Homework 1

1. Nearest neighbor predictor. We have a collection of \(n\) observations, \(x^i \in \mathbb{R}^d, y^i \in \mathbb{R}, i = 1, \ldots, n.\) Based on these observations, the nearest neighbor predictor is defined as \(g^{\text{nn}}(x) = y^k,\) where \(x^k\) is a nearest neighbor of \(x\) among the data points. (We can break ties arbitrarily. Recall that \(x^k\) is a nearest neighbor of \(x\) means that \(\|x - x^k\| \leq \|x - x^i\|\) for \(i = 1, \ldots, n.\))

(a) Write a Julia function \(y\_hat = \text{nn\_predictor}(x, X, y)\) that implements \(g^{\text{nn}},\) given the argument \(x.\) The second and third arguments give the data on which the predictor is based: \(X\) is an \(n \times d\) matrix whose \(i\)th row is \((x^i)^\text{T},\) and \(y\) is an \(n\)-vector with \(i\)th entry \(y^i.\)

(b) Report the train and test RMSE of the predictor in part (a) on the data provided in \(\text{nearest\_neighbor\_data.json},\) training on the first 1500 data points and testing on the rest. Briefly interpret your results.

(c) Approximately how many flops (floating point operations) are required to evaluate \(g^{\text{nn}}(x),\) in terms of \(n\) and \(d?\)

Solution.

(a) A sample function is implemented below

```julia
function nn_predictor(x, X, y)
    y_hat = nothing
    dist = Inf
    n, d = size(X)

    for i=1:n
        curr_dist = norm(x - X[i,:])
        if curr_dist < dist
            y_hat = y[i]
            dist = curr_dist
        end
    end

    return y_hat
end
```

(b) The training RMSE for the nearest neighbor predictor is zero, since the squared difference between a point and itself is zero (you should check this!). The RMSE error we got for the test set was around .142, with the code found below
include("nn_predictor.jl") # contains nn_predictor(x, X, y) function
include("readclassjson.jl")

all_data = readclassjson("nearest_neighbor_data.json")
X = all_data["X"]
y = all_data["y"]

N = 1500

X_train, y_train = X[1:N,:), y[1:N]
X_test, y_test = X[N+1:end,:), y[N+1:end]

nTest = length(y_test)
y_pred = []

for i=1:nTest
    x = X_test[i,:]
y = y_test[i]
y_hat = nn_predictor(x, X_train, y_train)
push!(y_pred, y_hat)
end

RMSE = norm(y_pred - y_test) / sqrt(nTest)
println("RMSE for NN $(RMSE)"

(c) Computing the distance between $x^i$ and $x$ takes roughly $d$ operations. Since there are $n$ points, it takes roughly $nd$ flops to compute $g_{nn}(x)$.

2. **Soft nearest neighbor.** In this exercise we examine an extension of the nearest neighbor predictor that can occasionally perform better. For data set $x^i \in \mathbb{R}^d$, $y^i \in \mathbb{R}$, $i = 1, \ldots, n$, the \textit{soft nearest neighbor predictor} is defined as

$$g_{snn}(x) = \frac{\sum_{i=1}^{n} y^i e^{-\|x-x^i\|^2/\sigma^2}}{\sum_{i=1}^{n} e^{-\|x-x^i\|^2/\sigma^2}},$$

where $\sigma \geq 0$ is a parameter. Note that $\sigma$ can be thought of as a distance; the function $e^{-\|x-x^i\|^2/\sigma^2}$ is near one when the distance between $x$ and $x^i$ is much less than $\sigma$, and it is very small when the distance is much more than $\sigma$.

(a) What does $g_{snn}(x)$ converge to as $\sigma \to 0$? (Explain briefly.)
(b) What does $g_{snn}(x)$ converge to as $\sigma \to \infty$? (Explain briefly.)
(c) Implement the soft nearest neighbor predictor in Julia as

\[ y_{\text{hat}} = \text{snn\_predictor}(x, X, y, \sigma). \]

(d) Using the data in nearest\_neighbor\_data.json, plot train and test RMSEs for \( g_{\text{snn}} \) as a function of \( \sigma \) over the range \([10^{-1}, 10^1]\). What value of \( \sigma \) would you choose? How does the test RMSE of predictor compare to the nearest neighbor predictor and the constant predictor?

Solution.

(a) We assume, for simplicity, that \( x \) has a unique nearest neighbor (say, \( x^k \)), as \( \sigma \to 0 \), \( g_{\text{snn}}(x) \) converges to the nearest neighbor’s label, \( y^k \). This is because all of the terms whose distance is larger than \( \|x - x^k\| \) become small in the numerator and denominator as \( \sigma \) tends to zero.

We can show this by multiplying the denominator and numerator of \( g_{\text{snn}}(x) \) by \( e^{\|x-x^k\|^2/\sigma^2} \). This gives

\[
g_{\text{snn}}(x) = \frac{\sum_{i=1}^{n} y_i e^{(\|x-x^k\|^2-\|x-x^i\|^2)/\sigma^2}}{\sum_{i=1}^{n} e^{(\|x-x^k\|^2-\|x-x^i\|^2)/\sigma^2}} = \frac{y_k + \sum_{i \neq k} y_i e^{(\|x-x^k\|^2-\|x-x^i\|^2)/\sigma^2}}{1 + \sum_{i \neq k} e^{(\|x-x^k\|^2-\|x-x^i\|^2)/\sigma^2}}.
\]

Since \( x^k \) is the nearest neighbor of \( x \) then we know that \( \|x - x^k\|^2 - \|x - x^i\|^2 < 0 \) whenever \( k \neq i \). Thus, taking \( \sigma \downarrow 0 \),

\[
e^{(\|x-x^k\|^2-\|x-x^i\|^2)/\sigma^2} \to 0, \ i \neq k,
\]

which implies

\[
g_{\text{snn}}(x) \to \frac{y_k + \sum_{i \neq k} y_i e^{(\|x-x^k\|^2-\|x-x^i\|^2)/\sigma^2}}{1 + \sum_{i \neq k} e^{(\|x-x^k\|^2-\|x-x^i\|^2)/\sigma^2}} \to \frac{y_k}{1} = y_k.
\]

(b) As \( \sigma \to \infty \), the predictor \( g_{\text{snn}}(x) \) converges to the mean of all labels \( y^1, \ldots, y^n \). This is because, as \( \sigma \to \infty \), we have

\[
e^{-\|x-x^i\|^2/\sigma^2} \to 1
\]

so

\[
g_{\text{snn}}(x) = \frac{\sum_{i=1}^{n} y_i e^{-\|x-x^i\|^2/\sigma^2}}{\sum_{i=1}^{n} e^{-\|x-x^i\|^2/\sigma^2}} \to \frac{\sum_{i=1}^{n} y_i}{n}
\]

(c) The code for an implementation of the predictor in Julia is below.

```julia
function snn_predictor(x, X, y, sigma)
    weights = exp.((-((sum((X - x') .^ 2, 2)) / sigma^2))
    return sum(y .* weights) / sum(weights)
end
```
(d) The test RMSE of the soft NN predictor we received, around .1013 with $\sigma \approx .263$, is lower than both the NN predictor with RMSE .143, and the constant predictor with RMSE .163.

Note that, as $\sigma$ gets closer to $10^1$, the RMSE of the soft NN predictor also gets close to that of the constant predictor, as you showed in part (b).

The plot for the train and test error is below.

Julia code for the problem is found below.

```julia
using PyPlot
include("snn_predictor.jl")
include("readclassjson.jl")

all_data = readclassjson("nearest_neighbor_data.json")
X = all_data["X"]
y = all_data["y"]

N = 1500

X_train = X[1:N,:]  
y_train = y[1:N]

X_test = X[N+1:end,:]  
y_test = y[N+1:end]
```
poss_sigma = logspace(-1, 1, 20)

best_rmse = Inf
best_sigma = nothing
all_rmse_test = zeros(length(poss_sigma))
all_rmse_train = zeros(length(poss_sigma))

for idx=1:length(poss_sigma)
    RMSE_snn_test = 0
    RMSE_snn_train = 0

    for i=1:size(X_train,1)
        x = X_train[i, :]
        y = y_train[i]

        y_hat_snn = snn_predictor(x, X_train, y_train, poss_sigma[idx])

        RMSE_snn_train += (y_hat_snn - y)^2
    end

    for i=1:size(X_test,1)
        x = X_test[i, :]
        y = y_test[i]

        y_hat_snn = snn_predictor(x, X_train, y_train, poss_sigma[idx])

        RMSE_snn_test += (y_hat_snn - y)^2
    end

    RMSE_snn_test = sqrt(RMSE_snn_test/size(X_test, 1))
    RMSE_snn_train = sqrt(RMSE_snn_train/size(X_train, 1))

    all_rmse_test[idx] = RMSE_snn_test
    all_rmse_train[idx] = RMSE_snn_train

    println("RMSE error $RMSE_snn_test with sigma = $(poss_sigma[idx])")

    if RMSE_snn_test < best_rmse
        best_rmse = RMSE_snn_test
        best_sigma = poss_sigma[idx]
    end
end
println("Best RMSE for SNN \$best_rmse with sigma = \$best_sigma")

const_RMSE = norm(y_test - mean(y_train))/sqrt(length(y_test))

semilogx(poss_sigma, all_rmse_train, label="Train")
semilogx(poss_sigma, all_rmse_test, label="Test")
xlabel("sigma")
ylabel("RMSE")
legend()
savefig("extend_nn_plot.pdf")
show()

println("Constant predictor RMSE is \$(const_RMSE)")

3. Selecting regressors to use. When considering different models, we want to consider which features give us the smallest test error. In selecting_regressors.json, you will find an \(n \times d\) matrix \(U\) of raw data, with rows \((u^T_i)\), with \(u^T_i \in \mathbb{R}^3\).

(a) Define \(S = \{s_1, \ldots, s_k\} \subseteq \{1, 2, 3\}\), and let \(\phi_S(u) = (1, u_{s_1}, \ldots u_{s_k})\). In words, \(\phi_S\) is the embedding that has a constant first feature, and the other features are the components of \(u\) with indices in \(S\). For example, if \(S = \{1, 3\}\), then \(\phi_S(u) = (1, u_1, u_3)\). Implement the following function,

\[
X = \text{embed}(S, U),
\]

which returns a data matrix \(X\), with rows \((x^T_i) = (\phi_S(u^T_i))^T\). The size of \(X\) is \(n \times (|S| + 1)\), where \(|S|\) denotes the number of elements in \(S\).

(b) With the data selecting_regressors.json and the code in selecting_regressors.jl, use the embedding function you implemented in part (a) to find parameter value \(\theta_S\) which minimizes the training set RMSE, for each \(S \subseteq \{1, 2, 3\}\). For each \(S\), compute and report the test RMSE for the associated predictor. Which are the two best embeddings? \textit{Hint:} Use the provided function \texttt{powerset()} to compute all the subsets of \(\{1, 2, 3\}\).

Solution.

(a) The following code solves the problem.

```julia
function embed(S, U)
    n, d = size(U)
    return [ones(n) U[:, S]]
end
```

(b) The following code imports the data.
The following code iterates through all subsets $S \subseteq \{1, 2, 3\}$, and prints out the test RMSE, using a 70-30 train-test split.

```julia
# Function to compute empirical loss
RMSE(y_hat, y) = norm(y - y_hat)/sqrt(length(y_hat))

# specify a train-test split of 70:30
n, d = size(U)
n_train = round(Int, 0.7*n)

# split (X, y) into (X_train, y_train) and (X_test, y_test)
ind = randperm(n)
U, y = U[ind, :], y[ind, :]
U_train, U_test = U[1:n_train, :], U[(n_train + 1):end, :]
y_train, y_test = y[1:n_train], y[(n_train + 1):end]

# evaluate test RMSE for each subset of \{1, 2, 3\}
for S in powerset(1:3)
    X_train, X_test = embed(S, U_train), embed(S, U_test)
    theta_S = X_train \ y_train
    y_hat = X_test * theta_S
    loss = RMSE(y_hat, y_test)
    println("S = \$S \t Test RMSE: \$loss")
end
```

The output of the code on the data we provided is

- $S: \{1, 3\}$, \text{RMSE: 0.5414863992191975.}
- $S: \{1, 2, 3\}$, \text{RMSE: 0.5416395434027858.}
- $S: \{1\}$, \text{RMSE: 0.5419517147822033.}
- $S: \{1, 2\}$, \text{RMSE: 0.5421080109776879.}
- $S: \{3\}$, \text{RMSE: 0.5434643804905595.}
- $S: \{2, 3\}$, \text{RMSE: 0.5436170353504275.}
By inspection, we see that best choices of $S$, i.e., those with the lowest test RMSE, are $S = \{1, 3\}$ and then $S = \{1, 2, 3\}$. 

$S: \text{Int64[]}$, \quad \text{RMSE: 0.5439862655355976.}$

$S: \{2\}$, \quad \text{RMSE: 0.5441419055417532.}$