ECE521: Inference Algorithms and Machine Learning
University of Toronto

Assignment 1:
k-NN and Linear Regression

TA: Use Piazza for Q&A
Due date: Feb 7 midnight, 2017
Electronic submission to: ece521ta@gmail.com

General Note:

- In this assignment, you will derive and implement some simple machine learning models using TensorFlow, apply your solutions to some simulated datasets, and analyze the results. You are encouraged to look up TensorFlow APIs for useful utility functions to help you complete the programming portion of the assignment at: `https://www.tensorflow.org/api_docs/python/`.
- Full points are given for complete solutions, including justifying the choices or assumptions you made to solve each question. Both your complete source code and written up report should be included in the final submission.
- Homework assignments are to be solved by yourself or in groups of two. You are encouraged to discuss the assignment with other students, but you must solve it within your own group. Make sure to be closely involved in all aspects of the assignment. If you work in a group, please indicate the contribution percentage from each group member at the beginning of your report.

1 k-Nearest Neighbour [16 pt]

k-NN is one of the simplest nonparametric machine learning models. In this part of the assignment, you will first implement the basic k-NN algorithm in TensorFlow. You will then derive a soft extension to the standard k-NN and (optionally) investigate its connection with the Gaussian process method for regression. In this assignment, when $k > 1$, we will aggregate the k nearest neighbours' predictions by averaging their target values for regression tasks. For classification tasks, a majority voting scheme is assumed and tie breaker is dealt by uniformly pick a class label from the ties.

1.1 Geometry of k-NN [6 pt.]

1. Describe a 1-D dataset of two classes such that when applying k-NN, the classification accuracy of the k-NN classifier on the *training set* is a periodic function of the hyper-parameter k. [2 pt.]

2. Owing to something called the *curse of dimensionality*, the k-NN model can encounter severe difficulty with high-dimensional data using the sum-of-squares distance function. In this question, we will investigate the curse of dimensionality in its worst-case scenario. Consider a dataset consisting of a set of N-dimensional input data $\{\mathbf{x}\}$, $\mathbf{x} \in \mathbb{R}^N$. The dataset is i.i.d. and is sampled from an N-dimensional Gaussian distribution such that each dimension is independent with mean zero and variance σ^2, i.e. $P(\mathbf{x}) = \prod_{n=1}^{N} \mathcal{N}(x_n; 0, \sigma^2)$. Show that for two random samples $\mathbf{x}^{(i)}$ and $\mathbf{x}^{(j)}$ from this dataset,

$$\text{var}\left(\frac{\|\mathbf{x}^{(i)} - \mathbf{x}^{(j)}\|_2^2}{\mathbb{E}[\|\mathbf{x}^{(i)} - \mathbf{x}^{(j)}\|_2^2]}\right) = \frac{N+2}{N} - 1.$$

   (This is known as the variance of the squared ℓ_2 (squared Euclidean) distance ratio.) Next, show that it vanishes as $N \to \infty$, i.e. a test data point drawn from this dataset is equally close to all the training examples in high dimensional space. [4 pt.]

Hints: You can make use of the following properties of the difference of two independent Gaussian random variables, $d_n = x_n^{(i)} - x_n^{(j)}$:

- The difference is an independent Gaussian, so $P(d_n) = \mathcal{N}(d_n; 0, 2\sigma^2)$ and $\mathbb{E}[d_n d_m] = \mathbb{E}[d_n]\mathbb{E}[d_m]$
- Its fourth moment is $\mathbb{E}[d_n^4] = 3(\sqrt{2}\sigma)^4$
- $\mathbb{E}[d_n^2 d_m^2] = \mathbb{E}[d_n^2]\mathbb{E}[d_m^2]$

1.2 Euclidean distance function [4 pt.]

The squared Euclidean distance between two N-dimensional vectors $\mathbf{x}$ and $\mathbf{z}$ is given by $\|\mathbf{x}-\mathbf{z}\|_2^2 = (\mathbf{x}-\mathbf{z})^\top(\mathbf{x}-\mathbf{z}) = \sum_{n=1}^{N}(x_n - z_n)^2$. Assume we have two groups of N-dimensional data points represented respectively by a $B \times N$ matrix $\mathbf{X} = \begin{bmatrix} \mathbf{x}^{(1)^T} \\ \vdots \\ \mathbf{x}^{(B)^T} \end{bmatrix}$ and a $C \times N$ matrix $\mathbf{Z} = \begin{bmatrix} \mathbf{z}^{(1)^T} \\ \vdots \\ \mathbf{z}^{(C)^T} \end{bmatrix}$. Let the function $D(\mathbf{X}, \mathbf{Z})$ return a $B \times C$ matrix $\begin{bmatrix} \|\mathbf{x}^{(1)} - \mathbf{z}^{(1)}\|_2^2 & \dots & \|\mathbf{x}^{(1)} - \mathbf{z}^{(C)}\|_2^2 \\ \vdots & \ddots & \vdots \\ \|\mathbf{x}^{(B)} - \mathbf{z}^{(1)}\|_2^2 & \dots & \|\mathbf{x}^{(B)} - \mathbf{z}^{(C)}\|_2^2 \end{bmatrix}$ containing the pairwise Euclidean distances.

1. **Inner product:** Consider the special case in which all the input vectors have the same magnitude in a training dataset, $\|\mathbf{x}^{(1)}\|_2^2 = \cdots = \|\mathbf{x}^{(M)}\|_2^2$. Show that in order to find the

nearest neighbour of a test point $\mathbf{x}^*$ among the training set, it is sufficient to just compare and rank the negative inner product between the training and test data: $-{\mathbf{x}^{(m)}}^T\mathbf{x}^*$. [2 pt.]

2. **Pairwise distances:** Write a vectorized Tensorflow Python function that implements the pairwise squared Euclidean distance function for two input matrices. It should not contain loops and you should make use of Tensorflow broadcasting. Include the snippets of the Python code. [2 pt.]

1.3 Making predictions [6 pt.]

Suppose a particular training dataset consists of M training examples with input features $\mathbf{X} = \begin{bmatrix} {\mathbf{x}^{(1)}}^T \\ \vdots \\ {\mathbf{x}^{(M)}}^T \end{bmatrix}$ and targets $\mathbf{Y} = \begin{bmatrix} {\mathbf{y}^{(1)}}^T \\ \vdots \\ {\mathbf{y}^{(M)}}^T \end{bmatrix}$ represented by two matrices. In k-NN modelling, a standard way to obtain a prediction for a new test point $\mathbf{x}^*$ is to average the targets/labels of the k training points nearest to the test point $\mathbf{x}^*$. Let $\mathcal{N}^k_{\mathbf{x}^*} \subseteq \{\mathbf{x}^{(1)}\}, \ldots, \mathbf{x}^{(M)}\}$ denote the neighbourhood set of the k training examples closest to $\mathbf{x}^*$. A k-NN prediction function $\hat{\mathbf{y}}(\mathbf{x}^*)$ can therefore be written concisely as follows:

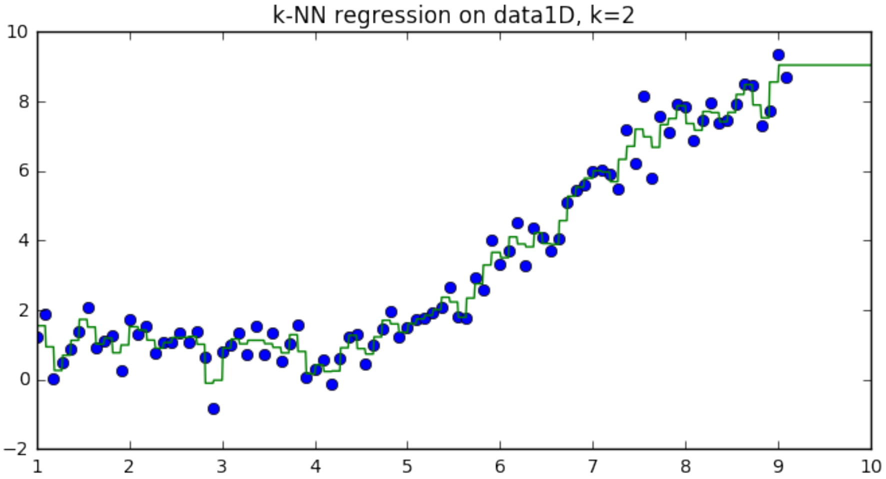
$$\hat{\mathbf{y}}(\mathbf{x}^*) = \mathbf{Y}^T\mathbf{r}^*, \text{ where } \mathbf{r}^* = [r_1, \ldots, r_M], r_m = \begin{cases} \frac{1}{k}, & \mathbf{x}^{(m)} \in \mathcal{N}^k_{\mathbf{x}^*} \\ 0, & \text{otherwise.} \end{cases}$$

$\mathbf{r}^*$ can be understood as a "responsibility" vector that encodes the contribution of each training example to the final prediction. In a standard k-NN model, the k nearest training points will have equal contribution. To measure the performance of the k-NN predictions, we will consider the mean squared error (MSE) loss:

$$\mathcal{L} = \frac{1}{2M} \sum_{m=1}^{M} \|\hat{\mathbf{y}}(\mathbf{x}^{(m)}) - \mathbf{y}^{(m)}\|_2^2$$

The first dataset for this assignment is an 1-D toy dataset, *data1D*, with 100 data points. We randomly partition 80 data points as training examples and split the rest into 10 validation points and 10 test points. You can generate the training and test dataset using the following python commands:

```
import numpy as np
np.random.seed(521)
Data = np.linspace(1.0 , 10.0 , num =100) [:, np. newaxis]
Target = np.sin( Data ) + 0.1 * np.power( Data , 2) \
         + 0.5 * np.random.randn(100 , 1)
randIdx = np.arange(100)
np.random.shuffle(randIdx)
trainData, trainTarget  = Data[randIdx[:80]], Target[randIdx[:80]]
validData, validTarget = Data[randIdx[80:90]], Target[randIdx[80:90]]
testData, testTarget = Data[randIdx[90:100]], Target[randIdx[90:100]]
```



1. **Choosing the nearest neighbours:** Write a vectorized Tensorflow Python function that takes a pairwise distance matrix and returns the responsibilities of the training examples to a new test data point. It should not contain loops. You may find the *tf.nn.top_k* function useful. Include the relevant snippets of your Python code. [2 pt.]

2. **Prediction:** For the dataset *data1D*, compute the above k-NN prediction function with $k = \{1, 3, 5, 50\}$. For each of these values of k, compute and report the training MSE loss, validation MSE loss and test MSE loss. Choose the best k using the validation error. For the different k value, plot the prediction function for $x \in [0, 11]$ similar to the figure shown above. (You may create an input matrix $X = np.linspace(0.0, 11.0, num = 1000)[:, np.newaxis]$ and use Matplotlib to plot the predictions.) Comment on this plot and how you would pick the best k from it. [4 pt.]

1.4 Soft k-NN and Gaussian process [Bonus: 8 pt.]

The standard k-NN algorithm makes a "hard" decision to pick exactly k points. It is often beneficial to relax this constraint and consider making prediction from the contributions of all the training examples, weighted by their distance to the test point. One particular way to turn a distance matrix into "responsibilities" is to exponentiate the negative distances. (Remember the smaller the distance, the more contribution it should have to the final prediction.) We use $\mathcal{K}$ notation for the exponentiated negative pairwise distances, which is also called the squared-exponential kernel. We have that

$$\mathcal{K}_{\mathbf{x}^*\mathbf{X}} = \exp(-\lambda D(\mathbf{x}^*, \mathbf{X})) = \begin{bmatrix} \exp(-\lambda \|\mathbf{x}^{(1)} - \mathbf{x}^*\|_2^2) \\ \vdots \\ \exp(-\lambda \|\mathbf{x}^{(M)} - \mathbf{x}^*\|_2^2) \end{bmatrix}$$

in which λ is known as a "hyper-parameter". There are different normalization schemes available for the contribution from each training example. In particular, the following two normalization schemes correspond to two widely used machine learning models:

$$\begin{aligned}\text{soft k-NN:} \quad & \mathbf{r}^* = \frac{\mathcal{K}_{\mathbf{x}^*\mathbf{X}}}{\sum_{m=1}^{M} \mathcal{K}_{\mathbf{x}^*\mathbf{x}^{(m)}}}, \\ \text{Gaussian process regression:} \quad & \mathbf{r}^* = \mathcal{K}_{\mathbf{XX}}^{-1}\mathcal{K}_{\mathbf{x}^*\mathbf{X}}.\end{aligned}$$

1. **Soft decisions:** Write a Tensorflow Python program based on the soft k-NN model to compute predictions on the *data1D.npy* dataset. Set $\lambda = 0.1$ and plot the test-set prediction of the model. Repeat all for the Gaussian process regression model. Comment on the difference you observe between your two programs. Include the relevant snippets of Python code. [2 pt.]

2. **Conditional distribution of a Gaussian:** Here we take a closer look at the Gaussian process regression model. In order to obtain a better intuition behind why it is sensible to normalize the "responsibilities" with the inverse of the training data kernel $\mathcal{K}_{\mathbf{XX}}^{-1}$, we will derive the Gaussian process prediction from a conditional distribution of the multivariate Gaussian distribution. Consider $M+1$ Gaussian random variables, $\mathbf{y} = \begin{bmatrix} y^* \\ y^{(1)} \\ \cdots \\ y^{(M)} \end{bmatrix}$ with a joint mean vector $\boldsymbol{\mu} = \mathbf{0}$ and a covariance matrix $\mathbf{\Sigma}$. Each of these random variables has a corresponding position in the input space denoted as $x^*, x^{(1)}, \dots, x^{(M)}$. You can think of $y^{(1)}, \dots, y^{(M)}$ as the training targets and y^* as our prediction. The covariance matrix of this multivariate Gaussian are related to the relative location and the distance between the data points in the input space. In particular, the entries of the covariance matrix are given by the squared-exponential kernel of the inputs, that is the i^{th} row and j^{th} column of the covariance matrix $\mathbf{\Sigma}$ is $\text{cov}(y^{(i)}, y^{(j)}) = \mathcal{K}_{x^{(i)}x^{(j)}}$. To simplify the notation, we absorb M random variables into a vector denoted as $\mathbf{y}_{\text{train}} = \begin{bmatrix} y^{(1)} \\ \vdots \\ y^{(M)} \end{bmatrix}$. We can then use stacked block notation for the joint covariance matrix $\mathbf{\Sigma} = \begin{bmatrix} \Sigma_{y^*y^*} & \Sigma_{y^*\mathbf{y}_{\text{train}}} \\ \Sigma_{\mathbf{y}_{\text{train}}y^*} & \Sigma_{\mathbf{y}_{\text{train}}\mathbf{y}_{\text{train}}} \end{bmatrix}$. The joint probability is

$$\begin{aligned} P(\hat{y}, y^{(1)}, \dots, y^{(M)}) = P(\mathbf{y}) = \mathcal{N}(\mathbf{y}; \mathbf{0}, \mathbf{\Sigma}) \\ =& (2\pi)^{-\frac{M+1}{2}} |\mathbf{\Sigma}|^{-0.5} \exp\{-\frac{1}{2}\mathbf{y}^T\mathbf{\Sigma}^{-1}\mathbf{y}\} \end{aligned}$$

Given that we observed the random variables $\mathbf{y}_{\text{train}}$, we would like to perform inference on the value of y^* through the conditional distribution $P(y^*|\mathbf{y}_{\text{train}}) = \mathcal{N}(y^*; \mu^*, \Sigma^*)$. Derive the expression for the conditional mean μ^* and variance Σ^* in terms of $\mathbf{y}_{\text{train}}$, $\Sigma_{y^*y^*}$, $\Sigma_{\mathbf{y}_{\text{train}}\mathbf{y}_{\text{train}}}$ and $\Sigma_{\mathbf{y}_{\text{train}}y^*}$. [8 pt.] You will find the following matrix-inverse identity useful:

$$\begin{bmatrix} A & B \\ C & D \end{bmatrix}^{-1} = \begin{bmatrix} (A - (BD^{-1}C)^{-1} & -A^{-1}B(D - CA^{-1}B)^{-1} \\ -D^{-1}C(A - BD^{-1}C)^{-1} & (D - CA^{-1}B)^{-1} \end{bmatrix}$$

2 Linear and logistic regression [14 pt.]

k-NN makes very mild structural assumptions: its predictions are often accurate but can be vulnerable to over-fitting for small datasets. On the contrary, linear models make a heavier assumption about structure, namely that the output prediction of the model is a weighted linear combination of the inputs. This assumption yields predictions that are stabler (less vulnerable to overfitting) but possibly less accurate. In this section, we will implement the stochastic gradient descent algorithm to train linear models with weight decay. In the first tutorial, you trained a simple linear regression model using gradient descent on a toy synthetic dataset. Linear regression can also be used for classification in which the training targets are either 0 or 1. Once the model is trained, we can determine an input's class label by thresholding the model's prediction. We will consider the following mean squared error (MSE) loss function $\mathcal{L}_\mathcal{D}$ and weight decay loss $\mathcal{L}_W$ for training a linear regression model, in which the goal is to find the best total loss,

$$\begin{aligned}\mathcal{L} =& \mathcal{L}_\mathcal{D} + \mathcal{L}_W \\ =& \sum_{m=1}^{M} \frac{1}{2M} \| W^T \mathbf{x}^{(m)} + b - y^{(m)} \|_2^2 + \frac{\lambda}{2} \| W \|_2^2\end{aligned}$$

2.1 Geometry of linear regression [8 pt.]

1. Is the ℓ_2 penalized mean squared error loss $\mathcal{L}$ a convex function of W? Show your results using the "Jensen inequality" (see the convexity inequality from the lecture of 16 January). Is the error a convex function of the bias, b? [2 pt.]

2. Consider learning a linear regression model of a single scalar output with a weight vector $W \in \mathbb{R}^N$, bias $b \in \mathbb{R}$ and no regularizer, on a dataset $\mathcal{D} = \{(\mathbf{x}^{(1)}, y^{(1)}), \dots, (\mathbf{x}^{(M)}, y^{(M)})\}$. If the nth input dimensions, x_n, is scaled by a constant factor of $\alpha > 1$ and shifted by a constant $\beta > 1$ for every datum in the training set, what will happen to the optimal weights and bias after learning as compared to the optimal weights and biases learned from the non-transformed (original) data? What will happen to the new minimum value of the loss function comparing to the original minimum loss? Explain your answer clearly. [2 pt.]

3. On the same linearly transformed dataset from the last question, consider learning an ℓ_2-regularized linear regression model in which the weight-decay coefficient is set to a constant λ. Describe what will happen to the weights and the bias learned on this transformed dataset as compared to those learned from non-transformed (original) data. What will happen to the new minimum value of the loss function, again comparing to the minimum loss before the transformation? Explain your answer clearly. [2 pt.]

4. Consider a multi-class classification task with $D > 2$ classes. Suppose that somehow you were constrained by the software packages such that you were only able to train binary classifiers. Design a procedure/scheme to solve a multi-class classification task using a number of binary classifiers. [2 pt.]

2.2 Stochastic gradient descent [6 pt.]

We will train a digit classifier on the Tiny MNIST dataset available in *tinymnist.npz*, using the stochastic gradient descent (SGD) algorithm. The training targets of linear regression are either 0 or 1, representing digit class 3 and 5. There are 700 training images, 100 validation images and 400 test images. All the images are 8 by 8 pixels. You can set up the dataset in Python using NumPy with the following:

```
import numpy as np
with np.load ("tinymnist.npz") as data :
    trainData, trainTarget = data ["x"], data["y"]
    validData, validTarget = data ["x_valid"], data ["y_valid"]
    testData, testTarget = data ["x_test"], data ["y_test"]
```

We will be using the validation set to tune the hyper-parameter of the weight decay regularizer.

1. **Tuning the learning rate**: Write a Tensorflow script that implements linear regression and the stochastic gradient descent algorithm with mini-batch size $B = 50$. Train the linear regression model on the tiny MNIST dataset by using SGD to optimize the total loss $\mathcal{L}$. Set the weight decay coefficient $\lambda = 1$. Tune the learning rate η to obtain the best overall convergence speed of the algorithm. Plot the total loss function $\mathcal{L}$ vs. the number of updates for the best learning rate found and discuss the effect of the learning-rate value on training convergence [2 pt.]

2. **Effect of the mini-batch size**: Run the SGD algorithm with $B = \{10, 50, 100, 700\}$ and tune the learning rate separately for each mini-batch size. Plot the total loss function $\mathcal{L}$ vs. the number of updates for each mini-batch size. What is the overall best mini-batch size in terms of training time? Comment on your observation. [2 pt.]

3. **Generalization**: Run SGD with mini-batch size $B = 50$ and use the validation set performance to choose the best weight decay coefficient that gives the best classification accuracy on the test set from $\lambda = \{0., 0.0001, 0.001, 0.01, 0.1, 1.\}$. Plot λ vs the test set accuracy. Comment on your plot and the effect of weight-decay regularization on the test performance. Also comment on why we need to tune the hyper-parameter λ using a validation set instead of the training set. [2 pt.]

Partial preview of Assignment 2

This question is not part of Assignment 1; please do not submit your answer yet.

2.3 Cross-entropy loss [10 pt.]

The MSE loss function works well for typical regression tasks in which the model outputs are real values. Despite its simplicity, MSE can be overly sensitive to mislabelled training examples and to outliers. A more suitable loss function for the classification task is the cross-entropy loss, which compares the log-odds of the data belonging to either of the classes. Because we only care about the probability of a data point belonging to one class, the real-valued linear prediction is first fed into a sigmoid function $\sigma(z) = \frac{1}{1+e^{-z}}$ that "squashes" the real-valued output to fall between zero and one. The model output is therefore $\hat{y}(\mathbf{x}) = \sigma(W^T\mathbf{x} + b)$. The cross-entropy loss is defined as:

$$\begin{aligned}\mathcal{L} =& \mathcal{L}_{\mathcal{D}} + \mathcal{L}_W \\ =& \sum_{m=1}^{M} \frac{1}{M} \Big[- y^{(m)} \log \hat{y}(\mathbf{x}^{(m)}) - (1 - y^{(m)}) \log(1 - \hat{y}(\mathbf{x}^{(m)})) \Big] + \frac{\lambda}{2} \|W\|_2^2\end{aligned}$$

The sigmoid function is often called the logistic function and hence a linear model with the cross-entropy loss is named "logistic regression".

1. **Learning**: Write a Tensorflow script that implements logistic regression and the cross-entropy loss. Train the logistic regression model using SGD and a mini-batch size $B = 50$ on the tiny MNIST dataset. Plot the best training and testing curves after tuning the learning rate and the weight-decay coefficient. Report the best test classification accuracy obtained from the logistic regression model compared to linear regression. What do you observe about the effect of the cross-entropy loss on the classification performance? [6 pt.]

   You may plot the cross-entropy loss vs squared error loss as a function of prediction $\hat{y}$ within the interval [0, 1] and a dummy target $y = 0$ in 1-D to help explain the observation.

2. **Maximum likelihood estimation**: In lecture we derived, from an i.i.d. data set, the MLE parameters of a Gaussian whose mean is parameterized by a linear function using weights W and b. It turned out that the MLE method was equivalent to minimizing the squared ℓ_2 loss from linear regression. Similarly, the cross-entropy loss can be derived from the MLE principle by assuming a Bernoulli distribution for the likelihood of the training labels $y \in \{0, 1\}$, i.e. $P(y = 1|\mathbf{x}, W) = 1 - P(y = 0|\mathbf{x}, W) = \hat{y}(\mathbf{x})$. We can write the Bernoulli distribution parameterized by W more concisely as $P(y|\mathbf{x}, W) = \hat{y}(\mathbf{x})^y(1 - \hat{y}(\mathbf{x}))^{(1-y)}$. Show that minimizing the cross-entropy loss is equivalent to maximizing the log-likelihood of the training data under the Bernoulli assumption. [4 pt.]