DSCC 465

Assignment 3

Uzair Tahamid Siam

Q1

a)

To summarize,

the paper "*Ethnicity, Insurgency, and Civil War*" by J Fearon and D Laitin explore factors that lead to civil wars. They try to find a link between civil conflicts and ethnicity. This paper argues that rather than recent changes in the post-Cold War world system, it is the steady growth in protracted crisis from the 1950s and 1960s onwards, that results in the current prevalence of civil wards. Additionally, issues such as poverty, political instability and weak states also contribute to the outbreak of civil wars as they favour rebel recruitment. According to this Stanford University paper, ethnicity, political grievances and the end of the Cold War have historically been cited as factors leading to civil wars. However, civil conflict can be viewed in terms of insurgency that is characterised by small, lightly armed groups engaged in guerilla warfare from rural bases. The existence of poverty, weak states, political instability and large populations leads to conditions that favour insurgency and recruitment to rebel groups. Fragile states are unable or unwilling to control internal conflicts due to weak local policing and corrupt counterinsurgency practices. Ethnic and religious diversity does not necessarily lead to civil conflict in a country, despite the fact that rebel groups are often mobilised along ethnic lines during warfare. They also argue that there is little evidence that the absence of democracy and respect for civil liberties and minority groups leads to the outbreak of civil war.

b)

In the section on **Descriptive Statistics** the authors mention they identified 127 conflicts between years 1945 to 1999. Each observation is a civil war conflict with 13 of those were anticolonial wars.

c)

> **Model 1:**

Dependent variable: Onset of Civil War (represented by 1 and absence of it by 0)

Independent variable:

- Prior war: -0.954**
- Per capita income: -0.344***
- log(population): 0.263***
- log(% mountainous): 0.219**

- Noncontiguous state: 0.443
- Oil exporter: 0.858**
- New state: 1.709***
- Instability: 0.618**
- Democracy^{a,c}: 0.021
- Ethnic fractionalization: 0.166
- Religious fractionalization: 0.285
- Constant: -6.731***

> **Model 2:**

Dependent variable: Onset of "Ethnic" War (represented by 1 and absence of it by 0)

Independent variable:

- Prior war: -0.849*
- Per capita income: -0.379***
- log(population): -0.389***
- log(% mountainous): 0.120
- Noncontiguous state: 0.481
- Oil exporter: 0.809*
- New state: 1.777***
- Instability: 0.385
- Democracy^{a,c}: 0.013
- Ethnic fractionalization: 0.146
- Religious fractionalization: 1.533*
- Constant: -8.450***

> **Model 3:**

Dependent variable: Onset of Civil War (represented by 1 and absence of it by 0)

Independent variable:

- Prior war: -0.916*
- Per capita income: 0.318***
- log(population): 0.272***
- log(% mountainous): 0.199*
- Noncontiguous state: 0.426
- Oil exporter: 0.751**
- New state: 1.658***
- Instability: 0.513*
- Ethnic fractionalization: 0.164
- Religious fractionalization: 0.326
- Anocracy: 0.521*
- Democracy^{a,d}: 0.127
- Constant: -7.019***

> **Model 4:**

Dependent variable: Onset of Civil War (Plus Empired) (represented by 1 and absence of it by 0)

Independent variable:

- Prior war: -0.688**
- Per capita income: -0.305***
- log(population): 0.267***
- log(% mountainous): 0.192*
- Noncontiguous state: 0.798**
- Oil exporter: 0.548*
- New state: 1.523***
- Instability: 0.548*
- Ethnic fractionalization: 0.490
- Constant: -6.801***

> **Model 5:**

Dependent variable: Onset of Civil War (COW) (represented by 1 and absence of it by 0)

Independent variable:

- Prior war: -0.551
- Per capita income: -0.309***
- log(population): 0.223**
- log(% mountainous): 0.418***
- Noncontiguous state : -0.171
- Oil exporter: 1.269***
- New state: 1.147*
- Instability: 0.584*
- Ethnic fractionalization: -0.119
- Religious fractionalization: 1.176*
- Anocracy: 0.597*
- Democracy^{a,d}: 0.219
- Constant: -7.503***

NOTES :

Any attribute with a *or* ***or*** is significant as the literature defines the stars as follows:**

* => p-value < 0.05

** => p-value < 0.01

*** => p-value < 0.001

and we know that p < 0.05 is usually seen as a significant value

d)

The coefficients in the table are the *weights* associated with each feature/independent variable for the model. The weights themselves have no meaning as these are **Logit Regression** models. However, raising the weights to an exponent gives us a value for **odds ratio**, i.e.

$$O_i = e^{w_i}$$

A change in the weight of attribute, w_i, will change (increase if the weight is positive and decrease if the weight is negative) the odds of onset of war vs no onset of war in a particular year (which is our dependent variable)

e)

PLEASE REFER BACK TO (C) as it has all the variables and their associated signed values for each model. Also the note at the end answers for which attributes are significant.

f)

At first glance of the weights, it seems that there is a large discrepancy between how much each of the attributes affect the dependent variable. But after a closer inspection, you can see that the scales of the independent variables vary to quite an extent. E.g. the **population** attribute is on a *log-scale* certain other attributes are *binary* and some are not in log-scale. This means that the weights do not entirely represent the impact of each variable. Obviously a variable on a log-scale has a very high range of possible values compared to other attributes. This means that the parametric equation we are trying to find to define our dependent variable must account for the large difference in range of the attributes. So, while the table associates a smaller weight to the log-scaled variables, it does so to ensure that certain variables do not completely overpower the rest. A better way to find out the relative impacts of the attributes would be normalize the data so that they all fall within the same range.

In conclusion,\ it is difficult to say which of the attributes have a greater impact by looking at the weights given to us.

In [1]:

```
import numpy as np
np.set_printoptions(precision=2)
import pandas as pd
import sklearn
import matplotlib.pyplot as plt
from numpy.typing import ArrayLike
from random import randrange
import random
random.seed(265)
import matplotlib
font = {'family' : 'Tahoma',
        'weight' : 'regular',
        'size'   : '20'}
matplotlib.rc('font', **font)
matplotlib.rc('axes',edgecolor='lightgrey')
matplotlib.rcParams['grid.color'] = 'lightgrey'
plt.style.use('seaborn-pastel')
```

Q2

a, b, c, d, e)

Matrices :

$X_{n\times m}= \begin{bmatrix} . & . & . & . \\ . & . & . & . \\ . & . & . & . \\ . & . & . & . \end{bmatrix}$

$y_{1\times m}= \begin{bmatrix} . & . & . & . \end{bmatrix}$

$\theta_{n\times 1}= \begin{bmatrix} \theta_1 \\ . \\ . \\ \theta_n \end{bmatrix}$

Equations:

$\sigma = \frac{1}{1+e^{-z}}$

$h(\theta) = \sigma(\theta^T X + b)$

In [2]:

```python
def sigmoid_f(z: ArrayLike) -> ArrayLike:
    '''
    Parameters
    -----------
    z: ArrayLike
        The argument of the sigmoid function

    Returns
    -----------
    sigmoid: ArrayLike
        The output of the sigmoid function
    '''
    sigmoid = 1/(1+np.exp(-z))
    return sigmoid

def classifier_f(w: ArrayLike, b:int, X: ArrayLike) -> ArrayLike:
    '''
    Parameters
    -----------
    w: ArrayLike
        Weights for each attribute
    b: int
        Bias value
    X: Arraylike
        Feature matrix

    Returns
    -----------
    pred: ArrayLike
        The output of the hypothesis function

    '''
    z = np.dot(w.T, X) + b
    pred = sigmoid_f(z)
    return pred

def binary_loss_f(y: ArrayLike, a: ArrayLike) -> int:
    '''
    Parameters
    -----------
    y: ArrayLike
        Target vector
```

```
    a: ArrayLike
        Prediction vector from the classifier

    Returns
    -----------
    cost: int
        Value of the cross-entropy function
    '''
    m = y.shape[1]
    cost = -1/m * np.sum(y * np.log(a) + (1-y) * np.log(1-a))
    return cost

def gradient_f(w: ArrayLike, b: int, X: ArrayLike, y: ArrayLike, alpha: int):
    '''
    Parameters
    ----------
    w: ArrayLike
        The weights that are to be optimized

    X: ArrayLike
        Numpy Array of features

    y: ArrayLike
        Numpy Array of targets

    alpha: int
        Used as a multiplier for the step-size

    Returns
    ----------
    w: int
        Returns the optimized values of the weights initially given as a parameter

    b: int
        Returns the optimized value of the bias initially given as a parameter

    cost:
        Returns the cost at the end of each iteration
    '''
    m = X.shape[1]
    n = X.shape[0]
    A = classifier_f(w=w, b=b, X=X)
    cost = binary_loss_f(y, A)
    dw = (1/m) * np.dot(A-y, X.T)
    db = (1/m) * np.sum(A-y)
    w = w - alpha * dw.T
    b = b - alpha * db

    return w, b, cost

def optimizer_f(X: ArrayLike, y: ArrayLike, alpha: int, iterations: int, print_cost=True):
    '''
    Parameters
    ----------
    w: ArrayLike
        The weights that are to be optimized

    X: ArrayLike
        Numpy Array of features

    y: ArrayLike
        Numpy Array of targets

    number_of_steps: int
        Number of iterations for the gradient descent algorithm
```

```
    alpha: int
        Used as a multiplier for the step-size

    print_cost: bool
        True if the cost is to be printed

    Returns
    ----------
    w: int
        Returns the optimized values of the weights initially given as a parameter

    b: int
        Returns the optimized value of the bias initially given as a parameter

    cost_l:
        Returns the list of the cost after all iterations
    '''
    X = X.T # m x n to n x m
    y = y.reshape((1, X.shape[1]))
    w = np.zeros((X.shape[0],1))
    b = 0
    cost_l = []
    for i in range(iterations):
        w, b, cost = gradient_f(w, b, X, y, alpha)
        cost_l.append(cost)
        if print_cost:
            if i % (iterations/50) == 0:
                print(f"Cost after {i}-iterations: {cost}")

    return w,b, cost_l
```

In [3]:

```
# importing data

from sklearn.datasets import load_breast_cancer
dataset = load_breast_cancer(as_frame=True).frame
```

In [4]:

```
dataset
```

Out[4]:

	mean radius	mean texture	mean perimeter	mean area	mean smoothness	mean compactness	mean concavity	mean concave points	mean symmetry	mea fract dimensio
0	17.99	10.38	122.80	1001.0	0.11840	0.27760	0.30010	0.14710	0.2419	0.078
1	20.57	17.77	132.90	1326.0	0.08474	0.07864	0.08690	0.07017	0.1812	0.056
2	19.69	21.25	130.00	1203.0	0.10960	0.15990	0.19740	0.12790	0.2069	0.059
3	11.42	20.38	77.58	386.1	0.14250	0.28390	0.24140	0.10520	0.2597	0.097
4	20.29	14.34	135.10	1297.0	0.10030	0.13280	0.19800	0.10430	0.1809	0.058
...	...	...	...	...	...	...	...	...	...	
564	21.56	22.39	142.00	1479.0	0.11100	0.11590	0.24390	0.13890	0.1726	0.056
565	20.13	28.25	131.20	1261.0	0.09780	0.10340	0.14400	0.09791	0.1752	0.055
566	16.60	28.08	108.30	858.1	0.08455	0.10230	0.09251	0.05302	0.1590	0.056
567	20.60	29.33	140.10	1265.0	0.11780	0.27700	0.35140	0.15200	0.2397	0.070
568	7.76	24.54	47.92	181.0	0.05263	0.04362	0.00000	0.00000	0.1587	0.058

569 rows × 31 columns

a, b)

In [5]:
```
# splitting data into features and targets

x = dataset.iloc[:, :-1].values
y = dataset.iloc[:, -1].values
```

c)

In [6]:
```
# Scaling x only since y already in [0,1]

from sklearn.preprocessing import MinMaxScaler
mms_x = MinMaxScaler()
x = mms_x.fit_transform(x)
```

d)

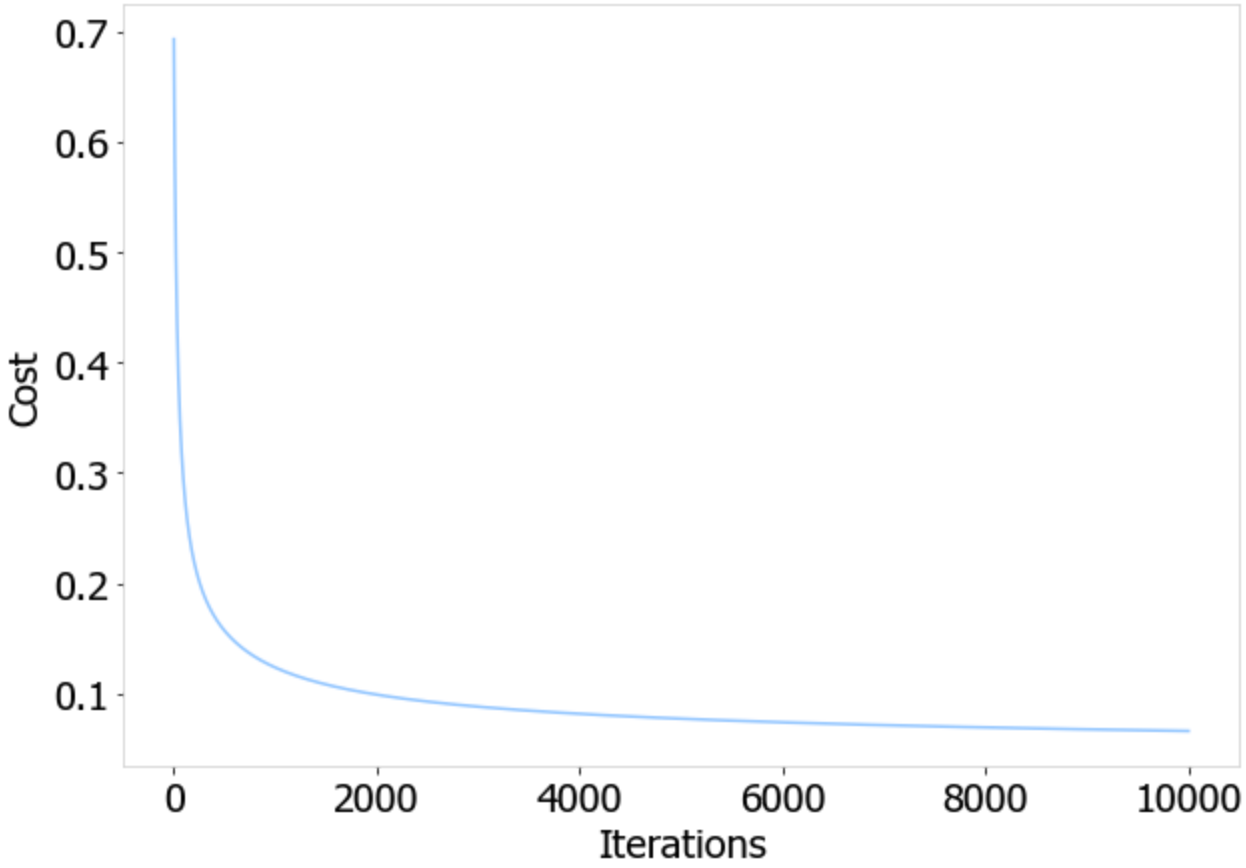
In [7]:
```
iterations = 10000
w, b, costs = optimizer_f(x, y, alpha=0.5, iterations=iterations)
```

```
Cost after 0-iterations: 0.6931471805599453
Cost after 200-iterations: 0.2190089573081807
Cost after 400-iterations: 0.16977855890618357
Cost after 600-iterations: 0.1471224255623226
Cost after 800-iterations: 0.13320273075450312
Cost after 1000-iterations: 0.1235070099271086
Cost after 1200-iterations: 0.11625789315691809
Cost after 1400-iterations: 0.11058179508064696
Cost after 1600-iterations: 0.1059883755798705
Cost after 1800-iterations: 0.10217707737224219
Cost after 2000-iterations: 0.09895175654648952
Cost after 2200-iterations: 0.09617823688344139
Cost after 2400-iterations: 0.09376128095991756
Cost after 2600-iterations: 0.09163123853205997
Cost after 2800-iterations: 0.08973589147353446
Cost after 3000-iterations: 0.08803525864865586
Cost after 3200-iterations: 0.08649816919210591
Cost after 3400-iterations: 0.08509993487791848
Cost after 3600-iterations: 0.08382072870895182
Cost after 3800-iterations: 0.08264443041624471
Cost after 4000-iterations: 0.08155778839419911
Cost after 4200-iterations: 0.08054980081380513
Cost after 4400-iterations: 0.07961125151318411
Cost after 4600-iterations: 0.07873435709692277
Cost after 4800-iterations: 0.07791249519799844
Cost after 5000-iterations: 0.07713999281994341
Cost after 5200-iterations: 0.07641195973249713
Cost after 5400-iterations: 0.07572415605572207
Cost after 5600-iterations: 0.07507288607282177
Cost after 5800-iterations: 0.07445491236935119
Cost after 6000-iterations: 0.07386738587292113
Cost after 6200-iterations: 0.07330778843999079
Cost after 6400-iterations: 0.07277388542428678
Cost after 6600-iterations: 0.07226368624642869
Cost after 6800-iterations: 0.07177541142301842
Cost after 7000-iterations: 0.07130746484543367
Cost after 7200-iterations: 0.07085841035198491
Cost after 7400-iterations: 0.07042695183211982
Cost after 7600-iterations: 0.07001191625260525
```

```
Cost after 7800-iterations: 0.06961223911375602
Cost after 8000-iterations: 0.06922695193670049
Cost after 8200-iterations: 0.0688551714562228
Cost after 8400-iterations: 0.06849609025231031
Cost after 8600-iterations: 0.06814896860046728
Cost after 8800-iterations: 0.06781312735866418
Cost after 9000-iterations: 0.06748794173941769
Cost after 9200-iterations: 0.06717283584041388
Cost after 9400-iterations: 0.06686727782747727
Cost after 9600-iterations: 0.06657077568043157
Cost after 9800-iterations: 0.06628287342622179
```

In [8]:

```
plt.figure(figsize=(10,7))
plt.plot(np.arange(0, iterations, 1), costs)
plt.xlabel('Iterations')
plt.ylabel('Cost')
plt.show()
```



e, f)

In [9]:

```
weights = (np.ravel(w))
attrNames = dataset.columns[:-1]
oddsRatio = np.exp(np.ravel(w))
eq = np.stack((weights, attrNames, oddsRatio), axis=1)
eq_df = pd.DataFrame(data=eq, index=None, columns=['Weights', 'Attribute Name', 'Odds Rati
eq_df.sort_values(by='Weights', ascending=False, inplace=True, ignore_index=True)
```

In [10]:

```
eq_df
```

Out[10]:

	Weights	Attribute Name	Odds Ratio
0	4.023169	mean fractal dimension	55.877883

	Weights	Attribute Name	Odds Ratio
1	3.874137	compactness error	48.141124
2	2.441101	fractal dimension error	11.485683
3	1.323793	mean compactness	3.757646
4	1.298801	symmetry error	3.664901
5	1.216559	concavity error	3.375554
6	0.430571	texture error	1.538135
7	0.087119	mean symmetry	1.091026
8	0.03104	concave points error	1.031526
9	-0.199341	mean smoothness	0.819271
10	-0.348645	worst compactness	0.705643
11	-0.771565	worst fractal dimension	0.462289
12	-0.904273	smoothness error	0.404836
13	-1.311277	mean radius	0.269476
14	-1.393625	mean perimeter	0.248174
15	-2.62534	mean area	0.072415
16	-3.182783	worst concavity	0.04147
17	-3.343976	mean texture	0.035296
18	-3.839267	worst smoothness	0.021509
19	-3.890705	area error	0.020431
20	-4.149751	worst symmetry	0.015768
21	-4.412854	worst perimeter	0.012121
22	-4.553091	perimeter error	0.010535
23	-4.573361	mean concavity	0.010323
24	-4.836219	worst area	0.007937
25	-5.150528	worst concave points	0.005796
26	-5.190218	worst radius	0.005571
27	-5.433743	worst texture	0.004367
28	-6.286055	mean concave points	0.001862
29	-6.321238	radius error	0.001798

Reference: https://medium.com/@mubarakb/how-to-interpret-the-weights-in-logistic-regression-89bb03249f27

A change in the weight of an attribute, w_i, will change (increase if the weight is positive and decrease if the weight is negative) the odds of breast cancer vs no breast cancer by the odds ratio, O_i associated to the weight of the attribute, w_i.

$$O_i = e^{w_i}$$

E.g. An increase in **mean fractal dimension** will **increase** the odds of having cancer vs not having cancer by a whopping "55.877883" times. While an increase in **worst texture** will **decrease** the odds of having

cancer over not having cancer by "0.004367" times.

We can interpret all the weights in a similar way.

Q3

a) Implementation of the CV methods

In [11]:

```
from sklearn.utils import shuffle
from sklearn.metrics import mean_squared_error
from sklearn.linear_model import LinearRegression


def LOOCV(dataset: ArrayLike, model, X: ArrayLike, y: ArrayLike):
    '''
    Parameters
    ----------
    dataset:
        A pandas dataframe

    model:
        A linear regression model

    X: ArrayLike
        An array of the feature names in the dataset

    y: ArrayLike
        An array of the target names in the dataset

    Returns
    ----------
        The result of K-Fold CV with k=len(dataset)
    '''
    assert isinstance(model, sklearn.linear_model.LinearRegression), "Your model must be a

    return KFCV(dataset, model, X, y, k=len(dataset))


def KFCV(dataset: ArrayLike, model, X: ArrayLike, y: ArrayLike, k: int = 5):
    '''
    Parameters
    ----------
    dataset:
        A pandas dataframe

    model:
        A linear regression model

    X: ArrayLike
        An array of the feature names in the dataset

    y: ArrayLike
        An array of the target names in the dataset

    k: int
        The number of folds that the dataset is to be subdivided into

    Returns
    ----------
    model_dict: dict
        A dictionary with the keys as the mean square error of the model and the values as
```

```
    '''
    assert isinstance(model, sklearn.linear_model.LinearRegression), "Your model must be a

    model_dict = {}
    dataset = shuffle(dataset, random_state=265)
    folds = len(dataset) // k

    for i in range(k):

        # splitting into train-test
        test = dataset.iloc[i*folds:(i+1)*folds, :]
        train = pd.concat([dataset.iloc[:i*folds, :], dataset.iloc[(i+1)*folds: , :]])

        # fitting the model
        model = model.fit(train[X], train[y])

        # finding mse
        err = mean_squared_error(y_true=test[y], y_pred=model.predict(test[X]))

        model_dict[err] = model

    return model_dict

def TTSCV(dataset: ArrayLike, model, X: ArrayLike, y: ArrayLike, split_split: float = 0.7)
    '''
    Parameters
    ----------
    dataset:
        A pandas dataframe

    model:
        A linear regression model

    X: ArrayLike
        An array of the feature names in the dataset

    y: ArrayLike
        An array of the target names in the dataset

    split: float
        The fraction of the dataset that is to be the train set

    Returns
    ----------
    err: float
        The mean square error of the model
    '''
    assert isinstance(model, sklearn.linear_model.LinearRegression), "Your model must be a

    dataset = shuffle(dataset, random_state=265)
    train_size = int(split_split * len(dataset))

    # splitting into train-test
    train = dataset.sample(n=train_size, random_state=265)
    test = dataset.drop(train.index, axis=0)

    # fitting the model

    model = model.fit(train[X], train[y])

    # finding mse
    err = mean_squared_error(y_true=test[y], y_pred=model.predict(test[X]))

    return err
```

b, c, d)

In [12]:
```
# importing california housing data from sklearn

housing_df = sklearn.datasets.fetch_california_housing(as_frame=True).frame
```

In [13]:
```
# scaling the entire dataframe

mms = MinMaxScaler()
mms.fit(housing_df)
df = mms.fit_transform(housing_df)
```

In [14]:
```
# converting scaled df into a dataframe
df = pd.DataFrame(df, columns=housing_df.columns)
```

In [15]:
```
# selecting features and targets
x = df.iloc[:, :-1]
y = df.iloc[:, -1]
```

In [16]:
```
# initializing model
model = LinearRegression()
```

e, f)

In [17]:
```
tts_mse = TTSCV(df, model, x.columns, y.name)
kf_mse = KFCV(df, model, x.columns, y.name)
llo_mse = LOOCV(df, model, x.columns, y.name)
```

In [18]:
```
print(f"Train-test CV: {tts_mse}\nK-fold CV: {np.mean(list(kf_mse.keys()))}\nLeave One Out
```

```
Train-test CV: 0.02234982261389473
K-fold CV: 0.02246052322536991
Leave One Out CV: 0.022456875235560305
```

Conclusion

Above we have implemented three different cross-validation techniques:

> Train-Test: This method partitions the dataset into **two sub-datasets** - the test and the train set. The model then learns using the training set and we test the model on "new" data from the test set. Finally we calculate the mean squared error associated to the model.
>
> K-fold: This method partitions the original dataset into **k sub-datasets**. The model then learns using the k-1 of the subsets and we test the model on "new" data from the other one subset. Finally we iterate this process k times and calculate the mean squared error associated to the k different models. Then we return some aggregiate form (in our case the mean) of all the mean squared errors.
>
> Leave One Out: This method is similar to the K-fold method except here we divide the dataset into **n sub-datasets** with n being the length of the dataset/the number of entries in the

> dataset.

From all these methods I get that the values:

Train-test CV: 0.02234982261389473

K-fold CV: 0.02246052322536991

Leave One Out CV: 0.022456875235560305

So from my analysis, all three methods give very comparable results. However, it appears Train-Test Cross Validation ***marginally*** gives the lowest error relative to the others followed by LLOCV. This is a little surprising at first because one could naively assume that Leave One Out Cross Validation would give the lowest error. However, it might be because of certain outliers that may have caused the mean of LLOCV to be higher than K-Fold CV as the LLOCV does many scans of the entire database.

In [ ]: