NAME OF CANDIDATE: ....

STUDENT ID: ....

SIGNATURE: ....

THE UNIVERSITY OF NEW SOUTH WALES

Term 2, 2021

COMP9417 Machine Learning and Data Mining – Final Examination

1. TIME ALLOWED — 24 HOURS
2. THIS EXAMINATION PAPER HAS 15 PAGES
3. TOTAL NUMBER OF QUESTIONS — 4
4. ANSWER **ALL 4 QUESTIONS**
5. TOTAL MARKS AVAILABLE — 100
6. OPEN BOOK EXAM - LECTURE NOTES, TUTORIALS, AND ONLINE RESOURCES ARE PERMITTED. PLEASE REFER TO EXAM INSTRUCTIONS ON MOODLE COURSE PAGE FOR SUBMISSION AND OTHER GUIDANCE.
7. DISCUSSION WITH OTHER STUDENTS IS STRICTLY PROHIBITED. EXAM SUBMISSIONS WILL BE CHECKED FOR PLAGIARISM. CHEATING WILL RESULT IN A FAILING GRADE FOR THE COURSE AND POTENTIAL FURTHER DISCIPLINARY ACTION.

Question 1

Please submit Question1.pdf on Moodle using the Final Exam - Question 1 object. You must submit a single PDF. You may submit multiple .py files if you wish. The parts are worth 4 + 6 + 6 + 6 + 3 = 25.

In this problem we will consider a slightly different dataset than the one used in Homework 2 question 1, which is provided in `Q1.csv` with binary response variable Y and 30 continuous features $X_1, \dots, X_{30}$. Recall the set-up for regularized logistic regression in that question. In this question we will use the ℓ_1 penalty, i.e., we wish to solve

$$(\hat{\beta}_0, \hat{\beta}) = \arg\min_{\beta_0, \beta} \{CL(\beta_0, \beta) + \|\beta\|_1\} \tag{1}$$

where $L(\beta_0, \beta)$ is the log-loss and where the penalty is usually not applied to the bias term β_0, and C is a hyper-parameter controlling the trade-off between fit and regularization. **Assume for the remainder of this question that $C = 1000$, and work with the entire data set provided. We will use the liblinear solver and l1 penalty throughout. Do not use any existing implementations of the Bootstrap and/or Jackknife algorithms; doing so will result in a grade of zero for this question**. In Homework 2 we built confidence intervals using the nonparametric bootstrap. In this question we will explore two different approaches to building the CIs.

(a) First, run an analysis identical to the one done in Homework 2 part (d) using the nonparametric bootstrap, and using the new dataset and updated parameter values: $B = 500; C = 1000$; and set a random seed of 12. Feel free to modify your own code from Homework 2, or modify the code in the sample solution. Be sure to use the correct C value in any sub-models fit in the bootstrap process. Generate a plot where the x-axis represents the different parameters $\beta_1, \dots, \beta_p$, and plot vertical bars to represent the CIs. For those intervals that contain 0, draw the bar in red, otherwise draw it in blue. Also, indicate on each bar the nonparametric bootstrap mean. Along with your plot, answer the following:

   (i) What effect does taking $C = 1000$ have?

   (ii) Do you think that taking $C = 1000$ will result in more reliable estimates of the confidence intervals relative to taking C to be small (e.g. $C = 0.1$)? Why?

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

(b) Here we will explore a variant called the parametric bootstrap, which is an approach that takes advantage of the specified (Logistic) model. Recall from lectures that in Logistic regression, we are modelling

$$\mathbb{P}(Y = 1 | X = x) = \frac{e^{\beta_0 + \langle \beta, x \rangle}}{1 + e^{\beta_0 + \langle \beta, x \rangle}}.$$

When we fit the model, we compute estimates $\hat{\beta}_0, \hat{\beta}$ for β_0, β that we can then plug into the above equation. Assuming we have trained a logistic model on our data (and so have obtained estimates $\hat{\beta}_0, \hat{\beta}$) the idea now is to construct bootstrap samples and compute 90% confidence intervals in the following way:

   1. Generate B bootstrap samples (datasets) that contain n data points each. Each of these bootstrap samples, for $b = 1, \dots, B$, will contain data points $(X_i^{(b)}, Y_i^{(b)})$, $i = 1, \dots, n$, which are chosen in the following way:

$$X_i^{(b)} = X_i, \qquad Y_i^{(b)} \sim \text{Bernoulli}(\hat{p}_i),$$

where

$$\hat{p}_i = \frac{e^{\hat{\beta}_0 + \langle \hat{\beta}, X_i \rangle}}{1 + e^{\hat{\beta}_0 + \langle \hat{\beta}, X_i \rangle}}.$$

In words, the i-th point in the bootstrap sample has the same feature value (X_i) as in the original dataset, but its response value is sampled (this is denoted by the $\sim$ notation) from a Bernoulli distribution with parameter $\hat{p}_i$, where $\hat{p}_i$ is built using our fitted model parameters $\hat{\beta}_0, \hat{\beta}$. You can sample from a Bernoulli distribution in NumPy using `numpy.random.binomial`.

2. Now that you have B Bootstrap samples, generate 90% confidence intervals exactly as in the nonparametric Bootstrap case.

Take $B = 500$ and set a random seed of 20. Generate a plot where the x-axis represents the different parameters $\beta_1, \dots, \beta_p$, and plot vertical bars to show your CIs. For those intervals that contain 0, draw the bar in red, otherwise draw it in blue. Also indicate on each bar the parametric bootstrap mean. Remember to use $C = 1000$ in any sub-models. Comment briefly on the results.

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

(c) In this part, we will explore a third approach to constructing confidence intervals called the Jackknife. The Jackknife is an extremely flexible tool for quantifying the mean and variance of any estimator and is used as an alternative to the Bootstrap. We now describe the algorithm for generating 90% confidence intervals using the Jackknife: Note that whenever we run logistic regression, we compute an estimate $\hat{\beta}_j$ of the j-th coefficient β_j. We can view $\hat{\beta}_j$ as a function of the data $D_n = \{(X_1, Y_1), \dots, (X_n, Y_n)\}$, i.e., we can write $\hat{\beta}_j = \hat{\beta}_j(D_n)$. Let $D_n^{(-i)}$ denote the dataset D_n with the i-th point removed. For example,

$$D_n^{(-2)} = \{(X_1, Y_1), (X_3, Y_3), \dots, (X_n, Y_n)\}.$$

Now, denote the estimated coefficient vector computed on this reduced dataset by $\hat{\beta}_j^{(-i)} = \hat{\beta}_j(D_n^{(-i)})$. With this new notation in hand, define the i-th pseudovalue,

$$\gamma_i^j = n\hat{\beta}_j(D_n) - (n-1)\hat{\beta}_j(D_n^{(-i)}), \qquad i = 1, \dots, n.$$

We define the mean and variance of the pseudovalues in the standard way:

$$\bar{\gamma}_j = \frac{1}{n}\sum_{i=1}^{n} \gamma_i^j, \qquad S_j^2 = \frac{1}{n-1}\sum_{i=1}^{n} (\gamma_i^j - \bar{\gamma}_j)^2.$$

$\bar{\gamma}_j$ is called the jackknife estimate (of β_j) and S_j^2 is the jackknife variance. We may then compute a 90% CI for β_j as follows:

$$\left(\bar{\gamma}_j - 1.645\sqrt{\frac{S_j^2}{n}}, \bar{\gamma}_j + 1.645\sqrt{\frac{S_j^2}{n}} \right)$$

Run the Jackknife algorithm for our problem and generate an identical plot to that of part (b). For those intervals that contain 0, draw the bar in red, and otherwise draw it in blue. Also indicate on each bar the jackknife mean ($\bar{\gamma}_j$). Remember to use $C = 1000$ in any sub-models. *What to submit: a single plot, a screen shot of your code for this section, a copy of your Python code in solutions.py*

(d) The coefficient vector β that generated the dataset used in parts (a), (b), (c) was generated using the following code:

```
np.random.seed(125)
p = 20
k = 8
betas = np.random.random(p) + 1
betas[np.random.choice(np.arange(p), p-k, replace=False)] = 0.0

```

Use this to assess the quality of the CIs constructed using the three methods in parts (a), (b) and (c): nonparametric Bootstrap (NP), parametric Bootstrap (P), Jackknife (JK) and report the number of false positives and false negatives found by each method. A false positive here would be a CI that does not contain zero when the true parameter is actually zero. A false negative would be a CI that does contain zero when the true parameter is non-zero. Summarise your results in a table of the form:

Algorithm	# False Positives	# False Negatives
NP	-	-
P	-	-
JK	-	-

What to submit: a table of results in the specified format, a screen shot of your code for this section, a copy of your Python code in solutions.py

(e) This question is unrelated to the previous parts (though it requires the usage of the definitions in part (c)). Consider the dataset $Z_1, \ldots, Z_n$. Compute the Jackknife mean and variance of the estimator $Z = \frac{1}{n}\sum_{i=1}^{n} Z_i$. Show your working for full marks. Hint: First compute the pseudo-values in terms of the Z_i's. *What to submit: your working out, either typed or handwritten.*

Question 2

Please submit Question2.pdf on Moodle using the Final Exam - Question 2 object. You must submit a single PDF. You may submit multiple .py files if you wish. The parts are worth 4 + 4 + 9 + 8 = 25.

Note that parts 2 (a) and 2 (b) and 2 (c) and 2(d) of this question are unrelated.

(a) Consider the following dataset:

$$D := \{((1,1,1),0),((1,0,0),0),((1,1,0),1),((0,0,1),1)\},$$

i.e., each data point is composed of a vector of 3 binary features, and a binary response. We will train a decision tree of depth 2 on this dataset using the standard ID3 algorithm covered in lectures and tutorials (please use the natural log (base e) instead of base 2 in all calculations for this question).

(i) Draw the tree computed and write down the training error it achieves. Be sure to show all working for full marks. *What to submit: your working out, either typed or handwritten. Please include a drawing/plot of your tree.*

(ii) Can you construct a depth 2 decision tree that achieves a lower training error than the one found via ID3? What does this tell you about ID3? Explain. *What to submit: your working out, either typed or handwritten. Please include a drawing/plot of your tree.*

(b) Assume that you have access to a function `train_perceptron` that trains a standard perceptron (with learning rate $\eta = 1$, and $w^{(0)} = 0_p$ where 0_p is the vector of zeroes in p dimensions. Here, p denotes the number of features plus 1 for the bias term). Design an algorithm that establishes whether a given dataset is linearly separable or not, and which makes a single call to `train_perceptron`. Describe your algorithm (a few dot points should suffice), and write a Python function implementing the algorithm. When running the perceptron function, terminate it after at most 10000 iterations. Test your function on the following datasets and report whether or not they are linearly separable.

(i) $\{010, 011, 100, 111\}$

(ii) {011,100,110,111}

(iii) $\{0100, 0101, 0110, 1000, 1100, 1101, 1110, 1111\}$

(iv) $\{1000000, 1000001, 1000101\}$.

Present a summary of your results using the following table:

Dataset	Linearly Separable (Yes/No)
(i)	-
(ii)	-
(iii)	-
(iv)	-

The datasets represent the points that are labelled $+1$. All other remaining binary vectors of the same length are labelled -1. For example, for (i), there are $2^3 = 8$ possible binary vectors of length 3, the four strings $\{010, 011, 100, 111\}$ belong to the positive class, the remaining 4 strings $\{000, 001, 101, 110\}$ belong to the negative class. You may find the following StackExchange post helpful in coding up the datasets; it describes a simple approach to generating all length k binary vectors in Python using `itertools.product`.

Note: Using an existing algorithm to determine linear separability here will result in a grade of zero for this part, but you are free to use existing implementations of the perceptron algorithm. *What to submit: a description of your algorithm in words, your filled out table, a screen shot of your code used for this section, a copy of your code in solutions.py*

(c) Gradient-based optimization algorithms are a driving force in modern machine learning, and we have already seen multiple examples in this course, namely: gradient descent, stochastic gradient descent (and Backpropagation) and steepest decent. In this question we will consider an elegant approach to generating a family of gradient-based iterative algorithms. Assume that you have some (differentiable) function $f : \mathbb{R}^p \to \mathbb{R}$ that you wish to minimize. The notation here just means that f is a function that takes inputs (vectors) from $\mathbb{R}^p$ and returns elements in $\mathbb{R}$ (real numbers). Let ψ be some differentiable function and consider the pseudo-distance:

$$H_\psi(x, z) = \psi(x) - \psi(z) - \langle \nabla\psi(z), x - z \rangle, \qquad x, z \in \mathbb{R}^p.$$

We will minimize f iteratively (like in gradient descent) using the following update rule at each step:

$$x^{(t+1)} = \arg\min_x \{\alpha \left\langle \nabla f(x^{(t)}), x \right\rangle + H_\psi(x, x^{(t)})\}, \tag{2}$$

where $\alpha > 0$ is a given learning rate (step size).

(i) Show that for the choice $\psi : \mathbb{R}^p \to \mathbb{R}$ defined as $\psi(x) = \frac{1}{2}\|x\|_2^2$, that solving the update step (2) recovers the gradient descent algorithm exactly. To do this, first solve for H_ψ, then plug your result into (2), differentiate with respect to x, set equal to zero and solve for x. Show all working.

(ii) Let Q be a $p \times p$ invertible matrix, and consider $\psi : \mathbb{R}^p \to \mathbb{R}$ defined as $\psi(x) = x^T Q x$. Redo the previous question with this choice of ψ. Show all working.

(iii) Let $\mathcal{S}^{p-1}$ denote the space of p dimensional vectors such that the coordinates of the vector are non-negative and sum to 1. For example, if $p = 3$, then $x = (1/3, 2/3, 0)$ belongs to $\mathcal{S}^2$, but $z = (1/3, -1/3, 1)$ doesn't belong to $\mathcal{S}^2$. Consider $\psi : \mathcal{S}^{p-1} \to \mathbb{R}$ defined by $\psi(x) = \sum_{i=1}^p x_i \log x_i$, where *log* is the natural logarithm (base e). What updates does this choice of ψ generate? Hint: it may be easier to work on the coordinate level rather than with vectors for this question.

Please show all working for each of part, and summarise your results using the following table:

	ψ	$H_\psi(x, z)$	$x^{(t+1)}$
(i)	$\frac{1}{2}\|x\|_2^2$	-	$x^{(t)} - \alpha\nabla f(x^{(t)})$
(ii)	$\frac{1}{2}x^T Q x$	-	-
(iii)	$\sum_{i=1}^p x_i \log x_i$	-	-

What to submit: your working out, either typed or handwritten, and your filled in table of results. You cannot use matlab/sympy etc to solve any calculations.

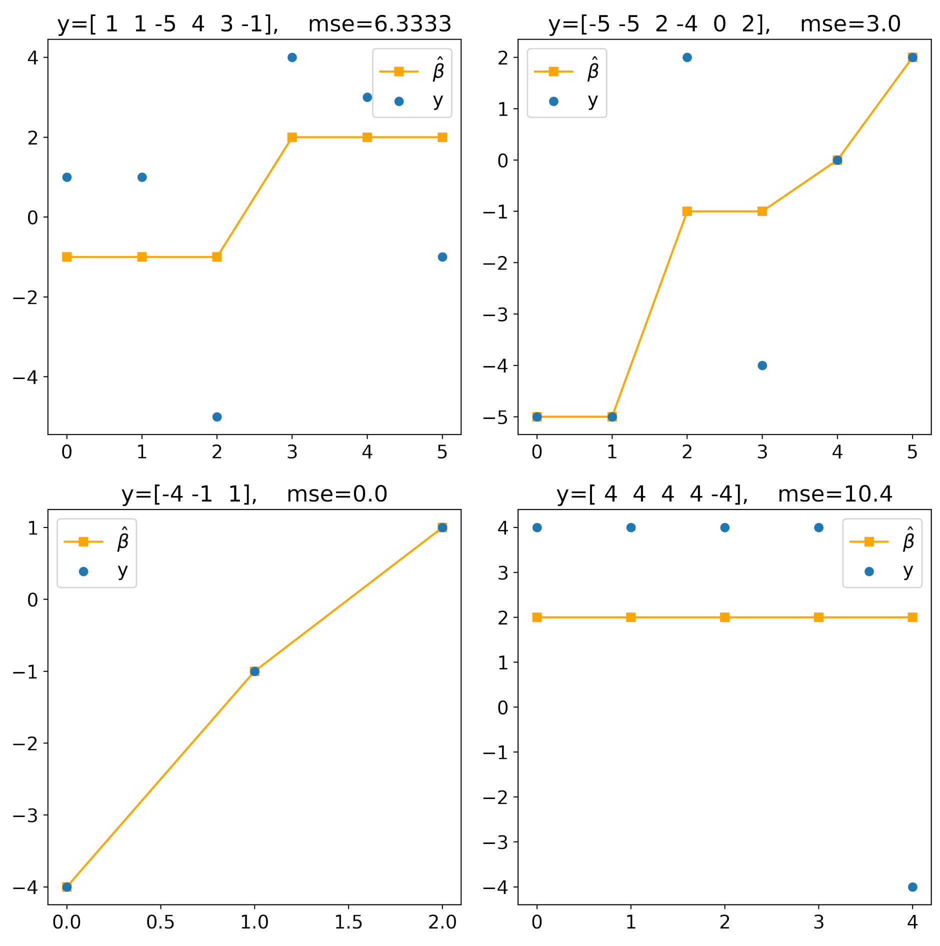
(d) In this part, we will explore a type of regression algorithm that imposes a restriction on the shape of the fitted model, rather than requiring the model to belong to some class of functions, such as the class of linear functions when doing linear regression, for example. In particular, we will look at fitting a non-decreasing sequence model. A sequence of numbers is non-decreasing if no element of the sequence is smaller than the preceding elements. For example, the sequences $(1, 1, 1, 1, 2)$ and $(-4, 20, 20, 120)$ are non-decreasing, while the sequence $(2, 1, 2, 2, 3)$ is not non-decreasing. Given a data sequence $(y_1, y_2, \dots, y_n)$, our goal is to find a non-decreasing sequence $(\beta_1, \dots, \beta_n)$ that best fits the data. Obviously, if $(y_1, \dots, y_n)$ is itself a non-decreasing sequence, the best possible fit is $(\hat\beta_1, \dots, \hat\beta_n) = (y_1, \dots, y_n)$. To put this into our usual loss minimization framework, we seek to find:

$$\hat{\beta} = (\hat{\beta}_1, \dots, \hat{\beta}_n) = \arg\min_{\beta \in \mathbb{R}^n} \sum_{i=1}^{n} (y_i - \beta_i)^2 \qquad \text{subject to} \qquad \beta_1 \le \beta_2 \le \cdots \le \beta_n.$$

A very simple algorithm exists that solves this problem (and gives a unique solution). The following is an outline:

```
input: $(x_1 = 1, y_1), \dots, (x_n = n, y_n)$
initialise: $(C_1^{(0)}, \dots, C_n^{(0)}) = (y_1, y_2, \dots, y_n)$
for $t = 1, \cdots$ :
    if there is an index $i$ such that $C_i^{(t)} < C_{i+1}^{(t)}$ :
        $C_i^{(t+1)} = C_{i+1}^{(t+1)} = \frac{C_i^{(t)} + C_{i+1}^{(t)}}{2}; \quad t = t + 1$
    else:
        output $(C_1^{(t)}, \dots, C_n^{(t)}), t$
```

In words, each y_i starts out in its own cluster. At each iteration of the algorithm, we check to see if there are any two clusters that violate the non-decreasing requirement, and if there is, we set their cluster values to be the average of the original cluster values. Effectively, we are merging the two clusters into one. The algorithm terminates when there are no violations of the non-decreasing requirement. The following plot depicts the algorithm running on four different datasets; the titles contain the true values of y as well as the MSE achieved by each of the fitted models. Note that the x values of the points just range from 0 to the length of y and therefore they do not play any role in the fitting process, they are just to be used for plotting.



In this question, you will implement a version of this algorithm on the six datasets provided in `Q2_mono_dict.npy` A straightforward implementation of the algorithm should suffice for datasets (i) and (ii) and (iii), but a more efficient implementation is needed for datasets (iv)-(vi) to run in a reasonable amount of time. Please plot your fits and report the MSE value just as in the provided example. To assist you with this question, the code snippet below loads in the data and generates the plots, so you can focus on writing the algorithm. Note: you are not permitted to use any other import statements for this part. As you build your algorithm, use the provided examples as test cases. *What to submit: the generated plot (3x2 grid), a screen shot of your code used for this section, a copy of your code in solutions.py*

```
import matplotlib
import matplotlib.pyplot as plt
import numpy as np
import panadas as pd # not needed
matplotlib.rc('font', **{'size'    : 14})    # make plot text more visible
# do not import anything else

def mse(y,yhat):
    return np.mean(np.square(y-yhat))
```

```

def your_function(y):
    ########## your code here

# load in data
y_dict = np.load('Q2_mono_dict.npy', allow_pickle=True).item()

# create plot
fig, ax = plt.subplots(3,2, figsize=(14,14))
plot_titles = ['(i)', '(ii)', '(iii)', '(iv)', '(v)', '(vi)']

for i, ax in enumerate(ax.flat):

    y = y_dict[plot_titles[i]]
    x =  np.arange(y.shape[0])
    betahat = your_function(y)          ### update this line

    # plot data and fit
    ax.scatter(x, y, marker='o', label="y")
    ax.plot(x, betahat,
            color='orange', linestyle='-', marker='s',
            label=r'$\hat{\beta}$') # if you have latex issues, replace with label='
hatbeta'

    # set title and put legend in a good spot
    mse_bh = np.round(mse(y,betahat), 4)
    ax.set_title(f'part={plot_titles[i]}, mse={mse_bh}')
    ax.legend(loc='best')

plt.tight_layout()
plt.savefig("pickAName.png", dpi=400)          ### update this line
plt.show()

```

Question 3

Please submit Question3.pdf on Moodle using the Final Exam - Question 3 object. You must submit a single PDF. You may submit multiple .py files if you wish. The parts are worth 3 + 3 + 6 + 8 + 5 = 25.

In this question we will take a deeper look at the perceptron algorithm. We will work with the following data files `Q3X.csv` and `Q3y.csv`, which are the X, y for our problem, respectively. Note that the structure of these arrays is identical to the data we worked with in the Neural Learning lab. You might find the perceptron and plotting functions in that lab useful here, though the perceptron implementation will need to be tweaked slightly to match the respective pseudocode outlined in each part below. In this question (all parts), you are only permitted to use the following import statements:

```python
import numpy as np
import pandas as pd # not really needed, only for preference
import matplotlib.pyplot as plt
```

(a) Write a function that implements the perceptron algorithm and run it on X, y. Your implementation should be based on the following pseudo-code:

```
input: $(x_1, y_1), \dots, (x_n, y_n)$
initialise: $w^{(0)} = (0, 0, \dots, 0) \in \mathbb{R}^p$
for $t = 1, \dots, \texttt{max\_iter}$ :
    if there is an index $i$ such that $y_i \langle w^{(t)}, x_i \rangle \leq 0$ :
        $w^{(t+1)} = w^{(t)} + y_i x_i; \quad t = t + 1$
    else:
        output $w^{(t)}, t$
```

Note that at each iteration of your model, you need to identify all the points that are misclassified by your current weight vector, and then sample **one of these points** randomly and use it to perform your update. For consistency in this process, please set the random seed to 1 inside of your function, i.e.,

```python
def perceptron(X, y, max_iter=100):
    np.random.seed(1)
    # your code here

    return w, nmb_iter
```

The `max_iter` parameter here is used to control the maximum number of iterations your algorithm is allowed to make before terminating and should be set to 100. Provide a plot of your final model superimposed on a scatter of the data, and print out the final model and number of iterations taken for convergence in the title. For example, you can use the following code for plotting:

```python
w = ## your trained perceptron
fig, ax = plt.subplots()
plot_perceptron(ax, X, y, w)        # from neural learning lab
ax.set_title(f"w={w},    iterations={nmb_iter}")
plt.savefig("name.png", dpi=300)       # if you want to save your plot as a png
plt.show()
```

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

(b) In this section, we will implement and run the dual perceptron algorithm on the same X, y. Recall the dual perceptron pseudo-code is:

```
input: $(x_1, y_1), \dots, (x_n, y_n)$
initialise: $\alpha^{(0)} = (0, 0, \dots, 0) \in \mathbb{R}^n$
for $t = 1, \dots, \texttt{max\_iter}$ :
    if there is an index $i$ such that $y_i \sum_{j=1}^{n} y_j \alpha_j \langle x_j, x_i \rangle \leq 0$ :
        $\alpha_i^{(t+1)} = \alpha_i^{(t)} + 1; \quad t = t + 1$
    else:
        output $\alpha^{(t)}, t$
```

In your implementation, use the same method as described in part (a) to choose the point to update on, using the same random seed inside your function. Provide a plot of your final perceptron as in the previous part (using the same title format), and further, provide a plot with x-axis representing each of the $i = 1, \dots, n$ points, and y-axis representing the value α_i. Briefly comment on your results relative to the previous part. *What to submit: two plots, some commentary, a screen shot of your code used for this section, a copy of your code in solutions.py*

(c) We now consider a slight variant of the perceptron algorithm outlined in part (a), known as the rPerceptron. We introduce the following indicator variable for $i = 1, \dots, n$:

$$I_i = \begin{cases} 1 & \text{if } (x_i, y_i) \text{ was already used in a previous update} \\ 0 & \text{otherwise.} \end{cases}$$

Then we have the following pseudo-code:

```
input: $(x_1, y_1), \dots, (x_n, y_n), r > 0$
initialise: $w^{(0)} = (0, 0, \dots, 0) \in \mathbb{R}^p, \quad I = (0, 0, \dots, 0) \in \mathbb{R}^n$
for $t = 1, \dots, \texttt{max\_iter}$ :
    if there is an index $i$ such that $y_i \langle w^{(t)}, x_i \rangle + I_i r \leq 0$ :
        $w^{(t+1)} = w^{(t)} + y_i x_i; \quad t = t + 1; \quad I_i = 1$
    else:
        output $w^{(t)}, t$
```

Implement the rPerceptron and run it on X, y taking $r = 2$. Use the same randomization step as in the previous parts to pick the point to update on, using a random seed of 1. Provide a plot of your final results as in part (a), and print out the final weight vectors and number of iterations taken in the title of your plot. *What to submit: a single plot, a screen shot of your code used for this section, a copy of your code in solutions.py*

(d) Derive a dual version of the rPerceptron algorithm and describe it using pseudo-code (use the template pseudocode from the previous parts to get an idea of what is expected here). Implement your algorithm in code (using the same randomization steps as above) and run it on X, y with $r = 2$. Produce the same two plots as requested in part (b). *What to submit: a pseudocode description of your algorithm, two plots, a screen shot of your code used for this section, a copy of your code in solutions.py*

(e) What role does the additive term introduced in the rPerceptron play? In what situations (different types of datasets) could this algorithm give you an advantage over the standard perceptron? What is a disadvantage of the rPerceptron relative to the standard perceptron? Refer to your results in the previous parts to support your arguments; please be specific. *What to submit: some commentary.*

Question 4

Please submit Question4.pdf on Moodle using the Final Exam - Question 4 object. You must submit a single PDF. You may submit multiple .py files if you wish. The parts are worth 2+3+4+4+12=25.

In this question, you are only permitted to use the following import statements:

```python
import numpy as np
import pandas as pd # not really needed, only for preference
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression
```

Recall the set up of linear regression: we are given data $\{(x_i, y_i)\}_{i=1}^n$ where $x_i \in \mathbb{R}^p$ and $y_i \in \mathbb{R}$. We believe that the data was generated from some true (linear) model, with additive noise, i.e., each observation is generated in the following manner:

$$y_i = x_i^T \beta^* + \sigma \epsilon_i,$$

where ϵ_i is a random noise term (e.g., a sample from a standard normal random variable), and σ controls the magnitude of this noise. Here we use β^* to denote the *true* parameter vector that generated the data, and the goal is to recover β^*. The standard approach (of which we have seen plenty of examples in the course) is to use the data to do this recovery, for example by solving the squared error minimization problem:

$$\hat{\beta} = \arg\min_{b \in \mathbb{R}^p} \sum_{i=1}^n (x_i^T b - y_i)^2. \tag{3}$$

In this problem we will consider the above scenario with a slight twist. You are provided train and test datasets in the files: `Q4_train.csv` and `Q4_test.csv`. You are told that the data were generated from M different linear regression models, but you are not told what M is, nor which data point was generated by which of the M models. More rigorously, there are M true parameter vectors $\beta_1^* \in \mathbb{R}^p, \beta_2^* \in \mathbb{R}^p, \dots, \beta_M^* \in \mathbb{R}^p$, and each parameter vector is responsible for generating an unknown subset of the data. If the i-th data point was generated by model m, then we believe that

$$y_i = x_i^T \beta_m^* + \sigma \epsilon_i,$$

where σ is the same for all data points, regardless of the true parameter. Given a value of M, your goal is to first design and then implement in code an algorithm to:

- partition the data into M non-overlapping subsets, denoted $\mathcal{D}_1, \dots, \mathcal{D}_M$ so that $(x_i, y_i) \in \mathcal{D}_k$ if and only if (x_i, y_i) was generated by the k-th model.
- learn the M parameter vectors: i.e., compute estimates $\hat{\beta}_1, \dots, \hat{\beta}_M$.

(a) Write an objective/loss function that could be used to solve this problem. The loss function should be a function of both the partitions found and the learned parameter vectors, i.e., $L(\mathcal{D}_1, \dots, \mathcal{D}_M, \hat{\beta}_1, \dots, \hat{\beta}_M)$. It should be a sum (not an average) of n terms (where n is the number of total samples), and each sample should only belong to one of these n terms. You must use the squared-loss for the individual terms: $(x_i^T \hat{\beta}_m - y_i)^2$. Hint: in the special case $M = 1$, your solution would just be the same loss used in (3).

(b) Design an algorithm to learn a model that minimizes the loss found in (a). Describe your algorithm using pseudocode. Hint 1: If $\hat{\beta}_1, \dots, \hat{\beta}_M$ were held fixed, how would you assign the n points to M different partitions? Next, if the partitions $\mathcal{D}_1, \dots, \mathcal{D}_M$ were held fixed, how would you learn the parameter vectors? Hint 2: Iterate between the two ideas in Hint 1.

(c) Assume you are given `X, y, Z, models`, where `X`, `y` is your data, `Z` is a vector in $\mathbb{R}^n$, with i-th element `Z[i]`$= m$ if $(x_i, y_i) \in \mathcal{D}_m$, and `models` is a list of M scikit-learn `LinearRegression` models, one for each of the partitions. Implement a loss function based on your answer to (a) in Python. A template is provided below. Note: if you prefer not to structure your code this way and wish to implement your algorithm in (b) directly, you can safely skip (c) and (d) and complete (e). If your answer to (e) is correct you will receive full marks for (c) and (d) and (e).

```python
def total_loss(X, y, Z, models):
    '''
    computes total loss achieved on X, y based on linear regressions stored in models
, and partitioning Z

    :param X: design matrix, n x p (np.array shape (n,p))
    :param y: response vector, n x 1 (np.array shape (n,1) or (n,))
    :param Z: assignment vector, n x 1 (assigns each sample to a partition)
    :param models: a list of M sklearn LinearRegression models, one for each of the
partitions

    :returns: the loss of your complete model as computed in (a)
    '''

    loss = 0
    M = len(models)

    # Your code here

    return loss


#Example, if M=1, we would just fit a single linear model
mod = LinearRegression().fit(Xtrain, ytrain)
Z = np.zeros(shape=Xtrain.shape[0])     # all points would belong to a single
partition.
print(total_loss(Xtrain, ytrain, Z, [mod]))  # outputs 298.328178158043
```

(d) Now write a function that takes as inputs `X, y, models` (**but not `Z`**) and partitions the data into M subsets (i.e., implement your solution in (b) on how to choose the partitions in the situation that $\hat{\beta}_1, \dots, \hat{\beta}_M$ are fixed). The following is a template

```python
def find_partitions(X, y, models):
    '''
    given M models, assigns points in X to one of the M partitions

    :param X: design matrix, n x p (np.array shape (n,p))
    :param y: response vector, n x 1 (np.array shape (n,1) or (n,))
    :param models: a list of M sklearn LinearRegression models for each
    of the partitions

    :returns: Z, a np.array of shape (n,) assigning each of the points in X to one of
 M partitions
    '''
    M = len(models)
    # your code here
    return Z


#Example, if M=1, using same 'mod' from example in (c)
Z = find_partitions(Xtest, ytest, models)
```

```
    print(total_loss(Xtest, ytest, Z, [mod])) # outputs 54.3679338727877

```

(e) Write code to implement your algorithm from part (b). The functions defined in (c) and (d) should be helpful here. Run the algorithm for different choices of $M = \{1, 2, 3, \dots, 30\}$. Generate a table with your train and test losses for $M = 5, 10, 15, 20, 25, 30$ (use the template table below). Further, generate a plot of your train/test loss for all M. Based on your results, what value of M do you think is most reasonable? Technical note: In order to ensure reproducibility of your implementation, when you initialize your assignment vector `Z`, do so in the following way: if $M = 3$ and $n = 10$, then Z is initialized as `Z = np.array([0,1,2,0,1,2,0,1,2,0])`, similarly if $M = 4$ and $n = 14$ then initialize `Z = np.array([0,1,2,3,0,1,2,3,0,1,2,3,0,1])`. You might find the function `np.tile` useful here.

M	train loss	test loss
5	-	-
10	-	-
15	-	-
20	-	-
25	-	-
30	-	-