Homework 1 Solution

1. [**7pts**] **Classification with Nearest Neighbours.**

   (a) [**2pts**] Write a function `load_data` which loads the data, preprocesses it using a vectorizer, and splits the entire dataset randomly into 70% training, 15% validation, and 15% test examples.

   solution.

```
def load_data():
        with open('data/clean_fake.txt', 'r') as f:
                fake = [l.strip() for l in f.readlines()]
        with open('data/clean_real.txt', 'r') as f:
                real = [l.strip() for l in f.readlines()]

        data = np.array(real + fake)
        labels = np.array([0]*len(real) + [1]*len(fake))

        # Train, validation, test split (70,15,15)
        # train_test_split shuffles the data before splitting it
        # Stratify keeps the proportion of labels the same in each split
        xtr, xte, ytr, yte = train_test_split(data, labels, test_size=0.15,
             stratify=labels)
        xtr, xval, ytr, yval = train_test_split(xtr, ytr, test_size
            =0.15/0.85, stratify=ytr)

        # Note that we fit the Vectorizer using the training set only!
        count_vect = CountVectorizer()
        xtr = count_vect.fit_transform(xtr)
        xval = count_vect.transform(xval)
        xte = count_vect.transform(xte)

        # sort and return term vocabulary used to create the features
        kw_to_ind = count_vect.vocabulary_
        term_vocab2 = np.array([[y,x] for x,y in kw_to_ind.items()])
        term_vocab_srt = np.sort(term_vocab2,axis=0)
        ind_to_kw = term_vocab_srt[:,1]

        return [xtr, ytr], [xval, yval], [xte, yte], ind_to_kw, kw_to_ind
```

   - [**-1**] Incorrect logistics (e.g. CountVectorizer applied to the entire dataset, does not return the necessary outputs for next steps).
   - [**-2**] Incorrect code.

   (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. 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.

   solution.

```
def select_knn_model(train_X, val_X, train_y, val_y, k):
    neigh = KNeighborsClassifier(n_neighbors=k)
    neigh.fit(train_X, train_y)

    train_accuracy = neigh.score(train_X, train_y)
    val_accuracy = neigh.score(val_X, val_y)
    return train_accuracy , val_accuracy
```

Then, we generate a plot showing the training and validation accuracy for each k.

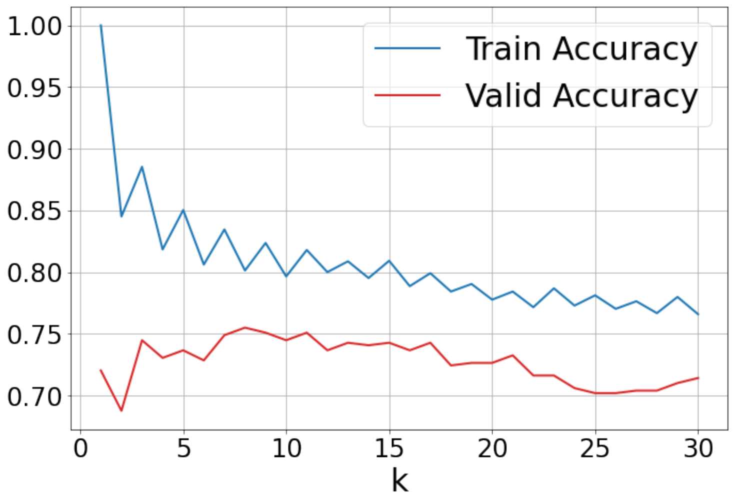

Figure 1: Training and validation accuracy for each k

The model had the best validation accuracy when $k = 3$ and its resulting test accuracy was 73.87%.

- **[-2]** Incorrect code for writing a function.
- **[-2]** Incorrect figures.
- **[-1]** Missing best k and its accuracy.

(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'].

solution. Repeating part (b) with `metric='cosine'`, the model had best validation accuracy when $k = 8$ and its resulting test accuracy was 77.75%. With Euclidean, [1 0] is closer to [0 1] than it is to [3 0], which builds a wrong inductive bias. With cosine distance, however, [1 0] is closer to [3 0].

- **[-0.5]** Not reporting k and its accuracy.
- **[-1.5]** Incorrect explanation.

2. **[8pts] Regularized Linear Regression.**

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

solution.
The gradient descent update rules for the regularized cost function $\mathcal{J}_{\text{reg}}^{\beta}$ will be of the form:

$$\begin{aligned} w_j &\leftarrow w_j - \alpha \frac{\partial \mathcal{J}_{\text{reg}}^{\beta}}{\partial w_j} \\ b &\leftarrow b - \alpha \frac{\partial \mathcal{J}_{\text{reg}}^{\beta}}{\partial b}, \end{aligned}$$

where α is the learning rate. Now for the weights w_j, we have:

$$\begin{aligned} \frac{\partial \mathcal{J}_{\text{reg}}^{\beta}}{\partial w_j} &= \frac{\partial \mathcal{J}}{\partial w_j} + \frac{\partial \mathcal{R}}{\partial w_j} \\ &= \frac{1}{N}\sum_{i=1}^{N} x_j(y^{(i)} - t^{(i)}) + \frac{\partial}{\partial w_j}\left(\frac{1}{2}\sum_j \beta_j w_j^2\right) \\ &= \frac{1}{N}\sum_{i=1}^{N} x_j(y^{(i)} - t^{(i)}) + \beta_j w_j \end{aligned}$$

For the bias term, we have:

$$\begin{aligned} \frac{\partial \mathcal{J}_{\text{reg}}^{\beta}}{\partial b} &= \frac{\partial \mathcal{J}}{\partial b} + \frac{\partial \mathcal{R}}{\partial b} \\ &= \frac{1}{N}\sum_{i=1}^{N} (y^{(i)} - t^{(i)}) + \frac{\partial}{\partial b}\left(\frac{1}{2}\sum_j \beta w_j^2\right) \\ &= \frac{1}{N}\sum_{i=1}^{N} (y^{(i)} - t^{(i)}) + 0, \end{aligned}$$

which is equivalent to the partial derivative without regularization. Thus, we have:

$$\begin{aligned} w_j &\leftarrow w_j - \alpha \frac{\partial \mathcal{J}}{\partial w_j} - \alpha \beta_j w_j \\ &= (1 - \alpha\beta_j) w_j - \alpha \frac{\partial \mathcal{J}}{\partial w_j} \\ b &\leftarrow b - \alpha \frac{\partial \mathcal{J}}{\partial b} \end{aligned}$$

This form of regularization is sometimes called "weight decay". At each step of gradient descent, weights are pulled towards zero. The further the weights are towards zero, the stronger the force of this "pull". The constant λ controls the strength of the pull.

- **[-1]** Mistakes in subscripts (e.g. i or j).
- **[-0.5]** Forget to propagate the $1/N$ term in the derivation.
- **[-0.5]** Calculus errors when calculating the gradients.
- **[-0.5]** Wrong description of "weight decay".

(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 $\mathbf{c}_j$.

solution.

Observe that:

$$\begin{aligned}\frac{\partial \mathcal{J}_{\text{reg}}^{\beta}}{\partial w_j} &= \frac{\partial \mathcal{J}}{\partial w_j} + \beta_j w_j \\ &= \frac{1}{N} \sum_{j'=1}^{D} \left(\sum_{i=1}^{N} x_j^{(i)} x_{j'}^{(i)} \right) w_{j'} - \frac{1}{N} \sum_{i=1}^{N} x_j^{(i)} t^{(i)} + \beta_j w_j \\ &= \sum_{j'=1}^{D} \left[\left(\frac{1}{N} \sum_{i=1}^{N} x_j^{(i)} x_{j'}^{(i)} \right) + \delta_{j,j'} \beta_j \right] w_{j'} - \frac{1}{N} \sum_{i=1}^{N} x_j^{(i)} t^{(i)},\end{aligned}$$

where

$$\delta_{j,j'} = \begin{cases} 1, & \text{if } j = j' \\ 0, & \text{otherwise} \end{cases}$$

Now, the values of $\mathbf{A}_{jj'}$ and $\mathbf{c}_j$ are:

$$\begin{aligned}\mathbf{A}_{jj'} &= \frac{1}{N} \left(\sum_{i=1}^{N} x_j^{(i)} x_{j'}^{(i)} \right) + \delta_{j,j'} \beta_j \\ \mathbf{c}_j &= \frac{1}{N} \sum_{i=1}^{N} x_j^{(i)} t^{(i)}\end{aligned}$$

- **[-0.5]** Minor calculus mistakes.
- **[-1]** Major mistake in final solution.
- **[-0.5]** Not factoring $\beta_j w_j$ in $A_{jj'}$.

(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.

solution.

Based on the previous solution, we have:

$$\begin{aligned}\mathbf{A} &= \frac{1}{N}(\mathbf{X}^{\top}\mathbf{X} + \lambda N \text{diag}(\boldsymbol{\beta})) \\ \mathbf{c} &= \frac{1}{N}\mathbf{X}^{\top}\mathbf{t},\end{aligned}$$

where $\boldsymbol{\beta}$ is a D-dimensional row vector with regularization weight in each dimension and $\text{diag}(\boldsymbol{\beta})$ maps it to a matrix where the diagonal entries contain corresponding element from $\boldsymbol{\beta}$. Then:

$$\frac{\partial \mathcal{J}_{\text{reg}}^{\beta}}{\partial \mathbf{w}} = \frac{1}{N}(\mathbf{X}^\top \mathbf{X} + \lambda N \text{diag}(\boldsymbol{\beta}))\mathbf{w} - \frac{1}{N}(\mathbf{X}^\top \mathbf{t})$$

Setting it equal to $\mathbf{0}$, we obtain the optimal solution:

$$\mathbf{w} = (\mathbf{X}^\top \mathbf{X} + \lambda N \text{diag}(\boldsymbol{\beta}))^{-1}\mathbf{X}^\top \mathbf{t}$$

- **[-1]** Incorrect matrix form.
- **[-0.5]** Minor mistake in final answer.
- **[-1]** Major mistake in final answer

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,

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

solution.
Let N denote the number of training examples, or the number of columns of the design matrix $\mathbf{X}$. Then, we have:

$$\begin{aligned}\mathbf{y} &= \mathbf{X}\mathbf{w} + b\mathbf{1}\\ \frac{\partial \mathcal{J}}{\partial \mathbf{y}} &= \frac{1}{N}\sin(\mathbf{y}-\mathbf{t})\\ \frac{\partial \mathcal{J}}{\partial \mathbf{w}} &= \mathbf{X}^\top \frac{\partial \mathcal{J}}{\partial \mathbf{y}}\\ \frac{\partial \mathcal{J}}{\partial b} &= \mathbf{1}^\top \frac{\partial \mathcal{J}}{\partial \mathbf{y}},\end{aligned}$$

where $\mathbf{1}$ is a row vector of 1's with dimension N. (Full credit is also given if you expanded out the formulas explicitly)

- **[-1]** Each incorrect final answer.

4. **[9pts] Cross Validation.**

   (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`.

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

   solution.

```
def shuffle_data(data):
    return np.random.permutation(data)

def split_data(data, num_folds, fold):
    if fold > num_folds or fold < 1 or num_folds > len(data)/2 or num_folds
        < 2 or num_folds != int(num_folds) or fold != int(fold):
        raise RuntimeError("Fold index invalid.")

    fold_length = len(data) // num_folds + (len(data) % num_folds > 0)
    data_fold_ids = np.arange((fold - 1)* fold_length, \
        min(int(fold * fold_length), len(data)))
    return data[data_fold_ids,], data[[i for i in range(len(data)) if i not
        in data_fold_ids],]

def train_model(data, lambd):
    y_train = data[:,0].reshape(-1, 1)
    x_train = data[:,1:]
    N = len(data)

    return np.linalg.solve(np.dot(np.transpose(x_train), x_train) + lambd *
        len(x_train[0]) * np.identity(len(x_train[0])), \
        np.dot(np.transpose(x_train), y_train))

def predict(data, model):
    return np.dot(data[:,1:], model)

def loss(data, model):
    error_dif = predict(data, model) - data[:,0].reshape(-1,1)
    return np.asscalar(np.dot(np.transpose(error_dif), error_dif) / (2 *
       len(data)))

def cross_validation(data, num_folds, lambd_seq):
    data = shuffle_data(data)
    cv_error = []

    for lambd in lambd_seq:
        cv_loss = 0
        for fold in range(1, num_folds + 1):
            validation_data, train_data = split_data(data, num_folds, fold)
            model = train_model(train_data, lambd)
            cv_loss += loss(validation_data, model)
        cv_error.append(cv_loss / num_folds)
    return cv_error

def train_test_error(data, test_data, lambd_seq):
    train_error = []
    test_error = []

    for lambd in lambd_seq:
        model = train_model(data, lambd)
        train_error.append(loss(data, model))
        test_error.append(loss(test_data, model))
    return train_error, test_error

data_train = {"X": np.genfromtxt("../data/data_train_X.csv", delimiter=",")
    ,
            "t": np.genfromtxt("../data/data_train_y.csv", delimiter=",")}
data_test = {"X": np.genfromtxt("../data/data_test_X.csv", delimiter=","),
            "t": np.genfromtxt("../data/data_test_y.csv", delimiter=",")}

train_X, train_y = data_train["X"], data_train["t"]
test_X, test_y = data_test["X"], data_test["t"]

np.random.seed(1)

lambdas = np.linspace(0.00005, 0.005, num=50)
cross5_error = cross_validation(train_data, 5, lambdas)
cross10_error = cross_validation(train_data, 10, lambdas)
train_error, test_error = train_test_error(train_data, test_data, lambdas)

#find minimum error
print(lambdas[np.argmin(cross5_error)])
print(lambdas[np.argmin(cross10_error)])

#create plot
plt.plot(lambdas, cross5_error, 'b--', label = '5-fold error')
plt.plot(lambdas, cross10_error, 'k--', label = '10-fold error')
plt.plot(lambdas, train_error, 'b-', label = 'Training error')
plt.plot(lambdas, test_error, 'k-', label = 'Testing error')
plt.legend(loc='upper right')

plt.xlabel(r'$\lambda$')
plt.ylabel('Error')
plt.title(r'Observed errors vs. $\lambda \; L^2$ regulator')
plt.show()
```

- **[-1]** Each incorrect implementation of the function.

(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.

solution. See below.

- **[-1]** Missing training and test errors.
- **[-1]** Incorrect plots / tables / values.

(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.

solution.

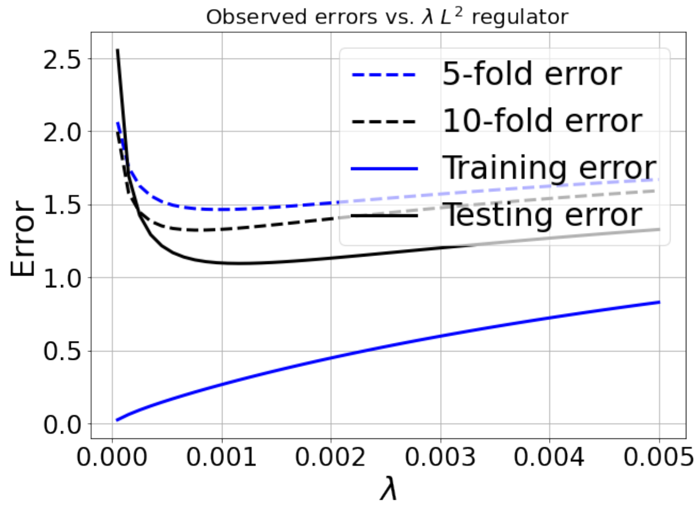

Figure 2: Relationship between average squared-error and ridge regularizing parameter

The proposed λs are 0.0012 and 0.00095 for 5-fold and 10-fold cross validation, respectively.

The training error increases as λ increases. This occurs because a higher value of λ results in a more constrictive constraint on parameter values, and therefore less flexibility to perfectly fit the training data. The testing error first decreases, because overfitting is prevented by the regularization procedure, but then increases because the constraint becomes overly tight. The shapes of the 5-fold and 10-fold cross validation errors are similar to the testing error for the same reasons.

The training error is lower than 5-fold, 10-fold, and testing error since the model is selected to best fit this data. The 5-fold, 10-fold, and testing errors may alternate in which is highest, depending on the properties of the data set and the random seed selected.

- **[-1]** Incorrect interval or incorrect plot.
- **[-0.5]** Not reporting λ.
- **[-0.5]** Incorrect comment on the shape of the curve.