
    tol = 0.0001
    losses = []
    losses.append(loss(Wk, X_train, Y_train).item())
    converged = False
    k = 0
    while not converged:
        print('k= ', k, 'Wk= ', Wk)
        W_grad = grad(loss, argnums=0)(Wk, X_train, Y_train)
        Wk = Wk - alpha * W_grad
        losses.append(loss(Wk, X_train, Y_train).item())
        k += 1
        if np.abs(losses[k]-losses[k-1]) < tol:
            converged = True

    print("w: ", Wk)
    print("train loss: ", loss(Wk, X_train, Y_train))
    print("test loss: ", loss(Wk, X_test, Y_test))

    plt.plot(np.arange(k+1), losses)
    plt.xlabel("k")
    plt.savefig("figures/GDSteps.png")
    plt.show()
```

(f) Finally, re-do the previous section but with steepest descent instead. In order to compute $\alpha_k$ at each step, you can either use JAX or it might be easier to use the minimize function in `scipy` (See lab3). Run the algorithm with the same $w^{(0)}$ as above, and take $\alpha_0 = 1$ as your initial guess when numerically solving for $\alpha_k$ (for each $k$). Terminate the algorithm when the loss value falls bellow $2.5$. Report the number of iterations it took, as well as the final weight vector, and the train and test losses achieved. Generate a plot of the losses as before and include it. *What to submit: a single plot, the final weight vector, the train and test accuracy of your final model, a screen shot of your code for this section, a copy of your python code in solutions.py*

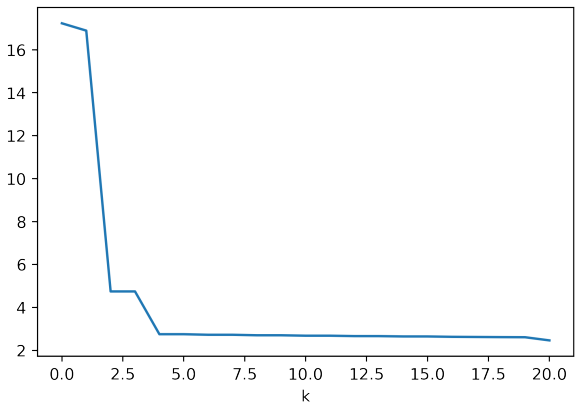
**Solution:**

This took a total of 19 steps, the final weight vector was

$$[39.428837, -16.099659, -23.186417, 20.156746]$$

and it achieved a train loss of 2.465345 and a test loss of 2.6919682. Acceptable number of iterations for full marks should be between 10 and 30 iterations. Train los sshould be between 2.45 and 2.5, test loss should be between 2.65 and 2.75.

The plot of losses (which may vary depending on optimisation scheme used for $\alpha$) looks like




The code used here is

```
    def loss_n(W, X, Y):
        # this version is the same as the previous one, but scipy.optimize does not
    seem
        # to be compatible with jnp arrays, so this is the 'numpy' version
        preds = np.dot(X, W)
        t1 = np.power(Y - preds, 2)
        t2 = np.sqrt(0.25 * t1 + 1) - 1
        return np.mean(t2)

    Wk = jnp.array([1,1,1,1], dtype=jnp.float32)
```

```
    alpha_k = 1
    alphas = [alpha_k]
    losses = []
    losses.append(loss(Wk, X_train, Y_train).item())
    converged = False
    k = 0
    while not converged:
        print('k= ', k, 'Wk= ', Wk)
        W_grad = grad(loss, argnums=0)(Wk, X_train, Y_train)
        Wk = Wk - alpha_k * W_grad
        losses.append(loss(Wk, X_train, Y_train).item())
        k += 1
        g = lambda a: loss_n(Wk.to_py()-a*W_grad.to_py(), X_train, Y_train)
        alpha_k = minimize(g, x0=[1], method='BFGS', tol=1e-6).x[0]
        alphas.append(alpha_k)
        if losses[k] < 2.5:
            converged=True


    print("w: ", Wk)
    print("train loss: ", loss(Wk, X_train, Y_train))
    print("test loss: ", loss(Wk, X_test, Y_test))

    plt.plot(np.arange(k+1), losses)
    plt.xlabel("k")
    plt.savefig("figures/SteepSteps.png", dpi=400)
    plt.show()
```

(g) In this question we have explored the gradient descent and steepest descent variants of the gradient method. Many other gradient based algorithms exist in the literature. Choose one and describe it. *What to submit: some commentary, any plots or code you used in this section (you don't need to write any code, or supply plots, but you may. Also be sure to cite any sources used here.)*