3F8 Inference: Coursework

José Miguel Hernández-Lobato & Richard E. Turner

Due: within 2 weeks of the demonstration session, 4pm on the deadline date

In this assignment you will implement a classifier, apply it to a simple dataset, evaluate the results using several metrics, and (hopefully) improve the performance using a non-linear feature expansion. Your answers should contain an explanation of what you do and where code is asked for you need only include the central commands (complete listings are unnecessary). You must also give an interpretation of what the numerical values and plots you provide mean. Why are the results the way they are? Emphasis should be on the understanding of the technical content and discussion of the results, rather than the writing style. Hand in a maximum of 6 pages. Longer reports will have 0.5 mark penalty.

1 Preparation exercises

Please complete the two exercises below before attending the first demonstrated practical session. The results and interpretations for these two exercises should also be included in the final report.

a) Consider the Logistic Classification model (aka Logistic Regression, see below). Derive the gradients of the log-likelihood of the parameters, $\frac{\partial}{\partial \beta}\mathcal{L}(\beta)$ where $\mathcal{L}(\beta) = \log P(y|X,\beta)$.

Logistic Classification

To assist you with the derivation in part (a), we briefly recapitulate Logistic Classification. In this model each datapoint's class label is assumed to be generated from a Bernoulli distribution in which the probability of a positive class label is given by a logistic function applied to weighted inputs,

$$P(y^{(n)} = 1|\tilde{x}^{(n)}) = \frac{1}{1+\exp(-\beta^T\tilde{x}^{(n)})} = \sigma(\beta^T\tilde{x}^{(n)})$$
$$P(y^{(n)} = 0|\tilde{x}^{(n)}) = 1 - \sigma(\beta^T\tilde{x}^{(n)}) = \sigma(-\beta^T\tilde{x}^{(n)}).$$

Here the inputs are augmented with a fixed unit input $\tilde{x}^{(n)} = (1, x^{(n)})$ which enables biases to be handled simply as $\beta^T\tilde{x}^{(n)} = \beta_0 + \sum_{d=1}^{D}\beta_d x_d^{(n)}$.
The probability of the dataset is a product of Bernoulli distributions,

$$P(y|X,\beta) = \prod_{n=1}^{N} P(y^{(n)}|\tilde{x}^{(n)}) = \prod_{n=1}^{N} \sigma(\beta^T\tilde{x}^{(n)})^{y^{(n)}}(1-\sigma(\beta^T\tilde{x}^{(n)}))^{1-y^{(n)}}.$$

b) Write pseudocode to estimate the parameters β using gradient ascent of the log-likelihood $\beta^{(new)} = \beta^{(old)} + \eta\frac{\partial}{\partial\beta}\mathcal{L}(\beta^{(old)})$ from X and y. You should use vectorized code: avoid having a loop over the rows of X. Include a short description detailing how you would choose the learning rate η.

2 Remaining exercises

c) Read the instructions for computer use in the DPO here. The dataset for this practical can be found in the Moodle page in the files `X.txt` and `y.txt`. You can load the data into python using

```
import numpy as np

X = np.loadtxt('X.txt')
y = np.loadtxt('y.txt')
```

This creates two variables given by

> `X` is a 1000×2 dimensional array containing two-dimensional input features for 1000 datapoints. Rows of `X` are denoted as column vectors $x^{(n)}$ below. Elements of `X` are denoted $x_d^{(n)}$.
>
> `y` is a 1000 dimensional binary vector containing the class labels. Elements of `y` are denoted $y^{(n)}$ below.

Visualise the dataset in the two-dimensional input space displaying each datapoint's class label. For this you can use the python function `plot_data` from the Appendix. Discuss how well a classifier with a linear class boundary is likely to perform on these data.

d) Split the data randomly into training and test sets with 800 and 200 data points, respectively. Transform the pseudocode from the preparation exercise (a) into python code and use it to train the Logistic Classification method on the dataset. Report training curves showing the log-likelihood on training and test datasets per datapoint (averaged) as the optimisation proceeds, for this, you can use the functions `plot_ll` and `compute_average_ll` from the Appendix. Visualise the predictions by adding probability contours to the plots made in part (c), for this you can use the functions `plot_predictive_distribution` from the Appendix, with argument the function `predict_for_plot`.

e) Report the final training and test log-likelihoods per datapoint. For the test data, apply a threshold to the probabilistic predictions so that those greater than $\tau = 1/2$ are assigned a positive predicted class label $\hat{y} = 1$ and those equal or below are assigned to a negative predicted class label $\hat{y} = 0$. Use these hard predictions to obtain and report the 2×2 confusion matrix:

		predicted label, $\hat{y}$	
		0	1
true label, y	0	fraction of true negatives $P(\hat{y} = 0 \mid y = 0)$	fraction of false positives $P(\hat{y} = 1 \mid y = 0)$
	1	fraction of false negatives $P(\hat{y} = 0 \mid y = 1)$	fraction of true positives $P(\hat{y} = 1 \mid y = 1)$

f) Expand the inputs through a set of radial basis functions (RBFs) centred on the training datapoints. The feature-expanded inputs now become $\tilde{x}_1^{(n)} = 1$ (to handle the bias terms as before) and $\tilde{x}_{m+1}^{(n)} = \exp\left(-\frac{1}{2l^2}\sum_{d=1}^{2}(x_d^{(n)} - x_d^{(m)})^2\right)$ (the RBF functions). In other words, the $(m+1)$th feature is given by a radial basis function centred on the mth training datapoint with width l. The dimensionality of the inputs should now be $N_{\text{train}} + 1$ where N_{train} is the number of training datapoints. For this, you can use the function `expand_inputs` from the Appendix.

Train the Logistic Classification model on the feature-expanded inputs and display the new predictions for three choices of RBF width $l = \{0.01, 0.1, 1\}$. Visualise the predictions using probability contours as in part (d). For this, you can again use the function `plot_predictive_distribution` from the Appendix, but this time with argument the function `predict_for_plot_expanded_features`. You will need to adjust the learning rate appropriately for each choice of length-scale, l.

g) Report the final training and test log-likelihoods per datapoint, the 2×2 confusion matrices for the three models trained in part (f). Compare the results to those obtained using the original inputs and explain your findings.

To avoid numerical errors when computing the log-likelihoods, you may find the following fact useful: $\log(\sigma(\beta^T \tilde{x}^{(n)})) \to \beta^T \tilde{x}^{(n)}$ as $\beta^T \tilde{x}^{(n)} \to -\infty$.

3 Appendix

```
import matplotlib.pyplot as plt

##
# X: 2d array with the input features
# y: 1d array with the class labels (0 or 1)
#

def plot_data_internal(X, y):
    x_min, x_max = X[ :, 0 ].min() - .5, X[ :, 0 ].max() + .5
    y_min, y_max = X[ :, 1 ].min() - .5, X[ :, 1 ].max() + .5
    xx, yy = np.meshgrid(np.linspace(x_min, x_max, 100), \
        np.linspace(y_min, y_max, 100))
    plt.figure()
    plt.xlim(xx.min(), xx.max())
    plt.ylim(yy.min(), yy.max())
    ax = plt.gca()
    ax.plot(X[y == 0, 0], X[y == 0, 1], 'ro', label = 'Class 1')
    ax.plot(X[y == 1, 0], X[y == 1, 1], 'bo', label = 'Class 2')
    plt.xlabel('X1')
    plt.ylabel('X2')
    plt.title('Plot data')
    plt.legend(loc = 'upper left', scatterpoints = 1, numpoints = 1)
    return xx, yy

##
# X: 2d array with the input features
# y: 1d array with the class labels (0 or 1)
#

def plot_data(X, y):
    xx, yy = plot_data_internal(X, y)
    plt.show()

##
# x: input to the logistic function
#

def logistic(x): return 1.0 / (1.0 + np.exp(-x))

##
# X: 2d array with the input features
# y: 1d array with the class labels (0 or 1)
# w: current parameter values
#

def compute_average_ll(X, y, w):
    output_prob = logistic(np.dot(X, w))
    return np.mean(y * np.log(output_prob) + (1 - y) * np.log(1.0 - output_prob))
```

```
##
# ll: 1d array with the average likelihood per data point and dimension equal
#     to the number of training epochs.
#

def plot_ll(ll):
    plt.figure()
    ax = plt.gca()
    plt.xlim(0, len(ll) + 2)
    plt.ylim(min(ll) - 0.1, max(ll) + 0.1)
    ax.plot(np.arange(1, len(ll) + 1), ll, 'r-')
    plt.xlabel('Steps')
    plt.ylabel('Average log-likelihood')
    plt.title('Plot Average Log-likelihood Curve')
    plt.show()

##
# x: 2d array with input features at which to compute predictions.
#
# (uses parameter vector w which is defined outside the function's scope)
#

def predict_for_plot(x):
    x_tilde = np.concatenate((np.ones((x.shape[ 0 ], 1 )), x), 1)
    return logistic(np.dot(x_tilde, w))

##
# X: 2d array with the input features
# y: 1d array with the class labels (0 or 1)
# predict: function that recives as input a feature matrix and returns a 1d
#          vector with the probability of class 1.

def plot_predictive_distribution(X, y, predict):
    xx, yy = plot_data_internal(X, y)
    ax = plt.gca()
    X_predict = np.concatenate((xx.ravel().reshape((-1, 1)), \
        yy.ravel().reshape((-1, 1))), 1)
    Z = predict(X_predict)
    Z = Z.reshape(xx.shape)
    cs2 = ax.contour(xx, yy, Z, cmap = 'RdBu', linewidths = 2)
    plt.clabel(cs2, fmt = '%2.1f', colors = 'k', fontsize = 14)
    plt.show()

##
# l: hyper-parameter for the width of the Gaussian basis functions
# Z: location of the Gaussian basis functions
# X: points at which to evaluate the basis functions

def expand_inputs(l, X, Z):
    X2 = np.sum(X**2, 1)
    Z2 = np.sum(Z**2, 1)
    ones_Z = np.ones(Z.shape[ 0 ])
    ones_X = np.ones(X.shape[ 0 ])
    r2 = np.outer(X2, ones_Z) - 2 * np.dot(X, Z.T) + np.outer(ones_X, Z2)
    return np.exp(-0.5 / l**2 * r2)
```

```
##
# x: 2d array with input features at which to compute the predictions
#    using the feature expansion
#
# (uses parameter vector w and the 2d array X with the centers of the basis
# functions for the feature expansion, which are defined outside the function's
# scope)
#

def predict_for_plot_expanded_features(x):
    x_expanded = expand_inputs(l, x, X)
    x_tilde = np.concatenate((np.ones((x_expanded.shape[ 0 ], 1 )), x_expanded), 1)
    return logistic(np.dot(x_tilde, w))
```