COMP9417 - Machine Learning
Homework 1: Cleaning, Fitting, and Optimizing

Introduction In this homework, we will explore key steps in preparing and applying machine learning models, with a focus on logistic regression.

- In the first question, you will wrangle and clean a dataset, ensuring it is properly formatted and ready for modeling.
- In the second question, you will take a deeper look at logistic regression, learning how to fit the model and tune its hyperparameters using cross-validation.
- Finally, you will implement gradient descent to solve logistic regression, reinforcing your understanding of optimization techniques in machine learning.

By the end of this assignment, you will have hands-on experience with data preprocessing, model fitting, and optimization, all essential skills for building effective machine learning models.

Points Allocation There are a total of 30 marks.

- Question 1 a)- 1 h): 1 mark each
- Question 2 a): 3 marks
- Question 2 b): 4 marks
- Question 2 c): 5 marks
- Question 2 d): 4 marks
- Question 2 e): 3 marks
- Question 2 f): 3 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 about 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 Moodle announcements for any 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.
- You may **not** use SymPy or any other symbolic programming toolkits to answer the derivation questions. This will result in an automatic grade of zero for the relevant question. You must do the derivations manually.

When and Where to Submit

- Due date: Week 4, Thursday **March 13th**, 2025 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 made on Moodle, **no exceptions**.

In this question we will work with the data `heart.csv`. The data contains information about a set of 100 patients. The goal is to use this information to predict whether or not a patient has heart disease.

Question 1. Data Wrangling

The data in its current form is not ready for modeling. We will need to wrangle (clean) the data. This is outlined in the following tasks. **Throughout this question, you may only use python. For each subquestion, provide commentary (if needed) along with screenshots of code used. Please also provide a copy of the code in your solutions.py file.**

(a) Create a variable `X` containing only features, and a variably `y` containing the target (Heart_Disease). Remove the Last_Checkup feature.

(b) For Age, some values are negative. You are informed that this is a data entry error and negatives should be replaced with their positive versions. That is, $-x$ should be replaced with x.

(c) For Gender and Smoker, the variables have been coded in inconsistent ways. For example, Female gender is encoded as 'Female' and as 'F'. Write code to make these codings consistent. Then use categorical encoding instead. For gender, map (Male/M,Female/F, Unknown) to (0,1,2). For Smoker, map (No/N,Yes/Y,Nan) to (0,1,2).

(d) Blood pressure is given in the form systolic/diastolic. Write code to create two variables, systolic and diastolic. Remove the original blood pressure variable.

(e) Using `sklearn.model_selection.train_test_split`, split the data into training and test sets. Set the `test_size` parameter to 0.3, and the `random_state` to '2' for reproducibility.

(f) Now, note that some values in 'Age' in your test data are missing. We will manually impute values according to the following rule: If a Male (Female) is missing their age, set it to be the median of all other Male (Female) patients. Note that you should NOT use test data for this step, so your medians should be computed based on training set data. Be careful not to include missing values when calculating your median.

(g) Scale the columns: 'Age', 'Height_feet', 'Weight_kg', 'Cholesterol', 'Systolic', 'Diastolic' using a min-max normalizer. This means that for each feature, you should replace x with

$$\frac{x - \min}{\max - \min},$$

where $\min, \max$ are the minimum and maximum values for that feature column, respectively. Make sure to do this separately for train and test data.

(h) Plot a histogram of your target variable (from your training data). You should notice that a large portion of the target value is clustered around zero. Do you think linear regression is a reasonable model for this data? Create a new target variable by *quantizing* the original target variable. You can do this by setting values below a certain threshold (say 0.1) to be 0 and those above the threshold to be 1.

Solution:

Commentary is not really needed here, except for part (h) where students should note that the target values can naturally be clustered into those being very close to zero and those that aren't, and there is no data in the middle of the range and so linear regression is not really appropriate.

```python
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from sklearn.model_selection import train_test_split

# Load the data from the CSV file
data = pd.read_csv("data/heart.csv")

# 1a
X = data.drop(columns=['Heart_Disease', 'Last_Checkup'])
y = data['Heart_Disease']

# 1b
X['Age'] = np.abs(X['Age'])

# 1c
X['Gender'] = X['Gender'].replace({'Male': 0, 'M': 0, 'Female': 1, 'F': 1, 'Unknown':2})
X['Smoker'] = X['Smoker'].replace({'No': 0, 'N': 0, 'Yes': 1, 'Y': 1})
X['Smoker'] = X['Smoker'].fillna(2)

# 1d
X[['Systolic', 'Diastolic']] = X['Blood_Pressure'].str.split('/', expand=True)
X = X.drop(columns=['Blood_Pressure'])
X['Systolic'] = pd.to_numeric(X['Systolic'])
X['Diastolic'] = pd.to_numeric(X['Diastolic'])

# 1e
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=2)

# 1f
is_male_train = (X_train['Gender']==0)
is_female_train = (X_train['Gender']==1)
is_male_test = (X_test['Gender']==0)
is_female_test = (X_test['Gender']==1)
is_na_male = (X_test['Age'].isna()) & (X_test['Gender']==0)
is_na_female = (X_test['Age'].isna()) & (X_test['Gender']==1)

X_test.loc[is_na_male, 'Age'] = np.nanmedian(X_train[is_male_train]['Age'])
X_test.loc[is_na_female, 'Age'] = np.nanmedian(X_train[is_female_train]['Age'])

# 1g
def min_max_normalize(df, columns):
    df_normalized = df.copy()
    for column in columns:
        min_val = df[column].min()
        max_val = df[column].max()
        df_normalized[column] = (df[column] - min_val) / (max_val - min_val)
    return df_normalized

# Specify the columns to normalize
columns_to_normalize = ['Age', 'Height_feet', 'Weight_kg', 'Cholesterol', 'Systolic', '
    Diastolic']

# Normalize the specified columns
X_train = min_max_normalize(X_train, columns_to_normalize)
X_test = min_max_normalize(X_test, columns_to_normalize)

# 1h
```

```
plt.hist(y_train, bins=20)
plt.savefig("y_train_hist.png")
thresh = 0.1
plt.axvline(thresh, color='red')

y_train_q = y_train.copy()
y_test_q = y_test.copy()
y_train_q[y_train>=thresh] = 1
y_train_q[y_train<thresh] = 0
y_test_q[y_test>=thresh] = 1
y_test_q[y_test<thresh] = 0

```

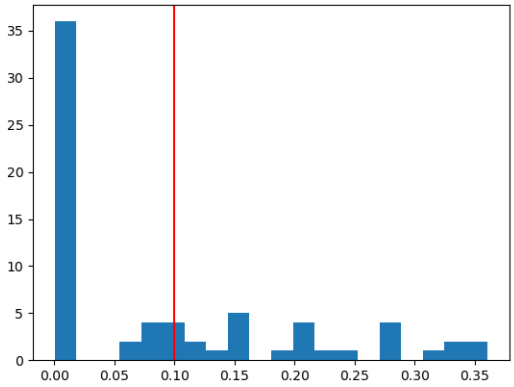


Figure 1: 1h

Question 2. Regularized Logistic Regression

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 an intercept $\beta_0 \in \mathbb{R}$, parameter vector $\beta = (\beta_1, \ldots, \beta_p)^T \in \mathbb{R}^p$, target $y_i \in \{0, 1\}$, and feature vector $x_i = (x_{i1}, x_{i2}, \ldots, x_{ip})^T \in \mathbb{R}^p$ for $i = 1, \ldots, n$, the (ℓ_2-regularized) log-loss that we will work with is:

$$L(\beta_0, \beta) = \text{penalty}(\beta) + \frac{\lambda}{n} \sum_{i=1}^{n} \left[y_i \ln \left(\frac{1}{\sigma(\beta_0 + \beta^T x_i)} \right) + (1 - y_i) \ln \left(\frac{1}{1 - \sigma(\beta_0 + \beta^T x_i)} \right) \right], \quad (1)$$

where $\sigma(z) = (1 + e^{-z})^{-1}$ is the logistic sigmoid, and λ is a hyper-parameter that controls the amount of regularization. The penalty term is a regularizer, for example we could take $\text{penalty}(\beta) = \frac{1}{2} \|\beta\|_2^2$. Note that you are provided with an implementation of this loss in `helper.py`.

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

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

It turns out that this objective is identical to our objective above (when the same penalty function is used), but only after re-coding the binary variables to be in $\{-1, 1\}$ instead of binary values $\{0, 1\}$. That is, $\widetilde{y}_i \in \{-1, 1\}$, whereas $y_i \in \{0, 1\}$. Argue rigorously that the two objectives 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 as you have seen in the class? *What to submit: some commentary/your working.*

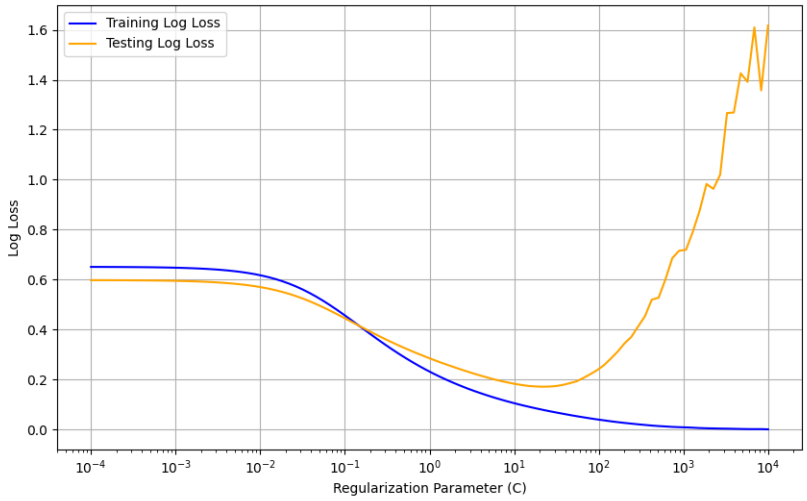
Solution:
We can focus on a single i for simplicity, note that the i-th summand in our loss equivalent to

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

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 the penalty parameter in the standard regularized regression problem we have seen in class. Note that this is a standard approach to rewriting loss functions, see for example the SVM objective from lectures. Different codings of the target variable lead to different looking loss functions, but they are effectively doing the same thing.

(b) Create a grid of 100 C values using the code `np.logspace(-4, 4, 100)`. For each C, fit a logistic regression model (using the `LogisticRegression` class in `sklearn`) on the training data. Plot a series showing the train and test log-losses against C. Be sure to use `predict_proba` to generate predictions from your fitted models to plug into the log-loss. Also, use ℓ_2 regularization, and the `lbfgs` solver when fitting your models. Discuss the shape of the two loss curves. How would you pick C based on these plots? State your choice of C.

Solution:
We see that as we increase C, the train loss keeps decreasing until it hits zero. This is a clear sign of over-fitting. The test loss on the other hand decreases until a point at which it starts to increase again. To pick C, we would look at the point at which the test loss is smallest.



The code used here is:

```python
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import log_loss
# Define a range of C values (inverse of regularization strength)
Cs = np.logspace(-4, 4, 100)  # Regularization parameters from 10^-4 to 10^4

train_ll = []
test_ll = []

# Fit Logistic Regression model for each C value
for C in Cs:
    model = LogisticRegression(C=C, solver='lbfgs', max_iter=200)
    model.fit(X_train, y_train_q)

    # Calculate training and testing accuracies
    y_train_pred = model.predict_proba(X_train)
    y_test_pred = model.predict_proba(X_test)

    train_ll.append(log_loss(y_train_q, y_train_pred))
    test_ll.append(log_loss(y_test_q, y_test_pred))

# Plotting the results
plt.figure(figsize=(10, 6))
plt.semilogx(Cs, train_ll, label='Training Log Loss', color='blue')
plt.semilogx(Cs, test_ll, label='Testing Log Loss', color='orange')
plt.xlabel('Regularization Parameter (C)')
plt.ylabel('Log Loss')
plt.legend()
plt.grid()
plt.savefig("figures/log-loss.png")
plt.show()

```

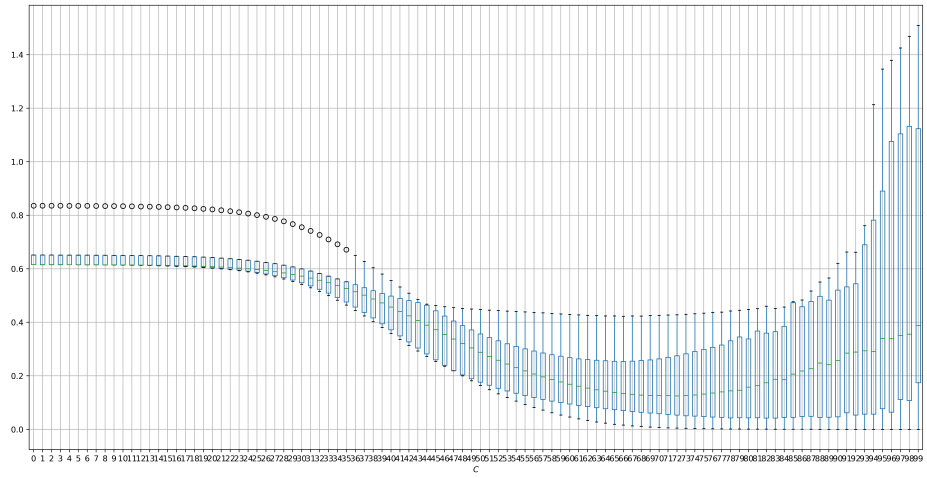
(c) In this part, we will take a closer look at choosing the hyperparameter C. Specifically, we will perform cross validation from scratch (**Do not use existing cross validation implementations here, doing so will result in a mark of zero.**)

Create a grid of C values as before. For each value of C in your grid, perform 5-fold cross validation (i.e. split the train data into 5 folds, fit logistic regression (using the settings from before) with the choice of C on 4 of those folds, and record the log-loss on the 5th, repeating the process 5 times.) For this question, we will take the first fold to be the first $N/5$ rows of the training data, the second fold to be the next $N/5$ rows, etc, where N denotes the number of observations in the training data.

To display the results, we will produce a plot: the x-axis should reflect the choice of C values and the y-axis will be the log-loss. For each C, plot a box-plot over the 5 CV scores. Report the value of C that gives you the best CV performance in terms of log-loss. 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 $\widetilde{y}$ ($\widetilde{y}_i \in \{-1, 1\}$) 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 best C is $C = 37.6494$, (or 1.576) on the log-scale. This model has perfect train accuracy and 0.93 test accuracy. The plot is:



The code used here is:

```
            from sklearn.metrics import log_loss, accuracy_score

```

```
N = X_train.shape[0]
nC = 100
Cs = np.logspace(-4, 4, nC)
K = 5
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='l2', solver='lbfgs', C=Cs[nc])
        mod.fit(X_train.iloc[cur_train_idx], y_train_q.iloc[cur_train_idx])

        # predictions
        pred_prob = mod.predict_proba(X_train.iloc[cur_test_idx])
        ll = log_loss(y_train_q.iloc[cur_test_idx], pred_prob)

        # 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='l2', solver='lbfgs', C=Cmin).fit(X_train,
    y_train_q)
print(f"C min = {Cmin}")
print("train acc: ", accuracy_score(y_train_q, final_mod.predict(X_train)))
print("test acc: ", accuracy_score(y_test_q, final_mod.predict(X_test)))


```

(d) 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:

```
from sklearn.model_selection import GridSearchCV
param_grid = {'C': Cs}
grid_lr = GridSearchCV(estimator=
                       LogisticRegression(penalty='l2',
                                          solver='lbfgs'),
                       cv=5,
                       param_grid=param_grid)
```

```
grid_lr.fit(X_train, y_train_q)

```

However, this gave me a very different answer to the result obtained by hand. 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 we computed by hand, 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 (c) is:

```
from sklearn.model_selection import KFold
grid_lr = GridSearchCV(estimator=LogisticRegression(penalty='l2',
                                                    solver='lbfgs'),
                       scoring='neg_log_loss',
                       cv=KFold(n_splits=5),
                       param_grid=param_grid)
grid_lr.fit(X_train, y_train_q)
print(grid_lr.best_params_['C'])

```

(e) Suppose that you were going to solve (1) using gradient descent and chose to update each coordinate individually. Derive gradient descent updates for each of the components $\beta_0, \beta_1, \ldots, \beta_p$ for step size η and regularization parameter λ. That is, derive explicit expressions for the terms † in the following:

$$\begin{aligned}
\beta_0^{(k)} &= \beta_0^{(k-1)} - \eta \times \dagger \\
\beta_1^{(k)} &= \beta_1^{(k-1)} - \eta \times \dagger \\
\beta_2^{(k)} &= \beta_2^{(k-1)} - \eta \times \dagger \\
&\vdots \\
\beta_p^{(k)} &= \beta_p^{(k-1)} - \eta \times \dagger
\end{aligned}$$

Make your expression as simple as possible, and be sure to include all your working. *what to submit: your coordinate level GD updates along with any working.*

Solution:

This follows using straight forward calculus:

$$\begin{aligned}
\beta_0^{(k)} &= \beta_0^{(k-1)} - \eta \times \left(-\frac{\lambda}{n} \sum_{i=1}^{n} (y_i - \sigma(\beta_0^{(k-1)} + x_i^T \beta^{(k-1)})) \right) \\
&= \beta_0^{(k-1)} + \eta \frac{\lambda}{n} \sum_{i=1}^{n} (y_i - \sigma(\beta_0^{(k-1)} + x_i^T \beta^{(k-1)})) \\
\beta_1^{(k)} &= \beta_1^{(k-1)} - \eta \times \left(\beta_1^{(k-1)} - \frac{\lambda}{n} \sum_{i=1}^{n} (y_i - \sigma(\beta_0^{(k-1)} + x_i^T \beta^{(k-1)})) x_{i1} \right) \\
&= \beta_1^{(k-1)} + \eta \frac{\lambda}{n} \sum_{i=1}^{n} (y_i - \sigma(\beta_0^{(k-1)} + x_i^T \beta^{(k-1)})) x_{i1} - \eta \beta_1^{(k-1)} \\
\beta_2^{(k)} &= \beta_2^{(k-1)} - \eta \times \left(\beta_2^{(k-1)} - \frac{\lambda}{n} \sum_{i=1}^{n} (y_i - \sigma(\beta_0^{(k-1)} + x_i^T \beta^{(k-1)})) x_{i2} \right) \\
&= \beta_2^{(k-1)} + \eta \frac{\lambda}{n} \sum_{i=1}^{n} (y_i - \sigma(\beta_0^{(k-1)} + x_i^T \beta^{(k-1)})) x_{i2} - \eta \beta_2^{(k-1)} \\
&\vdots \\
\beta_p^{(k)} &= \beta_p^{(k-1)} - \eta \times \left(\beta_p^{(k-1)} - \frac{\lambda}{n} \sum_{i=1}^{n} (y_i - \sigma(\beta_0^{(k-1)} + x_i^T \beta^{(k-1)})) x_{ip} \right) \\
&= \beta_p^{(k-1)} + \eta \frac{\lambda}{n} \sum_{i=1}^{n} (y_i - \sigma(\beta_0^{(k-1)} + x_i^T \beta^{(k-1)})) x_{ip} - \eta \beta_p^{(k-1)}.
\end{aligned}$$

(f) For the non-intercept components $\beta_1, \ldots, \beta_p$, re-write the gradient descent updates of the previous question in vector form, i.e. derive an explicit expression for the term † in the following:

$$\beta^{(k)} = \beta^{(k-1)} - \eta \times \dagger$$

Your expression should only be in terms of β_0, β, x_i and y_i. Next, let $\gamma = [\beta_0, \beta^T]^T$ be the $(p+1)$-dimensional vector that combines the intercept with the coefficient vector β, write down the update

$$\gamma^{(k)} = \gamma^{(k-1)} - \eta \times \dagger.$$

Note: This final expression will be our vectorized implementation of gradient descent. The point of the above exercises is just to be careful about the differences between intercept and non-intercept parameters. Doing GD on the coordinates is extremely innefficient in practice. *what to submit: your vectorized GD updates along with any working.*

Solution:

$$\begin{aligned}
\beta^{(k)} &= \beta^{(k-1)} - \eta \times \left(\beta^{(k-1)} - \frac{\lambda}{n} \sum_{i=1}^{n} (y_i - \sigma(\beta_0^{(k-1)} + x_i^T \beta^{(k-1)}) x_i \right) \\
&= \beta^{(k-1)} - \eta \beta^{(k-1)} + \eta \frac{\lambda}{n} \sum_{i=1}^{n} (y_i - \sigma(\beta_0^{(k-1)} + x_i^T \beta^{(k-1)}) x_i \\
&= \beta^{(k-1)} - \eta \beta^{(k-1)} + \eta \frac{\lambda}{n} X^T (y - \sigma(\beta_0^{(k-1)} 1_n + X \beta^{(k-1)})),
\end{aligned}$$

where the sigmoid in the final line is applied element-wise. Either of the final two expressions is sufficient for full marks. Now, we also immediately have

$$\gamma^{(k)} = \gamma^{(k-1)} - \eta \times \begin{bmatrix} -\frac{\lambda}{n} (y - \sigma(\beta_0^{(k-1)} 1_n + X \beta^{(k-1)})) \\ \beta^{(k-1)} - \frac{\lambda}{n} X^T (y - \sigma(\beta_0^{(k-1)} 1_n + X \beta^{(k-1)})) \end{bmatrix},$$

where 1_n is the vector of ones.