Homework 1

Deadline: Wednesday, Sept. 30, at 11:59pm.

Submission: You need to submit three files through MarkUs[1]:

- Your answers to Questions 1, 2, 3, and 4, and outputs requested for Question 1 and 4, as a PDF file titled `hw1_writeup.pdf`. You can produce the file however you like (e.g. LaTeX, Microsoft Word, scanner), as long as it is readable.
- Your code for Question 1, as the Python file `hw1_q1_code.py`. This should contain the functions `load_data` and `select_knn_model`.
- Your code for Question 4, as the Python file `hw1_q4_code.py`. This should contain the all functions such as `split_data` and `train_model`.

Neatness Point: One point will be given for neatness. You will receive this point as long as we don't have a hard time reading your solutions or understanding the structure of your code.

Late Submission: 10% of the marks will be deducted for each day late, up to a maximum of 3 days. After that, no submissions will be accepted.

Computing: To install Python and required libraries, see the instructions on the course web page.

Homeworks are individual work. See the Course Information handout[2] for detailed policies.

1. **[9pts] Classification with Nearest Neighbours.** In this question, you will use the `scikit-learn`'s KNN classifer to classify real vs. fake news headlines. The aim of this question is for you to read the `scikit-learn` API and get comfortable with training/validation splits.

   We will use a dataset of 1298 "fake news" headlines (which mostly include headlines of articles classified as biased, etc.) and 1968 "real" news headlines, where the "fake news" headlines are from https://www.kaggle.com/mrisdal/fake-news/data and "real news" headlines are from https://www.kaggle.com/therohk/million-headlines. The data were cleaned by removing words from titles not part of the headlines, removing special characters and restricting real news headlines after October 2016 using the word "trump". The cleaned data are available as `clean_real.txt` and `clean_fake.txt` in `hw1_data.zip` on the course webpage. It is expected that you use these cleaned data sources for this assignment.

   You will build a KNN classifier to classify real vs. fake news headlines. Instead of coding the KNN yourself, you will do what we normally do in practice — use an existing implementation. You should use the `KNeighborsClassifier` included in `sklearn`. Note that figuring out how to use this implementation, its corresponding attributes and methods is a part of the assignment.

   (a) **[2pts]** Write a function `load_data` which loads the data, preprocesses it using a `CountVectorizer` (http://scikit-learn.org/stable/modules/classes.html#module-sklearn.feature_extraction.text), and splits the entire dataset randomly into 70% training, 15% validation, and 15% test examples.

[1] The MarkUs URL will be announced soon, once the enrolled students are given access.
[2] http://www.cs.toronto.edu/~rgrosse/courses/csc311_f20/syllabus.pdf

(b) **[5pts]** Write a function `select_knn_model` that uses a KNN classifer to classify between real vs. fake news. Use a range of k values between 1 to 20 and compute both training and validation errors, leaving other arguments to `KNeighborsClassifier` at their default values. Generate a plot showing the training and validation accuracy for each k. Report the generated plot in your write-up. Choose the model with the best validation accuracy and report its accuracy on the test data.

(c) **[2pts]** Repeat part (b), passing argument `metric='cosine'` to the `KNeighborsClassifier`. You should observe an improvement in accuracy. How does `metric='cosine'` compute the distance between data points, and why might this perform better than the Euclidean metric (default) here? Hint: consider the dataset ['cat', 'bulldozer', 'cat cat cat'].

2. **[8pts] Regularized Linear Regression.** For this problem, we will use the linear regression model from the lecture:

$$y = \sum_{j=1}^{D} w_j x_j + b.$$

In lecture, we saw that regression models with too much capacity can overfit the training data and fail to generalize. We also saw that one way to improve generalization is regularization: adding a term to the cost function which favors some explanations over others. For instance, we might prefer that weights not grow too large in magnitude. We can encourage them to stay small by adding a penalty:

$$\mathcal{R}(\mathbf{w}) = \frac{\lambda}{2}\mathbf{w}^\top \mathbf{w} = \frac{\lambda}{2}\sum_{j=1}^{D} w_j^2$$

to the cost function, for some $\lambda > 0$. It is also possible to apply different regularization penalties in each dimension. The formulation would be:

$$\mathcal{J}_{\text{reg}}^{\beta}(\mathbf{w}) = \underbrace{\frac{1}{2N}\sum_{i=1}^{N}\left(y^{(i)} - t^{(i)}\right)^2}_{=\mathcal{J}} + \underbrace{\frac{1}{2}\sum_{j=1}^{D}\beta_j w_j^2}_{=\mathcal{R}},$$

where i indexes the data points, $\beta_j \geq 0$ for all j, and $\mathcal{J}$ is the same squared error cost function from lecture. Note that in this formulation, *there is no regularization penalty on the bias parameter.* Also note that when $\beta_j = 0$, you don't apply any regularization on j-th dimension. For this question, show your work in detail as most points are allocated in showing how you obtained your answer.

(a) **[3pts]** Determine the gradient descent update rules for the regularized cost function $\mathcal{J}_{\text{reg}}^{\beta}$. Your answer should have the form:

$$w_j \leftarrow \cdots$$
$$b \leftarrow \cdots$$

This form of regularization is sometimes called "weight decay". Based on this update rule, why do you suppose that is?

(b) **[3pts]** It's also possible to solve the regularized regression problem directly by setting the partial derivatives equal to zero. In this part, for simplicity, *we will drop the bias term from the model*, so our model is:

$$y = \sum_{j=1}^{D} w_j x_j.$$

It is possible to derive a system of linear equations of the following form for $\mathcal{J}_{\text{reg}}^{\beta}$:

$$\frac{\partial \mathcal{J}_{\text{reg}}^{\beta}}{\partial \mathbf{w}_j} = \sum_{j'=1}^{D} \mathbf{A}_{jj'} \mathbf{w}_{j'} - \mathbf{c}_j = 0.$$

Determine formulas for $\mathbf{A}_{jj'}$ and $\mathbf{c}_j$.

(c) **[2pts]** Based on your answer to part (b), determine formulas for $\mathbf{A}$ and $\mathbf{c}$, and derive a closed-form solution for the parameter $\mathbf{w}$. Note that, as usual, the inputs are organized into a design matrix $\mathbf{X}$ with one row per training example.

3. **[4pts] Loss Functions.** Suppose we have a prediction problem where the target t corresponds to an angle, measure in radians. A reasonable loss function we might use is:

$$\mathcal{L}(y, t) = 1 - \cos(y - t).$$

Suppose we make predictions using a linear model:

$$y = \mathbf{w}^\top \mathbf{x} + b.$$

As usual, the cost is the average loss over the training set:

$$\mathcal{J} = \frac{1}{N} \sum_{i=1}^{N} \mathcal{L}(y^{(i)}, t^{(i)}).$$

Derive a sequence of vectorized mathematical expressions for the gradients of the cost with respect to $\mathbf{w}$ and b. Recall that the inputs are organized into a design matrix $\mathbf{X}$ with one row per training example. The expressions should be something you can translate into a Python program without requiring a `for`-loop. Your answer should look like:

$$\begin{aligned} \mathbf{y} &= \cdots \\ \frac{\partial \mathcal{J}}{\partial \mathbf{y}} &= \cdots \\ \frac{\partial \mathcal{J}}{\partial \mathbf{w}} &= \cdots \\ \frac{\partial \mathcal{J}}{\partial b} &= \cdots \end{aligned}$$

You can use $\sin(\mathbf{A})$ to denote the sin function applied elementwise to $\mathbf{A}$. Remember that $\partial \mathcal{J} / \partial \mathbf{w}$ denotes the gradient vector,

$$\frac{\partial \mathcal{J}}{\partial \mathbf{w}} = \begin{pmatrix} \frac{\partial \mathcal{J}}{\partial w_1} \\ \vdots \\ \frac{\partial \mathcal{J}}{\partial w_D} \end{pmatrix}$$

4. **[9pts] Cross Validation.** In this problem, you will write a function that performs K-fold cross validation (CV) procedure to tune the penalty parameter λ in Ridge regression. CV procedure is one of the most commonly used methods for tuning hyperparameters. *In this question, you shouldn't use the package* `scikit-learn` to perform CV. *You should implement all of the below functions yourself. You may use* `numpy` *and* `scipy` *for basic math operations such as linear algebra, sampling etc.*

   In lecture, we learned *training*, *test*, and *validation* procedures which assumes that you have enough data and you can set aside a validation set and a test set to use it for assessing the performance of your machine learning algorithm. However in practice, this may be problematic since we may not have enough data. A remedy to this issue is K-fold cross validation which uses a part of the available data to fit the model, and a different part to test it. K-fold CV procedure splits the data into K equal-sized parts; for example, when $K = 5$, the scenario looks like this:

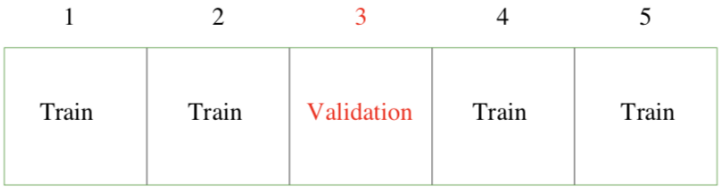

Figure 1: credit: Elements of Statistical Learning

   (i) We first set aside a test dataset and never use it until the training and parameter tuning procedures are complete. We will use this data for final evaluation. In this question, test data is provided to you as a separate dataset.

   (ii) CV error estimates the test error of a particular hyperparameter choice. For a particular hyperparameter value, we split the training data into K blocks (See the figure), and for $k = 1, 2, ..., K$ we use the k-th block for validation and the remaining $K-1$ blocks are for training. Therefore, we train and validate our algorithm K times. Our CV estimate for the test error for that particular hyperparameter choice is given by the average validation error across these K blocks.

   (iii) We repeat the above procedure for several hyperparameter choices and choose the one that provides us with the smalles CV error (which is an estimate for the test error).

   Below, we will code the above procedure for tuning the regularization parameter in linear regression which is a hyperparameter. Your `cross_validation` function will rely on 6 short functions which are defined below along with their variables.

   - `data` is a variable and refers to a $(\mathbf{t}, \mathbf{X})$ pair (can be test, training, or validation) where $\mathbf{t}$ is the target (response) vector, and $\mathbf{X}$ is the feature matrix.
   - `model` is a variable and refers to the coefficients of the trained model, i.e. $\mathbf{w}$.
   - `data_shf = shuffle_data(data)` is a function and takes `data` as an argument and returns its randomly permuted version along the samples. Here, we are considering a uniformly random permutation of the training data. Note that $\mathbf{t}$ and $\mathbf{X}$ need to be permuted the same way preserving the target-feature pairs.

- `data_fold, data_rest = split_data(data, num_folds, fold)` is a function that takes `data`, number of partitions as `num_folds` and the selected partition `fold` as its arguments and returns the selected partition (block) `fold` as `data_fold`, and the remaining data as `data_rest`. If we consider 5-fold cross validation, `num_folds`=5, and your function splits the data into 5 blocks and returns the block `fold` ($\in \{1, 2, 3, 4, 5\}$) as the validation fold and the remaining 4 blocks as `data_rest`. Note that `data_rest` $\cup$ `data_fold` = `data`, and `data_rest` $\cap$ `data_fold` $= \emptyset$.
- `model = train_model(data, lambd)` is a function that takes `data` and `lambd` as its arguments, and returns the coefficients of ridge regression with penalty level λ. *For simplicity, you may ignore bias parameter for this question.* You may use your solution from question 3.
- `predictions = predict(data, model)` is a function that takes `data` and `model` as its arguments, and returns the `predictions` based on `data` and `model`.
- `error = loss(data, model)` is a function which takes `data` and `model` as its arguments and returns the average squared `error` loss based on `model`. This means if `data` is composed of $\mathbf{t} \in \mathbb{R}^N$ and $\mathbf{X} \in \mathbb{R}^{N \times D}$, and model is $\mathbf{w}$, then the return value is $\|\mathbf{X}\mathbf{w} - \mathbf{t}\|^2/2N$.
- `cv_error = cross_validation(data, num_folds, lambd_seq)` is a function that takes the training `data`, number of folds `num_folds`, and a sequence of λ's as `lambd_seq` as its arguments and returns the cross validation error across all λ's. Take `lambd_seq` as evenly spaced 50 numbers over the interval (0.00005, 0.005). This means `cv_error` will be a vector of 50 errors corresponding to the values of `lambd_seq`. Your function will look like:

```
data = shuffle_data(data)
for i = 1,2,...,length(lambd_seq)
  lambd = lambd_seq(i)
  cv_loss_lmd = 0.
  for fold = 1,2, ...,num_folds
    val_cv, train_cv = split_data(data, num_folds, fold)
    model = train_model(train_cv, lambd)
    cv_loss_lmd += loss(val_cv, model)
  cv_error(i) = cv_loss_lmd / num_folds
return cv_error
```

(a) **[0pts]** Download the dataset from the course webpage `hw1_data.zip` and place and extract in your working directory, or note its location `file_path`. For example, file path could be `/Users/yourname/Desktop/`

- In `Python`:

```
import numpy as np
data_train = {'X': np.genfromtxt('data_train_X.csv', delimiter=','),
              't': np.genfromtxt('data_train_y.csv', delimiter=',')}
data_test = {'X': np.genfromtxt('data_test_X.csv', delimiter=','),
             't': np.genfromtxt('data_test_y.csv', delimiter=',')}
```

(b) **[5pts]** Write the above 6 functions, and identify the correct order and arguments to do cross validation.

(c) **[2pts]** Report the training and test errors corresponding to each λ in `lambd_seq`. This part does not use the `cross_validation` function but you may find the other functions helpful.

(d) **[2pts]** Plot training error, test error, and 5-fold and 10-fold cross validation errors on the same plot for each value in `lambd_seq`. What is the value of λ proposed by your cross validation procedure? Comment on the shapes of the error curves.

We have made the following question optional, because some students are taking probability theory concurrently this year and don't yet have the background to solve this question. However, you may find it is valuable to try to solve it.

5. **[Optional] Nearest Neighbours and the Curse of Dimensionality.** In this question, you will verify the claim from lecture that "most" points in a high-dimensional space are far away from each other, and also approximately the same distance. There is a very neat proof of this fact which uses the properties of expectation and variance. You may wish to review the Tutorial 1 slides[3], or the Metacademy resources[4].

   (a) **[0pts]** Consider two independent univariate random variables X and Y sampled uniformly from the unit interval $[0, 1]$. Determine the expectation and variance of the random variable Z, defined as the squared distance $Z = (X - Y)^2$.

   (b) **[0pts]** Now suppose we sample two points independently from a unit cube in d dimensions. Observe that each coordinate is sampled independently from $[0, 1]$, i.e. we can view this as sampling random variables $X_1, \ldots, X_d, Y_1, \ldots, Y_d$ independently from $[0, 1]$. The squared Euclidean distance can be written as $R = Z_1 + \cdots + Z_d$, where $Z_i = (X_i - Y_i)^2$. Using the properties of expectation and variance, determine $\mathbb{E}[R]$ and $\mathrm{Var}[R]$. You may give your answer in terms of the dimension d, and $\mathbb{E}[Z]$ and $\mathrm{Var}[Z]$ (the answers from part (a)).

   (c) **[0pts]** Based on your answer to part (b), compare the mean and standard deviation of R to the maximum possible squared Euclidean distance (i.e. the distance between opposite corners of the cube). Why does this support the claim that in high dimensions, "most points are far away, and approximately the same distance"?

[3] http://www.cs.toronto.edu/~rgrosse/courses/csc311_f20/tutorials/tut01/tut01.pdf
[4] https://metacademy.org/graphs/concepts/expectation_and_variance