CMDA-4654 Project 1

See Canvas for Due Date

Instructions:

- You are required to work in pairs, although I reserve the right to make re-assignments if I feel there are issues with your pairings. Both your names should be included on the report and `.R` file.
- You are required to use R for this project. Sorry Python users, it's just easier this way for this project.
- You must submit a written report. Failure for your report to compile results in a zero. Your report must compile directly to a `.pdf` file, that is you are not permitted to compile to a `.html` file first. You will name this file `Lastname1_Lastname2_project1.pdf`
- You must additionally submit a separate `.R` file containing only your names and the functions you wrote for this project, name it `Lastname1_Lastname2_project1.R`.

Part 1 - LOESS/LOWESS Regression

Overview

LOESS (aka LOWESS) regression is a non-parametric regression method that uses locally weighted regression to fit a smooth curve through points in a scatterplot. The main idea is that we can control the amount of "wiggliness" of our regression by fitting polynomials (usually lines or parabolas) on small windows of the data as opposed to the entire data at one time. This is a smoothing procedure that aims to capture a more quickly changing relationship between two variables while simultaneously being cautious to not over-fit the noise within the data.

Essentially we can control a LOESS regression in following ways:

1. The user determines the "bandwidth" or "smoothing parameter" that determines how much of the data is used to fit in each local polynomial.

We control the bandwidth or size of the windows in the R function `loess()` by using the `span` option. In this instance, `span` is a proportion that determines the number of points to use in each window in the following manner. If `N` is the total number of data points, and `span = 0.5`, then the `loess()` regression procedure will use the `0.5 * N` closest points to x for the local regression fit.

Hence we see that the subsets of data used for each weighted least squares fit in LOESS is determined by a nearest neighbors algorithm.

2. The degree of the local polynomial. We almost always use a first or second degree polynomial (aka line or parabola) in LOESS regression. We generally assume that any function can be well approximated in a small neighborhood by a low-order polynomial. Using a high-degree polynomial tends to overfit the data.
3. The weight function. As we recall from weighted least-squares, a weight function can re-assign weights in a regression so that we can give more weight to data points closer to the point of estimation and less weight to data points further away. The main idea here is that points that are closer to each other are more likely to be related to each other than points that are further.

We typically use the Tukey tri-weight in LOESS regression:

$$W(u) = \begin{cases} \left(1 - |u|^3\right)^3 & \text{if } |u| \leq 1 \\ 0 & \text{if } |u| > 1 \end{cases}$$

where u is the distance of a given data point from the point on the curve being fitted, scaled to line in the range from 0 to 1.

You should definitely read more on the subject in the following links:

https://rafalab.github.io/dsbook/smoothing.html#local-weighted-regression-loess

http://r-statistics.co/Loess-Regression-With-R.html

http://geog.uoregon.edu/bartlein/courses/geog495/lec14.html

The following links are particularly valuable:

https://www.itl.nist.gov/div898/handbook/pmd/section1/pmd144.htm

https://www.itl.nist.gov/div898/handbook/pmd/section1/dep/dep144.htm

The last page contains a thorough example. My hope is that by working out the details of this example, you can write your own function to implement LOESS.

Main Goal

- You must write your own function that carries out LOESS regression. Your function must be well commented to explain what's going on.
- Your function must be of the form:

```
# Note span and degree are shown with their default values.
#  degree should be 1 or 2 only
#  span can be any value in (0, 1) non-inclusive.

myloess <- function(x, y, span = 0.5, degree = 1, show.plot = TRUE){

  # Your code goes here

  return(list of objects seen below)
}

# Your function should return a named list containing the following:
# span: proportion of data used in each window (controls the bandwidth)
# degree: degree of polynomial
# N_total: total number of points in the data set
# Win_total: total number of windows
# n_points: number of points in each window in a vector
# SSE: Error Sum of Squares (Tells us how good of a fit we had).
# loessplot: An object containing the ggplot so that we can see the plot later.
#  We want this even if show.plot = FALSE
#  Note: you are NOT allowed to simply use stat_smooth() or geom_smooth() to have it automatically do LOESS.
#  You should use geom_line() or similar to plot your final the LOESS curve.

# Make sure you can access the objects properly using the $ notation.
```

Permissions and Restrictions

You are permitted to read about the subject further from any resource that you like. However, you must submit a list of all of the resources that you used including examples, etc.

You are not permitted to copy any already existing code from any other sources.

We will be taking plagiarism very seriously on this assignment. Attempting to look at the source code for the built-in `lowess()` function for R is a violation along with trying to find other codes online. My TAs and I have already gathered a library of codes (including SAS, R, Python) from online to check your solution against. So don't be tempted to copy!

Using your Function

You will use your function on a couple of data sets below, but feel free to practice using it on as many data sets as you like on your own.

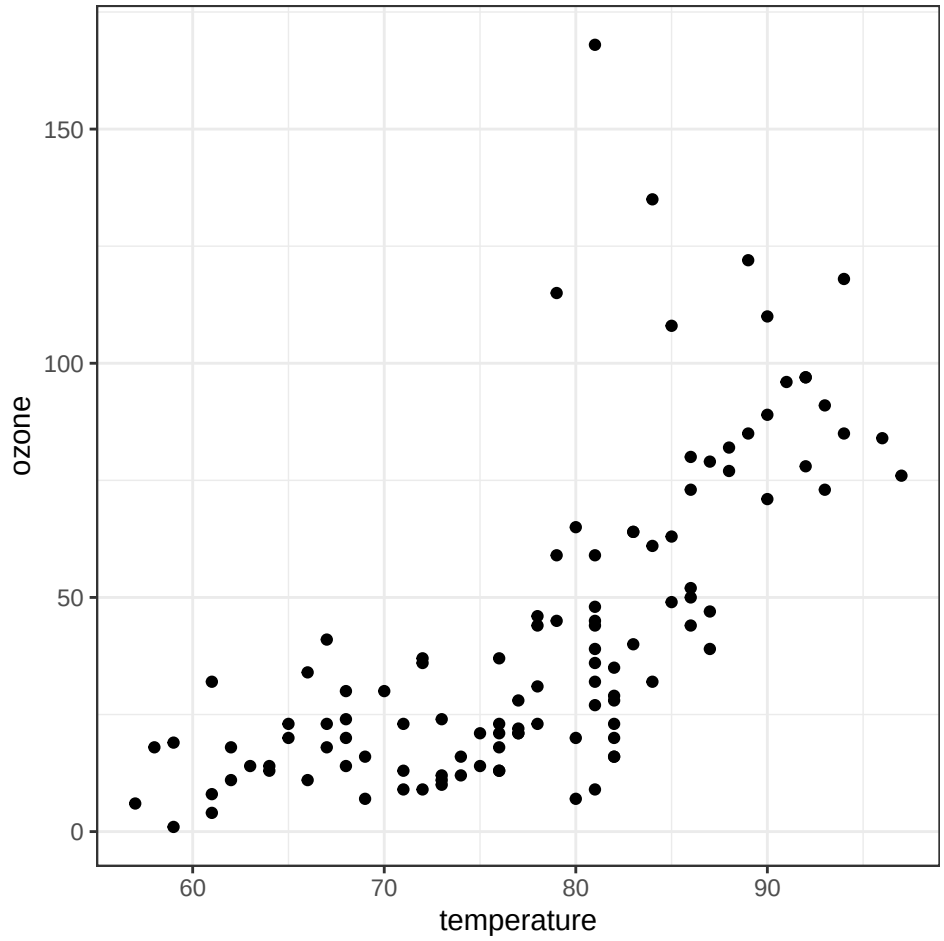
You will compare your results with the `loess()` function that is built into R.

You will also compare your plots with the output from `ggplot() + geom_smooth()/stat_smooth()` that uses `loess` (you need to look at examples to see how to use it). Use `se = F` as we don't need to see confidence bands.

Problem 1 First consider the `ozone` dataset, see Canvas for the `ozone.RData` file.

```
library(ggplot2)
load("data/ozone.RData")
data("ozone")
```

```
ggplot(ozone, aes(x = temperature, y = ozone)) + theme_bw() + geom_point()
```

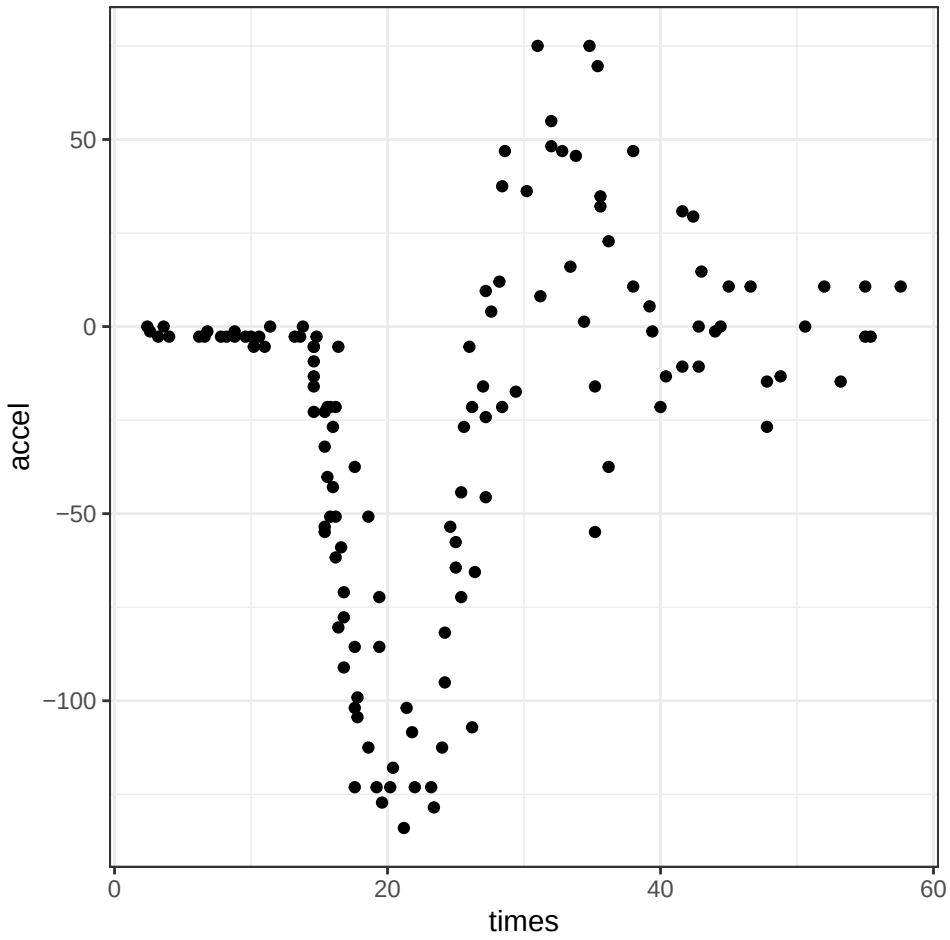


1. Fit polynomials of different degrees between 1 and 6 with `ozone` being regressed upon `temperature`. Which polynomial fit appears to work the best?
2. Use your function to determine LOESS regression fits on the data. Use span = 0.25 to 0.75 with steps of 0.05. Do this for both `degree = 1` and `degree = 2`. List all of the results in a table. Plot the three "best" `degree = 1` and three "best" `degree = 2` fits that you determined, make sure to put the appropriate span in the title for each plot (there should be a total of 6 plots). You determined the "best" by comparing the residual standard errors. However if you visually inspect the best compared to the 2nd and 3rd best fits, do you feel that you may have over-fit the data?
3. Be sure to compare your results with that found from the built-in `loess()` function and your plots with that which is obtained from `ggplot() + geom_smooth()/stat_smooth()`.

Problem 2 Consider the `mcycle` dataset from the `MASS` library package.

```
library(MASS)
data("mcycle")
```

```
ggplot(mcycle, aes(x = times, y = accel)) + theme_bw() + geom_point()
```



1. This dataset is notoriously difficult to fit with polynomial regression. Later we will study how to use **regression trees** to fit this data. Until then, determine the three "best" `degree = 1` and three "best" `degree = 2` LOESS regression fits by finding the best `span` between 0.25 and 0.75 with steps of 0.05 for each degree. Report your answers in a table. Plot the three best first for both `degree = 1` and `degree = 2`. Based upon a visual inspection, which models provide the "best" fit?

2. Be sure to compare your results with that found from the built-in `loess()` function and your plots with that which is obtained from `ggplot() + geom_smooth()/stat_smooth()`.

Final Notes:

Some of the resources I have shared with you or you have found on your own are likely to discuss strategies for determining the optimal `span`. One of the easier approaches is to use the `optim()` function. However, we should also consider using cross-validation to assess the performance of our models for various values for `span`.

Part 2 k-Nearest Neighbors for Classification

k-Nearest Neighbors (kNN) is one of the most simple non-parametric methods in supervised learning. It can be used for both regression and classification. The general idea is that it tries to group observations together based upon similarity.

The algorithm

1. Calculate the distance between each training data point and the data point we're examining.

   The distance can be computed a number of different ways, for example:

   Manhattan distance $= \sum_{i=1}^{n} |x_i - y_i|$

   Euclidean distance $= \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2}$. i.e. the L2-norm. Go ahead and use this for your function.

   Minkowski distance $= \sqrt[p]{\sum_{i=1}^{n} |x_i - y_i|^p}$

2. Identify the first k data points with the smallest distance to our data point of interest.
3. Find the label which occurs the most out of our k closest training data points (aka majority label). This is the label that you will assign to this observation.

Some resources:

https://towardsdatascience.com/k-nearest-neighbors-algorithm-with-examples-in-r-simply-explained-knn-1f2c88da405c
https://www.edureka.co/blog/knn-algorithm-in-r/

Main Goal

Write your own function that carries out k-Nearest Neighbors algorithm for classification. Keep in mind it's also possible to use kNN for regression as well, but your particular implementation is for classification.

The `class` library packages contains a `knn()` function that can be used for classification. Although there are other libraries that contain kNN functions, you will build your function to work in a similar way to the one from the `class` library.

```
# Your function will have the following inputs similar to what you would find with the
#  knn() function
# train - matrix or data frame of training set cases
# test - matrix or data frame of test set cases.
#   A vector will be interpreted as a row vector for a single case.
# cl_train - factor of true classification of training set
# cl_test - factor of the true classification of the test set
# k - number of neighbors considered, the default value is 3

mykNN <- function(train, test, cl_train, cl_test k = 3) {

  # Your code goes here

  return(list of objects seen below)

}

# You will return the following
# A categorical vector for the predicted categories for the testing data
# The accuracy of the classification
# The error rate = 1 - accuracy
# A confusion matrix
# The value of k used
```

Example of using the `knn()` function

Ultimately our goal is to improve upon the `knn()` function a bit but we should get the same general answer.

A good dataset to practice with is the `iris` dataset. You should randomly sample from `iris` splitting it into training and testing subsets. Below we will split the `iris` dataset more simply by taking the first 25 observations from each `Species` group for the training data and the remaining 25 observations for the testing data.

```
# Consider the example that makes use of the knn() function from the class library
library(class)
# We'll subset iris into training and testing sets this is a poor split but good enough for demonstration
train <- iris[c(1:25, 51:75, 101:125), 1:4]
test <- iris[c(26:50, 76:100, 126:150), 1:4]

# The actual categories from  the training set
actual_train <- iris[c(1:25, 51:75, 101:125), 5]
actual_test <- iris[c(26:50, 76:100, 126:150), 5]

knn_pred <- knn(train, test, actual_train, k = 3, prob = F)
# The predicted categories for the testing data observations
# In practice we don't want to see this, but we want to use it.
knn_pred
```

```
 [1] setosa     setosa     setosa     setosa     setosa     setosa     setosa     setosa
 [9] setosa     setosa     setosa     setosa     setosa     setosa     setosa     setosa
[17] setosa     setosa     setosa     setosa     setosa     setosa     setosa     setosa
[25] setosa     versicolor versicolor virginica  versicolor versicolor versicolor versicolor
[33] versicolor virginica  versicolor versicolor versicolor versicolor versicolor versicolor
[41] versicolor versicolor versicolor versicolor versicolor versicolor versicolor versicolor
[49] versicolor versicolor virginica  versicolor versicolor virginica  virginica  virginica
[57] virginica  virginica  versicolor virginica  virginica  virginica  virginica  versicolor
[65] virginica  virginica  virginica  virginica  virginica  virginica  virginica  virginica
[73] virginica  virginica  virginica
Levels: setosa versicolor virginica
```

```
# Your function would differ in that it has the actual_test included
#  so that you can also return a confusion matrix.

# We can easily make a confusion matrix using one of the table() functions
# The confusion matrix
table(knn_pred, actual_test)
```

```
            actual_test
knn_pred     setosa versicolor virginica
  setosa         25          0         0
  versicolor      0         23         4
  virginica       0          2        21
```

```
# The accuracy of kNN's prediction
mean(knn_pred == actual_test)
```

```
[1] 0.92
```

Permissions and Restrictions

You are permitted to read about the subject further from any resource that you like. However, you must submit a list of all of the resources that you used including examples, etc.

You are not permitted to copy any already existing code from any other sources.

We will be taking plagiarism very seriously on this assignment. Attempting to look at the source code for any of the `knn()` functions from any R library is a violation along with trying to find other codes online. My TAs and I have already gathered a library of codes (including SAS, R, Python) from online to check your solution against. So don't be tempted to copy!

Using your Function

Use your function on the `Auto` dataset from the `ISLR` library. You can read about the variables using `help("Auto")`

```
# Some pre-processing
library(ISLR)
# Remove the name of the car model and change the origin to categorical with actual name
Auto_new <- Auto[, -9]
# Lookup table
newOrigin <- c("USA", "European", "Japanese")
Auto_new$origin <- factor(newOrigin[Auto_new$origin], newOrigin)

# Look at the first 6 observations to see the final version
head(Auto_new)
```

```
  mpg cylinders displacement horsepower weight acceleration year origin
1  18         8          307        130   3504         12.0   70    USA
2  15         8          350        165   3693         11.5   70    USA
3  18         8          318        150   3436         11.0   70    USA
4  16         8          304        150   3433         12.0   70    USA
5  17         8          302        140   3449         10.5   70    USA
6  15         8          429        198   4341         10.0   70    USA
```

1. Randomly split your data into two data frames. Use 70% of your data for the training data and 30% for the testing data.
2. Use your `mykNN()` function with your training and testing data for several values of k and record your accuracy.
3. Make a table of the accuracy for different values of k. Display this table nicely using a table maker like `kable()`.
4. Make a plot of the accuracy versus k to determine the best number of neighbors to use.
5. Show the final confusion matrix for the best value of k and make sure the accuracy is stated.