COMP9417 - Machine Learning
Homework [illegible]tion models - Regularized Logistic [illegible] and the Perceptron

Introduction In this homework we first look at a regularized version of logistic regression. You will implement the algorithm from scratch and compare it to the existing `sklearn` implementation. Special care will be taken to ensure that our implementation is numerically stable. We then move on to consider the important issue of hyperparameter tuning. In the second question we shift our focus to the perceptron and dual perceptron algorithms. We will implement this algorithm from scratch and compare it to a variant known as the rPerceptron.

Points Allocation There are a total of 28 marks.

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

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. [illegible] new page if necessary.
- You **cannot** sub[illegible] this will receive a mark of zero. This does not stop you from developing you[illegible]d then copying it into a .py file though, or using a tool such as **nbconvert** or si[illegible]
- We will set up a [illegible]tions about this homework. Please read the existing questions before posting r[illegible]o some basic research online before posting questions. Please only post clarifi[illegible] questions deemed to be *fishing* for answers will be ignored and/or deleted.
- Please check Moodle announcements 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(s) 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, Monday **March 28th**, 2022 by **5pm**. Please note that the forum will not be actively monitored on weekends.
- Late submissions will incur a penalty of 5% per day **from the maximum achievable grade**. For example, if you achieve a grade of 80/100 but you submitted 3 days late, then your final grade will be $80 - 3 \times 5 = 65$. Submissions that are more than 5 days late will receive a mark of zero.
- Submission must be done through Moodle, no exceptions.

Question 1. Regularized Logistic Regression

Note: throughout this question do not use any existing implementations of any of the algorithms discussed unless explicitly asked to in the question. Using existing implementations can result in a grade of zero for the entire question. In this question we will work with the Regularized Logistic Regression model for binary classification (i.e. we are trying to predict a binary target). Instead of using mean squared err[illegible] problems with a continuous target (such as linear regression), we instead minim[illegible] ferred to as the cross entropy loss. Recall that for a parameter vector $\beta = (\beta_1, \dots$ [illegible] $x_i \in \mathbb{R}^p$ for $i = 1, \dots, n$, the log-loss is

$$L[illegible] \frac{1}{[illegible] + \beta^T x_i)}\right) + (1 - y_i) \ln\left(\frac{1}{1 - \sigma(\beta_0 + \beta^T x_i)}\right),$$

where $\sigma(z) = (1$ [illegible] sigmoid. In practice, we will usually add a penalty term, and solve the optimization:

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

where the penalty is usually not applied to the bias term β_0, and C is a hyper-parameter. For example, in the ℓ_2 regularization case, we take penalty$(\beta) = \frac{1}{2}\|\beta\|_2^2$ (a Ridge version of logistic regression).

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

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

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 Ridge parameter λ? *What to submit: some commentary/your working.*

(b) In the logistic regression loss, and indeed in many machine learning problems, we often deal with expressions of the form

$$\text{LogSumExp}(x_1, \dots, x_n) = \log(e^{x_1} + \cdots + e^{x_n}).$$

This can be problematic from a computational perspective because it will often lead to numerical overflow. (To see this, note that $\log(\exp(x)) = x$ for any x, but try computing it naively with a large x, e.g. running `np.log(np.exp(1200))`). In this question we will explore a smart trick to avoid such issues.

(I) Let $x_* = \max(x_1, \dots, x_n)$ and show that

$$\text{LogSumExp}(x_1, \dots, x_n) = x_* + \log\left(e^{x_1 - x_*} + \cdots + e^{x_n - x_*}\right).$$

(II) Show each term inside of the log is a number greater than zero and less than equal to 1.

(III) Hence, explain why rewriting the LogSumExp function in this way avoids numerical overflow.

What to submit: some commentary/your working.

(c) For the remainder of this question we will work with the `songs.csv` dataset. The data contains information about various songs, and also contains a class variable outlining the genre of the song. If you are interested, you can read more about the data here, though a deep understanding of each of the features will not be crucial for the purposes of this assessment. Load in the data and preform the follwing preprocessing:

(I) Remove … "Artist Name", "Track Name", "key", "mode", "time_signature", "instrum…

(II) The curr… s, but logistic regression in the form we have described it here only wo… ion. We will restrict the data to classes 5 (hiphop) and 9 (pop). After re… s, re-code the variables so that the target variable is $y = 1$ for hiphop a…

(III) Remove … at have missing values for any of the features. Your remaining dataset s… 386 rows.

(IV) Use the `sklearn.model_selection.train_test_split` function to split your data into X_train, X_test, Y_train and Y_test. Use a `test_size` of 0.3 and a `random_state` of 23 for reproducibility.

(V) Fit the `sklearn.preprocessing.StandardScaler` to the resulting training data, and then use this object to scale both your train and test datasets.

(VI) Print out the first and last row of X_train, X_test, y_train, y_test (but only the first three columns of X_train, X_test).

What to submit: the print out of the rows requested in (VI). A copy of your code in solutions.py

(d) In homework 1 we fit the `sklearn.preprocessing.StandardScaler` to the entire dataset yet here we are being more careful by only fitting it to the training data. In the context of a real-world prediction problem, explain why what we are doing here is the proper thing to do. *What to submit: some commentary.*

(e) Write a function (in Python) `reg_log_loss(W, C, X, y)` that computes the loss achieved by a model $W = (c, w)$ with regularization parameter C on data X, y (consiting of n observations, and $y \in \{-1, 1\}$), where the loss is defined by:

$$\frac{1}{2}\|w\|_2^2 + C\sum_{i=1}^{n} \log(1 + e^{-y_i(w^T x_i + c)}).$$

Be careful to write your function in such a way to avoid overflow issues like the one described in (b). To do so, make use of `np.logaddexp` function when writing your implementation. **Hint:** $1 = e^0$. For comparison, you should get:

```
c = -1.4
w = 0.1 * np.ones(X_train.shape[1])
W = np.insert(w, 0, c)
reg_log_loss(W=W, C=0.001, X=X_train, y=y_train)    # returns 1.9356538918069666

```

Print out the result of running the above code but with parameters set to:
`w = 0.35 * np.ones(X_train.shape[1]), c=1.2.`

Note that combining the intercept and weight parameters into a single array (W) is important for getting your code to work in the next question.

What to submit: Your computed loss with `w = 0.35 * np.ones(X_train.shape[1]), c=1.2`, a screen shot of your code, and a copy of your code in solutions.py.

(f) Now that we have a way of quantifying the loss of any given model $W = (w, c)$ in a stable way, we are in a position to fit the logistic regression model to the data. Implement a function (in Python) `reg_log_fit(X, y, C)` that returns fitted parameters $\hat{W} = (\hat{w}, \hat{c})$. Your function should only make use of the `scipy.optimize.minimize` function which performs numerical optimization and can be applied directly to your `reg_log_loss` function from the previous question. Do this minimization [illegible] al parameters set to the same values as in the code snippet in (e), i.e.

```
w = 0.1 * np[illegible])
W0 = np.inse[illegible]

```

Further, use [illegible]ad' and `tol=1e-6`. Report the following quantities:

(I) Use the [illegible]og_loss` to compute the train and test losses of your resulting model. Re[illegible] of your model can be calculated using the formula: $\sigma(\hat{w}^T x_i + \hat{c})$ where x_i is a single feature vector.

(II) Fit a logistic regression model using `sklearn.linear_model.LogisticRegression` and use the parameters: `C=1, tol=1e-6, penalty='l2', solver='liblinear'`. Compute the train and test losses for this model.

Optional hint: you may find it easier to use `scipy.optimize.minimize` with a lambda function rather than directly applied to your `reg_log_loss` implementation. This would mean defining: `g = lambda x: reg_log_loss(x, *args)`

What to submit: train and test losses from your implementation as well as the `sklearn` model, a screen shot of your code, and a copy of your code in solutions.py.

(g) Up to this point, we have chosen the hyperparameter C arbitrarly. In this question we will study the effect that C has on the fitted model. We will focus on the ℓ_1 penalized version of the problem:

$$\|w\|_1 + C \sum_{i=1}^{n} \log(1 + e^{-y_i(w^T x_i + c)}),$$

and we will use the `sklearn` implementation:

`LogisticRegression(penalty='l1', solver='liblinear', C=c).`

Use the following code `Cs = np.linspace(0.001, 0.2, num=100)` to generate a list of C values. For each value of C, fit the model and store the coefficients of each feature. Create a plot with $\log(C)$ on the x-axis, and the coefficient value on the y-axis for each feature in each of the fitted models. In other words, the plot should describe what happens to each of the coefficients in your model for different choices of C. Based on this plot, explain why ℓ_1 regularization can be thought of as performing *feature selection*, and further comment on which features seem the most important to you (based on the plot) and why.

What to submit: a single plot, some commentary, a screen shot of your code and a copy of your code in solutions.py. Your plot must have a legend clearly representing the different features using the following color scheme: ['red', 'brown', 'green', 'blue', 'orange', 'pink', 'purple', 'grey', 'black', 'y']

(h) We now work through an example of using cross validation to choose the best choice of C based on the data. Specifically, we will use Leave-One-Out Cross Validation (LOOCV). Create a grid of C values using the code `Cs = np.linspace(0.0001, 0.8, num=25)`. LOOCV is computationally intesive so we will only work with the first 20% of the training data (the first $n = 544$ observations). For each data point in the training set $i = 1, \ldots, n$:

(I) For a given C value, fit the logistic regression model with ℓ_1 penalty on the dataset with point i removed. You should have a total of n models for each C value.

(II) For your i-th model (the one trained by leaving out point i) compute the leave-one-out error of predicting y_i (the log-loss of predicting the left out point).

(III) Average the losses over all n choics of i to get your CV score for that particular choice of C.

(IV) Repeat f[illegible]

Plot the leave[illegible] C and report the best C value. Note that for this question you are **not** perm[illegible] packages that implement cross validation though you are permitted to use [illegible]egression to fit the models, and `sklearn.metrics.log_loss` to compute t[illegible]ite the LOOCV implementatio code yourself from scratch and you must cre[illegible]m scratch using basic matplotlib functionality.

What to subm[illegible]mentary, a screen shot of any code used for this section.

Question 2. Perceptron Learning Variants

In this question we will take a closer look at the perceptron algorithm. We will work with the following data files `PerceptronX.csv` and `Perceptrony.csv`, which are the X, y for our problem, respectively. 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
from utils import *
```


In `utils.py` we have provided you with the `plot_perceptron` function that automatically plots a scatter of the data as well as your perceptron model.

(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 $t = 1, \dots,$ `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 parameters and number of iterations taken for convergence in the title. For example, you can use the following code for plotting:


```
w = ## your
fig, ax = pl
plot_percept                         # your implementation
ax.set_title                         ns={nmb_iter}")
plt.savefig(                             # if you want to save your plot as a png
plt.show()
```

What to subm… n 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 $t = 1, \dots,$ max_iter :

if there is an index i such that $y_i \sum_{j=1}^{n} y_j \alpha_j \langle x_j, x_i \rangle \le 0$:

$$\alpha_i^{(t+1)} = \alpha_i^{(t)} + 1; \quad t = t + 1$$

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 $t = 1, \dots,$ max_iter :

… index i such that $y_i \langle w^{(t)}, x_i \rangle + I_i r \leq 0$:

… $= w^{(t)} + y_i x_i; \quad t = t + 1; \quad I_i = 1$

… $w^{(t)}, t$

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.*

(f) (Optional Reading) The following post by Ben Recht gives a nice history of the perceptron and its importance in the development of machine learning theory.