Sardar Patel Institute of Technology,Mumbai
Department of Electronics and Telecommunication Engineering
B.E. Sem-VII- PE-IV (2024-2025)
IT 24 - AI in Healthcare

Experiment 4: KNN

Name: Adwait Purao **UID: 2021300101** **Date:** 19-09-2024

Objective:

Write a program to implement k-Nearest Neighbor algorithm for Classification and Regression on healthcare dataset.

Outcomes:

- **Appropriately interpret results of classification and regression**
- **Print both correct and wrong predictions.**

System Requirements:

Linux OS with Python and libraries or R or windows with MATLAB

Theory:

K-Nearest Neighbors (KNN) Algorithm

Overview: K-Nearest Neighbors (KNN) is a simple, versatile, and widely used machine learning algorithm for classification and regression tasks. It is a non-parametric, instance-based learning algorithm that classifies new data points based on their proximity to the training data.

Key Concepts:

1. **Instance-Based Learning:**
    - KNN does not explicitly learn a model; instead, it stores the training instances and uses them directly to make predictions. This means KNN can adapt to new data without the need for retraining.
2. **Distance Metrics:**
    - KNN relies on distance metrics to find the nearest neighbors. Commonly used distance measures include:
        - **Euclidean Distance:** The most commonly used metric, calculated as the straight-line distance between two points in Euclidean space.
        - **Manhattan Distance:** Also known as L1 distance, it is calculated as the sum of absolute differences between points.
        - **Minkowski Distance:** A generalization of Euclidean and Manhattan distances, allowing for a customizable parameter.

3. **Choosing K:**
    - The parameter KKK represents the number of nearest neighbors to consider when making a prediction. A smaller KKK can be sensitive to noise in the data, while a larger KKK smooths out the decision boundary but may overlook local patterns. The optimal KKK is typically found through cross-validation.
4. **Classification and Regression:**
    - In classification tasks, KNN predicts the class of a data point by finding the most common class among its KKK nearest neighbors.
    - In regression tasks, KNN predicts a value based on the average (or weighted average) of the values of its KKK nearest neighbors.
5. **Feature Scaling:**
    - Since KNN relies on distance calculations, it is sensitive to the scale of the features. Therefore, it is crucial to standardize or normalize the dataset to ensure all features contribute equally to the distance computation.

Advantages:

- **Simplicity:** Easy to understand and implement.
- **No Assumptions:** KNN does not make any assumptions about the underlying data distribution (non-parametric).
- **Adaptability:** Can be used for both classification and regression tasks.

Disadvantages:

- **Computationally Intensive:** KNN can be slow during prediction, especially with large datasets, since it needs to compute distances for all training instances.
- **Curse of Dimensionality:** The performance of KNN deteriorates with high-dimensional data due to increased distance variability.
- **Sensitive to Noise:** Outliers can affect the classification results significantly.

Applications: KNN is commonly used in various fields, including:

- Recommendation systems (e.g., collaborative filtering)
- Image recognition
- Document classification
- Medical diagnosis

How does KNN works?

The K-NN working can be explained on the basis of the below algorithm:

- Step-1: Select the number K of the neighbors
- Step-2: Calculate the Euclidean distance of K number of neighbors
- Step-3: Take the K nearest neighbors as per the calculated Euclidean distance.
- Step-4: Among these k neighbors, count the number of the data points in each category.
- Step-5: Assign the new data points to that category for which the number of the neighbor is maximum.
- Step-6: Our model is ready.

Code:

1. Regression

```
# -*- coding: utf-8 -*-

"""AIH_Exp_4.1_KNN_Regression


Automatically generated by Colab.




 """


from google.colab import drive

drive.mount('/content/drive')


"""<font size=5><p style="color:purple"> EDA and Visualizations"""


# Commented out IPython magic to ensure Python compatibility.

import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

# %matplotlib inline

import seaborn as sns

import warnings

warnings.filterwarnings('ignore')


df = pd.read_csv('/content/drive/MyDrive/AIH datasets/insurance.csv')
```

```
df.head()


df.shape


df.describe()


df.dtypes


df.isnull().sum()


"""<font size='2' font>We have 0 missing values which is very good.
Now let's do EDA with some cool graphs :) First we'll see how the charges are distributed according to given factors
"""


sns.set(style='whitegrid')
f, ax = plt.subplots(1,1, figsize=(12, 8))
ax = sns.distplot(df['charges'], kde = True, color = 'c')
plt.title('Distribution of Charges')


f, ax = plt.subplots(1, 1, figsize=(12, 8))
ax = sns.distplot(np.log10(df['charges']), kde = True, color = 'r' )


charges = df['charges'].groupby(df.region).sum().sort_values(ascending = True)
f, ax = plt.subplots(1, 1, figsize=(8, 6))
ax = sns.barplot(x=charges.head().index, y=charges.head().values, palette='Blues')
```

```
f, ax = plt.subplots(1, 1, figsize=(12, 8))
ax = sns.barplot(x='region', y='charges', hue='sex', data=df,
palette='cool')
```

```
f, ax = plt.subplots(1,1, figsize=(12,8))
ax = sns.barplot(x = 'region', y = 'charges',
                 hue='smoker', data=df, palette='Reds_r')

f, ax = plt.subplots(1, 1, figsize=(12, 8))
ax = sns.barplot(x='region', y='charges', hue='children', data=df,
palette='Set1')

ax = sns.lmplot(x = 'age', y = 'charges', data=df, hue='smoker',
palette='Set1')
ax = sns.lmplot(x = 'bmi', y = 'charges', data=df, hue='smoker',
palette='Set2')
ax = sns.lmplot(x = 'children', y = 'charges', data=df, hue='smoker',
palette='Set3')

"""<font size='2' font>Smoking has the highest impact on medical costs,
even though the costs are growing with age, bmi and children. Also people
who have children generally smoke less, which the following violinplots
shows too"""

f, ax = plt.subplots(1, 1, figsize=(10, 10))
ax = sns.violinplot(x = 'children', y = 'charges', data=df,
                 orient='v', hue='smoker', palette='inferno')
```

```
##Converting objects labels into categorical

df[['sex', 'smoker', 'region']] = df[['sex', 'smoker',
'region']].astype('category')

df.dtypes
```

```
##Converting category labels into numerical using LabelEncoder

from sklearn.preprocessing import LabelEncoder

label = LabelEncoder()

label.fit(df.sex.drop_duplicates())

df.sex = label.transform(df.sex)

label.fit(df.smoker.drop_duplicates())

df.smoker = label.transform(df.smoker)

label.fit(df.region.drop_duplicates())

df.region = label.transform(df.region)

df.dtypes


f, ax = plt.subplots(1, 1, figsize=(10, 10))

ax = sns.heatmap(df.corr(), annot=True, cmap='cool')


# Import necessary libraries

from sklearn.neighbors import KNeighborsRegressor

from sklearn.model_selection import train_test_split, GridSearchCV

from sklearn.preprocessing import StandardScaler

from sklearn import metrics


# One-hot encoding for categorical columns ('sex', 'smoker', 'region')

df_encoded = pd.get_dummies(df, columns=['sex', 'smoker', 'region'],
drop_first=True)
```

```
# Define X (features) and y (target)
X = df_encoded.drop('charges', axis=1)  # Features (dropping 'charges')
y = df_encoded['charges']               # Target variable ('charges')

# Feature scaling
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Splitting the dataset into training and testing sets
x_train, x_test, y_train, y_test = train_test_split(X_scaled, y,
test_size=0.2, random_state=0)

# Initialize the KNN model
knn_model = KNeighborsRegressor()

# Hyperparameter tuning using GridSearchCV
param_grid = {
    'n_neighbors': [3, 5, 7, 9],  # Testing different values of neighbors
    'weights': ['uniform', 'distance'],   # Uniform vs distance-based
weighting
    'metric': ['euclidean', 'manhattan']  # Different distance metrics
}

# Grid search with cross-validation
grid_search = GridSearchCV(knn_model, param_grid, cv=5, scoring='r2',
return_train_score=True)
grid_search.fit(x_train, y_train)
```

```
# Best model from grid search
best_knn_model = grid_search.best_estimator_

# Predictions using the best model
y_train_pred_knn = best_knn_model.predict(x_train)
y_test_pred_knn = best_knn_model.predict(x_test)

# Print the best parameters
print("Best K-Nearest Neighbors Parameters:")
print(grid_search.best_params_)
```

```
# Print R-squared for training and testing data
train_r2 = best_knn_model.score(x_train, y_train)
test_r2 = best_knn_model.score(x_test, y_test)

print("Training Accuracy (R-squared):", train_r2)
print("Testing Accuracy (R-squared):", test_r2)

# Print Mean Squared Error for training and testing data
train_mse = metrics.mean_squared_error(y_train, y_train_pred_knn)
test_mse = metrics.mean_squared_error(y_test, y_test_pred_knn)

print("Training Mean Squared Error:", train_mse)
print("Testing Mean Squared Error:", test_mse)

# Print R-squared error using r2_score (for completeness)
```

```
train_r2_error = 1 - train_r2

test_r2_error = 1 - test_r2


print("Training R-squared Error:", train_r2_error)

print("Testing R-squared Error:", test_r2_error)


# Printing results for all values of n_neighbors considered during GridSearchCV

print("\nResults for different values of n_neighbors:")


# Extracting the results from GridSearchCV

results = grid_search.cv_results_


# Iterate over the different parameter combinations and print R-squared scores

for i in range(len(results['params'])):

    params = results['params'][i]

    mean_train_r2 = results['mean_train_score'][i]  # Mean R-squared score for training

    mean_test_r2 = results['mean_test_score'][i]    # Mean R-squared score for testing

    std_test_r2 = results['std_test_score'][i]      # Standard deviation of the test score


    print(f"n_neighbors: {params['n_neighbors']}, metric: {params['metric']}, weights: {params['weights']}")

    print(f"Training R-squared (mean): {mean_train_r2:.4f}")

    print(f"Testing R-squared (mean): {mean_test_r2:.4f} ± {std_test_r2:.4f}")
```

```
    print("-" * 50)
```

2. Classification

```
# -*- coding: utf-8 -*-

"""AIH_Exp_4.2_KNN_Classification


Automatically generated by Colab.


Original file is located at


https://colab.research.google.com/drive/1FH__rJloUVlRMqsiu4IK_9sG2CisIwhB


## **1. Importing Required Libraries**

"""


from google.colab import drive

drive.mount('/content/drive')


# Data Analysis

import pandas as pd

import numpy as np


# Statistical Analysis

import scipy.stats as stats


# Data Visualization

import matplotlib.pyplot as plt
```

```
import seaborn as sns


# Ignore warnings

import warnings

warnings.filterwarnings('ignore')


"""### **Import the dataset**

Kindly check the directory path where the dataset is stored and change accordingly.

"""


# Breast Cancer Wisconsin (Diagnostic) Data Set

breast_cancer_data = pd.read_csv('/content/drive/MyDrive/AIH datasets/Breast Cancer data.csv')


breast_cancer_data.head(3)


"""## **2. Data Exploration**


**How many features are there?**

"""


# Features in the dataset

print("Features in the dataset: \n{cols} \n\n Number of features in the dataset is {num_features}"

.format(cols = list(breast_cancer_data.columns), num_features = len(breast_cancer_data.columns)))
```

```
"""**Description of the dataset**


Tabulation of descriptive statistics for the Wisconsin Breast Cancer
dataset.
"""


# Dataset Description
breast_cancer_data.describe()


"""**DataFrame Information**
Information about a DataFrame including the index dtype and columns,
non-null values.


**What type are each of the variables (eg. Categorical, ordinal,
continuous, binary etc.?)**
"""


# Information about the dataframe
breast_cancer_data.info()


"""From the above description, we can see that *most of the features in
the dataset* is of type `float64` whereas, the id is of type `int64` and
diagnosis is `object` datatype."""


breast_cancer_data['diagnosis'].unique()


"""The `diagnosis` column is a `categorical` type as it contains the
values `B` - Benign and `M` - Malignant."""
```

```
df_A = breast_cancer_data[['id','diagnosis','Unnamed: 32']]


df_A.head(3)


print("Number of Null values in the Unnamed: 32 column is", df_A['Unnamed: 32'].isnull().sum())


"""**Note :** The below colums needs some attention

* `id` - cannot be used for classification

* `diagnosis` - class labels

* `Unnamed: 32` - Includes NaN so will not be helpful for classification

Thus *dropping* the above columns from the dataset.

"""


# The labels are the diagnosis column

labels_data = breast_cancer_data['diagnosis']


list_a = ['Unnamed: 32','id','diagnosis']


# The features are the rest of the columns

features_data = breast_cancer_data.drop(list_a, axis=1)


features_data.head(3)


"""## **3. Data Visualization and Feature Selection**"""
```

```
# Benign and Malignant count

print("Number of Benign {benign_count}\nMalignant count:
{malignant_count}"

.format(benign_count = labels_data.value_counts()[0], malignant_count =
labels_data.value_counts()[1]))


plt.figure(figsize=(8,6))

labels_count = sns.countplot(labels_data, label="Count")


"""**Note:** We can see that the `dataset is imbalanced`, as the number of
Benign samples is larger than the Malignant samples, so we `will need to
balance the dataset`.


#### **3.1 Normalization**


* Normalization is a process where values are shifted and scaled between 0
and 1.


* Also called as *Min-Max Scaling*


* It is used to normalize the data so that the features have a similar
scale.


$$X' = \frac{X - X_{min}}{X_{max} - X_{min}}$$


Where $X_{min}$ is the mean and $X_{max}$ are the minimum, and maximum
values of the feature.
```

```
* Normalization is an optimal strategy when the dataset doesn't have a *Gaussian Distribution*. Useful for algorithms like K-NN *that do not assume any distribution.*

"""
```

```
from sklearn.preprocessing import MinMaxScaler


# Creating an Object of StandardScaler

scaler = MinMaxScaler()


# Fit the dataframe to the scaler
```

```
print(scaler.fit(features_data))


# Transform the dataframe

features_scaled = scaler.transform(features_data)


# Convertin the scaled array to dataframe

features_scaled = pd.DataFrame(features_scaled,
columns=features_data.columns)


features_scaled.head()


"""**Violin plot by group of 10 features** for observing the distribution of numeric data, useful for comparision of distributions between multiple groups


**a. First 10 Features**

"""
```

```
# First ten features
violin_data = pd.concat([features_scaled.iloc[:,0:10], labels_data],
axis=1)


violin_data = pd.melt(violin_data, id_vars="diagnosis" ,
var_name='features', value_name='value')


plt.figure(figsize=(15,8))
sns.violinplot(data=violin_data, x="features", y="value" ,palette="Set2",
hue="diagnosis",
    split=True, inner="quart", scale="count")


plt.xticks(rotation=90)


"""From the above plot we can observe, that few features such as
`radius_mean` and `texture_mean` have a similar distribution, where the
median of Benign is separated from Malignant. Such features can be useful
for classification.


Whereas, `fractal_dimension_mean` feature has almost similar median of
Benign and Malignant, Thus it doesn't make sense to use this feature for
classification.


**b. Second 10 Features**
"""


violin_data = pd.concat([features_scaled.iloc[:,10:20], labels_data],
axis=1)
```

```
violin_data = pd.melt(violin_data, id_vars="diagnosis" ,
var_name='features', value_name='value')


plt.figure(figsize=(15,8))
sns.violinplot(data=violin_data, x="features", y="value" ,palette="Set2",
hue="diagnosis",
    split=True, inner="quart", scale="count")


plt.xticks(rotation=90)


"""**c. Rest of the Features**"""


violin_data = pd.concat([features_scaled.iloc[:,20:31], labels_data],
axis=1)


violin_data = pd.melt(violin_data, id_vars="diagnosis" ,
var_name='features', value_name='value')


plt.figure(figsize=(15,8))
sns.violinplot(data=violin_data, x="features", y="value" ,palette="Set2",
hue="diagnosis",
    split=True, inner="quart", scale="count")


plt.xticks(rotation=90)


"""**Note:** We can observe that the features `concavity_worst` and
`concave point_worst` looks similar, but with better understanding of the
distribution and if the features are *correlated* with each other one
feature can be dropped."""
```

```
# Jointplot for the two features
graph = sns.jointplot(x="concavity_worst", y="concave points_worst",
data=features_scaled, kind="reg")
```

```
r, p = stats.pearsonr(x=features_scaled['concavity_worst'],
y=features_scaled['concave points_worst'])


phantom, = graph.ax_joint.plot([], [], linestyle="", alpha=0)


graph.ax_joint.legend([phantom],['r={:f}, p={:f}'.format(r,p)])
```

```
"""From the above *Joint Plot* we can observe the data distribution of the
above features are very similar, The `Pearsonr value` is *0.85*, which is
very close to 1.0, thus we can drop the feature `concavity_worst` and
`concave point_worst` from the dataset as they are *correlated with each
other*.


`Pearson Correlation` measures the strength of linear relationship between
two variables, values ranging between -1 and 1 where,
* -1 means two variables are negatively correlated
* 0 means the variables have no uncorrelation
* +1 means two variables are positively correlated


**Correlation between all features in the dataset**
"""


# Correlation by Heatmap
f,ax = plt.subplots(figsize=(20,25))
```

```
sns.heatmap(features_scaled.corr(), annot=True, linewidths=.5, fmt=
'.1f',ax=ax)


"""From the above *Correlation Heat Map*, we can see that the features
radius_mean, perimeter_mean, area_mean are correlated with each other.


Similarly, compactness_mean, concavity_mean, concave points_mean are
correlated with each other.
"""


# Swarm Plots
swarm_data =
pd.concat([features_scaled[["radius_mean","perimeter_mean","area_mean",
    "compactness_mean", "concavity_mean", "concave points_mean"]],
labels_data], axis=1)


swarm_data = pd.melt(swarm_data, id_vars="diagnosis" ,
var_name='features', value_name='value')


plt.figure(figsize=(15,8))


sns.swarmplot(data=swarm_data, x="features", y="value" ,palette="Set2",
hue="diagnosis")


# Swarm Plots
swarm_data =
pd.concat([features_scaled[["radius_se","perimeter_se","area_se",
"radius_worst",
    "perimeter_worst", "area_worst", 'texture_mean']], labels_data],
axis=1)
```

```
swarm_data = pd.melt(swarm_data, id_vars="diagnosis" ,
var_name='features', value_name='value')


plt.figure(figsize=(15,8))


sns.swarmplot(data=swarm_data, x="features", y="value" ,palette="Set2",
hue="diagnosis")


"""**Inference**

* From the above Heatmap, we can see that the features *radius_mean*,
*perimeter_mean*, *area_mean* are *correlated* with each other, so we can
use *area_mean*.


Similarly,

* *compactness_mean*, *concavity_mean*, *concave points_mean* are
*correlated* with each other, so that we can use *concavity_mean*.


* *radius_se*, *perimeter_se* and *area_se* are correlated, thus dropping
all features except *area_se*.


* *radius_worst*, *perimeter_worst*, *area_worst* and area_worst is taken
for classification.


* I use *area_mean* with *texture_mean*, *area_worst*


**How am I choosing only selective features from other correlated features
?**
```

```
* From the above swarm plot the area_mean is distinct and widely
distributed, features with similar nature of distribution will not be
useful for classification.

* With this reasoning, we can make use of the essential features only.
```

```
#### **3.2 Why Feature Selection?**


* Feature Selection is a technique for selecting the most relevant
features from a dataset.


* This process reduces the number of input variables and hence reduces the
complexity of the model.


**Feature Selection for K-NN**

* Distance Algorithms such as the K-NN, K-means and SVM are much affected
by the range of features in the dataset.


* These algorithms are based on distance metric, which uses the distance
between data points to determine the similarity. Thus relevant features
provide better accuracy.

"""


# List of correlated features, planned to drop

drop_list = ['perimeter_mean','radius_mean','compactness_mean','concave
points_mean','radius_se',


'perimeter_se','radius_worst','perimeter_worst','compactness_worst','conca
ve points_worst',

    'compactness_se','concave points_se','texture_worst','area_worst']
```

```
# Updated scaled features dataset

features_updated = features_scaled.drop(drop_list,axis = 1)


features_updated.head()


print("Shape after dropping correlated features",features_updated.shape)


"""**Note :** The updated features dataset is reduced to 16 features from 30. Considering only the essential features will help in improving the accuracy and reducing the complexity of the model.


#### **3.2.1 Is our Feature Selection right?**


* The selected features can be evaluated using a `RandomForestClassifier` and relevant evaluation metrics such as `Confusion Matrix, Accuracy Score` model.


* But I am not quoting the working of the RandomForest model in this notebook, as this notebook is about K-NN Classifier, and my approach towards an efficient classification.


* In this notebook, I *have explained my detailed approach for an efficient classification using K-NN and validating its performance using relevant evaluation metrics*.


## **4. Training Pipeline**


The below section includes:

1. Train-Test Split
```

```
2. K-NN Classifier


3. Evaluation Metrics


4. Fine-tuning K-NN Model - Hyperparameter Optimization


#### **4.1 Train-Test Split**


* Split the dataset into training and testing sets using
`train_test_split` from `sklearn.model_selection` module.

* In this case, the `test size is set to 0.35`, which means 35% of the
dataset will be used for testing the model performance.
"""


from sklearn.model_selection import train_test_split


# Train-Test Split
X_train, X_test, y_train, y_test = train_test_split(features_updated,
labels_data, test_size=0.35, random_state=42)


"""#### **4.2 K-NN Classifier**


* K-NN Classifier is a classification algorithm that uses `the distance
between` the training data points and the test data points to determine
the most similar data points.


* By default K-NN uses the `EUCIidean Distance` as the distance metric.
```

```
**Euclidean Distance**


<img
src="https://cdn-images-1.medium.com/max/800/1*ZrwEraj9S-u_KOWdKWc8sQ.png"
width="720" height="650" style="margin-left:auto; margin-right:auto"/>


**Hyperparameter in K-NN**


* Hyperparameters are the `adjustable parameters` that can be tuned to
improve the performance of the model.


* Hyperparameter to consider in K-NN are the `number of neighbors`,
`weights`, and the `distance metric`. Most common and simple tunable
parameter is the `number of neighbors`.


* The `n_neighbors` parameter is set to 5, which is the *default value*.


* In this project, I have optimized the K-NN Model on `number of
neighbors`, `weights`, and the `distance metric` using GridSearchCV from
`sklearn.model_selection` module, discussed in the upcoming sections.
"""


from sklearn.neighbors import KNeighborsClassifier


## KNN Classifier with K=5 (Initial)
knn = KNeighborsClassifier(n_neighbors=5)
```

```
## Fit the model

knn.fit(X_train, y_train)


## Predict the values

pred = knn.predict(X_test)


"""#### **4.3 Evaluation Metrics**


The Evaluation Metrics are imported from `sklearn.metrics` module.


* **Classification Report:** Used to evaluate the quality of predictions by classifier. The report shows the main classification metrics such as precision, recall, f1-score per class or label basis.


  * *Precision*: The proportion of the predicted classes that are correct.

  * *Recall*: The proportion of the actual classes that are correctly predicted.

  * *F1-score*: The harmonic mean of precision and recall.



* **Confusion Matrix:** *N x N* matrix for evaluating the classification model. Compares actual target values to predicted target values.

  * *True Positives:* The number of correct predictions in the actual class.

  * *True Negatives:* The number of correct predictions in the actual class.

  * *False Positives:* The number of incorrect predictions in the actual class.
```

```
  * *False Negatives:* The number of incorrect predictions in the actual class.


<img src="https://www.simplilearn.com/ice9/free_resources_article_thumb/confusion-matrix.JPG" width="330" height="230" style="margin-left:auto; margin-right:auto"/>


* **Accuracy Score:** Takes true values and predicted values as input and returns the accuracy of the model.


* **Cross Val Score:** Taking inputs as dataset and cross validation configuration and returns a list of accuracy for each fold.
  * Each Fold is a training and testing set.


<img src="https://www.mathworks.com/discovery/cross-validation/_jcr_content/mainParsys/image.adapt.full.medium.jpg/1630394319812.jpg" width="700" height="430" style="margin-left:auto; margin-right:auto;">
"""


# Classification Report and Confusion Matrix
from sklearn.metrics import classification_report, confusion_matrix


# Accuracy Score
from sklearn.metrics import accuracy_score


# Cross Validation Score
from sklearn.model_selection import cross_val_score
```

```
# Classification Report and Confusion Matrix

print("Classification Report\n",classification_report(y_test, pred),

"\n\nConfusion Matrix\n",confusion_matrix(y_test, pred))


# Accuracy Score for whole of Test Data

print("Accuracy Score",accuracy_score(y_test, pred))


# Accuracy score for a cross validation split of 5

print(cross_val_score(knn, X_train, y_train, cv=5))


"""#### **Explore how the performance of your model varies on both the
train and the validation data change as you vary the amount of training
data used ?**


Simple experiment by `changing the amount of train and test data` and see
how the model performs for `default neighbors of 5`.


* Experiment 1 - Train data 67% and Test data 33%


* Experiment 2 - Train data 80% and Test data 20%


* Experiment 3 - Train data 50% and Test data 50%



**Experiment 1 - Train data 67% and Test data 33%**

"""


# Train data 67% and Test data 33%
```

```
X_train, X_test, y_train, y_test = train_test_split(features_updated,
labels_data,
train_size=0.67, random_state=42)
```

```
knn = KNeighborsClassifier(n_neighbors=5)


## Fit the model
knn.fit(X_train, y_train)


## Predict the values
pred = knn.predict(X_test)
```

```
print("Experiment 1 - Train data 0.67 and Test data 0.33\n\nClassification
Report\n",
    classification_report(y_test, pred), "\n\nConfusion
Matrix\n",confusion_matrix(y_test, pred))


print("\nAccuracy Score",accuracy_score(y_test, pred))


"""**Experiment 2 - Train data 80% and Test data 20%**"""


# Train data 80% and Test data 20%
X_train, X_test, y_train, y_test = train_test_split(features_updated,
labels_data, train_size=0.80,
    random_state=42)


knn = KNeighborsClassifier(n_neighbors=5)
```

```
## Fit the model

knn.fit(X_train, y_train)


## Predict the values

pred = knn.predict(X_test)


print("Experiment 1 - Train data 0.80 and Test data 0.20\n\nClassification
Report\n",

    classification_report(y_test, pred), "\n\nConfusion
Matrix\n",confusion_matrix(y_test, pred))


print("\nAccuracy Score",accuracy_score(y_test, pred))


"""**Experiment 3 - Train data 50% and Test data 50%**"""


# Train data 50% and Test data 50%

X_train, X_test, y_train, y_test = train_test_split(features_updated,
labels_data, train_size=0.50,

    random_state=42)


knn = KNeighborsClassifier(n_neighbors=5)


## Fit the model

knn.fit(X_train, y_train)


## Predict the values

pred = knn.predict(X_test)
```

```
print("Experiment 1 - Train data 0.80 and Test data 0.20\n\nClassification Report\n",
    classification_report(y_test, pred), "\n\nConfusion Matrix\n",confusion_matrix(y_test, pred))
```

```
print("\nAccuracy Score",accuracy_score(y_test, pred))
```

```
"""**Inference**

For a *fixed value of 5 neighbors*, the results of the above experiments are as follows:
```

Experiment	Train Data	Test Data	Accuracy Score
1	67	33	0.936
2	80	20	0.929
3	50	50	0.947

```
* From the above observation we can see that the `accuracy score varies` as the *train data* and *test data* are varied.

* Ideally, `slightly larger proportion` of *train data* will help the algorithm to learn better. And a small proportion of *test data* will help the algorithm to evaluate the model.

**How to configure Train-Test Split ?**

* There is no optimal configuration train-test split percentage of the dataset
```

```
* The data split depeneds on several factors such as:
  * Computation cost in training and evaluating the model

  * Classes in the train and test dataset

**So What is a Good Validation Strategy..**

* In this experiment of breast cancer classification, the `dataset is imbalanced`. That is, number of benign samples is much more than the number of malignant samples.
```

```
  * Count of Benign: 357 and Malignant: 212 [Bar Plot Section 3]

* When data is split, there are possibilities *specific type of data point may go into either training or testing dataset*. This leads to problems such as `Overfitting` and `Underfitting`.

* Thus we need to balance the dataset. Splitting the dataset such that an `equal proportion` of benign and malignant samples are maintained.

* This task can be carried using a `Stratified Train-Test Split` method, where the proportions of both labels are equally maintained in train and test split.

**Stratified Train-Test Split**
"""

# Train data 67% with Stratified Split
```

```
X_train, X_test, y_train, y_test = train_test_split(features_updated,
labels_data, train_size=0.67,
    random_state=42, stratify=labels_data)
```

```
knn = KNeighborsClassifier(n_neighbors=5)


## Fit the model
knn.fit(X_train, y_train)


## Predict the values
pred = knn.predict(X_test)
```

```
print("Stratified Train-Test Split\n\nClassification
Report\n",classification_report(y_test, pred),
    "\n\nConfusion Matrix\n",confusion_matrix(y_test, pred))


print("\nAccuracy Score",accuracy_score(y_test, pred))


"""**Note:** From the above observation, we can see that `stratified
split` of the dataset has improved the accuracy of the classification
model.


* As *Stratified Split* provides the model with an opportunity for a
better understanding of the dataset, as each fold conttained a `balanced
proportion of labels` for train data.
```

```
#### **4.4 Finetuning the K-NN model - Hyperparameter Optimization**
```

```
* Hyperparamerter tuning is a process of tuning the hyperparameters of a
model to obtain the best possible performance.


* Optimal values for hyperparameters `reduced` the `noise on
classification` and `overfitting` of the model.


* In this project the Hyperparameter Optimization is done using two
techniques:


  * Grid Search


  * Elbow Method


#### **A. Elbow Method**


* In *Elbow Method* the optimal value of n_neighbors is found by `varying
the number of clusters` from a range of values.


* Calculating `Within-Cluster Sum of Squares (WCSS)` for each value of
n_neighbors or K.


* The `sum of squared distance` (WCSS) between the centroid of a cluster
and each point in the cluster. With a plot, we can observer the curve dips
after certain value, creating an elbow in the plot.


* The value corresponding to this dip is the optimal value of K.


**Accuracy Rate vs N_Neighbors**
"""
```

```
accuracy_rate = []


# Range of n_neighbors for KNN

for i in range(1,40):

    knn = KNeighborsClassifier(n_neighbors=i)

    scores = cross_val_score(knn, X_train, y_train, cv=10,
scoring='accuracy')

    accuracy_rate.append(scores.mean())


plt.figure(figsize=(12,6))
```

```
accuracy_plot = plt.plot(range(1,40), accuracy_rate, color='blue',
linestyle='dashed', marker='o',

    markerfacecolor='red', markersize=10)


accuracy_plot = plt.title('Accuracy Rate vs. K Value')


accuracy_plot = plt.xlabel('N_neighbors')


accuracy_plot = plt.ylabel('Accuracy Rate')


"""From the above plot its evident that `Accuracy Rate` is  `decreasing`
for higher values of n_neighbors. The *optimal value* for n_neighbors from
the above plot is *11*.


**Error Rate vs N_Neighbors**

"""
```

```
error_rate = []


# Range of n_neighbors for KNN

for i in range(1,50):

    knn = KNeighborsClassifier(n_neighbors=i)

    scores = cross_val_score(knn, X_train, y_train, cv=10,
scoring='accuracy')

    error_rate.append(1 - scores.mean())


plt.figure(figsize=(12,6))
```

```
error_rate_plot = plt.plot(range(1,50), error_rate, color='blue',
linestyle='dashed', marker='o',

    markerfacecolor='red', markersize=10)


error_rate_plot = plt.title('Error Rate vs. K Value')


error_rate_plot = plt.xlabel('N_neighbors')


error_rate_plot = plt.ylabel('Error Rate')


"""From the above plot its evident that `Error Rate` is  `increasing` for
higher values of n_neighbors. The *optimal value* for n_neighbors from the
above plot is *11*.



**Classification Report of KNN Model for n_neighbors = 11**
```

```
"""


# Train data 67% with Stratified Split
X_train, X_test, y_train, y_test = train_test_split(features_updated,
labels_data, train_size=0.67,
    random_state=42, stratify=labels_data)


knn = KNeighborsClassifier(n_neighbors=11)


## Fit the model
knn.fit(X_train, y_train)
```

```
## Predict the values
pred = knn.predict(X_test)


print("Stratified Train-Test Split\n\nClassification
Report\n",classification_report(y_test, pred),
    "\n\nConfusion Matrix\n",confusion_matrix(y_test, pred))


print("\nAccuracy Score",accuracy_score(y_test, pred))


"""#### **B. Grid Search**


* Loops through a predefined hyperparameter grid and returns the best
model based on the validation data.
```

```
* In this technique, the hyperparameters are `divided into discrete grid
points` and the model is trained on each grid point. And evaluated on
performance metrics.
```

```
**Note:** Unlike *Elbow Method*, *Grid Search* can be used to find optimal values of several hyperparameters, `not just limited to value of n_neighbors`.
"""


# Grid Search from Model Selection Library
from sklearn.model_selection import GridSearchCV


grid_params = { 'n_neighbors' : [5,7,9,11,13,15, 20, 25, 30, 35, 40, 45, 50, 55, 60, 65,
                70, 75, 80, 85, 90, 95, 100],
               'weights' : ['uniform','distance'],
               'metric' : ['minkowski','eUCIidean','manhattan']}


# Grid Search on KNN for 10-fold cross validation
gs = GridSearchCV(KNeighborsClassifier(), grid_params, verbose = 1, cv=10, n_jobs = -1)


g_res = gs.fit(X_train, y_train)


g_res.best_score_


g_res.best_params_


# Train data 67% with Stratified Split
X_train, X_test, y_train, y_test = train_test_split(features_updated, labels_data, train_size=0.67,
    random_state=42, stratify=labels_data)
```

```
knn = KNeighborsClassifier(n_neighbors=11, weights='uniform',
metric='manhattan')
```

```
## Fit the model
knn.fit(X_train, y_train)
```

```
## Predict the values
pred = knn.predict(X_test)
```

```
print("Stratified Train-Test Split\n\nClassification
Report\n",classification_report(y_test, pred),
    "\n\nConfusion Matrix\n",confusion_matrix(y_test, pred))
```

```
print("\nAccuracy Score",accuracy_score(y_test, pred))
```

```
"""## **Decision Boundary for the K-NN Model**
```

```
* Functioning of K-NN model depends on the local geometry of the data
distribution on a feature hyperplane.
```

```
* The line - decision boundary separates the classes in the dataset. Thus
it is not smooth and is non-linear.
```

```
The below is a plot of the decision boundary for the K-NN model used in
this project. The decision boundary is visualized using the
`plot_decision_boundary` function from the `mlxtend` module.
```

```
* The required inputs for plotting decision boundary is `array` datatype. Thus `lambda` is used to convert the `Categorical data of labels` to binaries.


* Then converted to array datatype using `np.asarray` function.


```python
key = {'B':0, 'M':1} # Benign as 0 and Malignant as 1


y = list(map(lambda i : d[i], y)) # Convert Categorical data of labels to binaries


np.array(y) # Convert to array datatype
```
"""

from mlxtend.plotting import plot_decision_regions
from sklearn.decomposition import PCA
```

```
pca = PCA(n_components = 2)


# X and Y values as inputs of array datatype
X_train2 = pca.fit_transform(X_train)
y =  np.array(y_train)
```

```
d = {'B':0, 'M':1}
y = list(map(lambda i : d[i], y))
labels_encoded = np.array(y)
```

```
# Plotting the decision regions
plt.figure(figsize=(15,10))
knn.fit(X_train2, labels_encoded)
plot_decision_regions(X_train2, labels_encoded, clf=knn, legend=2)

"""## **Conclusion**

* For the `Example Task 2` of Section C in the Summative Assessment for Statistical Computing and Empirical Methods, I have successfully implemented the `K-NN based classification for breast cancer`.

* The Wisonsin Breast Cancer dataset is an `Imbalanced Dataset` with 357 Benign samples and 212 Malignant samples.

* Detailed Data exploration and visualization is done for `Feature Selection`, as the dataset has several *correlated features*.

* Experiments based on `varying sample size` is conducted to observe the variation in performance of the Classifier.

* `Stratified` Data Split is optimal for such imbalanced dataset as it *avoids the problems* of `Overfitting` and `Underfitting`, by ensuring that the `proportion of labels` in each fold is equal. This improved the model accuracy from 0.95 to 0.95

* `Hyperparameter Optimization` is done using *Grid Search* and the traditional *Elbow Method* technique for comparison. Unlike the Elbow method, `Grid Search` can be used to find optimal values of several hyperparameters, `not just limited to value of n_neighbors`.
```

```
* Relevant `Evaluation Metrics` such as Confusion Matrix, Classification
Report, Precision, Recall, F1-Score are calculated for the K-NN model.


* The `Decision Boundary` for the K-NN model is visualized to understand
the clustering algorithm better.
"""
```

Output:

1. Regression

```
Best K-Nearest Neighbors Parameters:
{'metric': 'euclidean', 'n_neighbors': 9, 'weights': 'distance'}
Training Accuracy (R-squared): 0.9982963931606104
Testing Accuracy (R-squared): 0.856102443142976
Training Mean Squared Error: 244239.55438233944
Testing Mean Squared Error: 22898412.800904147
Training R-squared Error: 0.0017036068393896375
Testing R-squared Error: 0.143897556857024

Results for different values of n_neighbors:
n_neighbors: 3, metric: euclidean, weights: uniform
Training R-squared (mean): 0.8831
Testing R-squared (mean): 0.7597 ± 0.0372
--------------------------------------------------
n_neighbors: 3, metric: euclidean, weights: distance
Training R-squared (mean): 0.9987
Testing R-squared (mean): 0.7594 ± 0.0362
--------------------------------------------------
n_neighbors: 5, metric: euclidean, weights: uniform
Training R-squared (mean): 0.8497
Testing R-squared (mean): 0.7645 ± 0.0303
--------------------------------------------------
n_neighbors: 5, metric: euclidean, weights: distance
Training R-squared (mean): 0.9987
Testing R-squared (mean): 0.7683 ± 0.0330
--------------------------------------------------
n_neighbors: 7, metric: euclidean, weights: uniform
Training R-squared (mean): 0.8280
Testing R-squared (mean): 0.7665 ± 0.0348
```

```
n_neighbors: 7, metric: euclidean, weights: distance
Training R-squared (mean): 0.9987
Testing R-squared (mean): 0.7734 ± 0.0344
--------------------------------------------------
n_neighbors: 9, metric: euclidean, weights: uniform
Training R-squared (mean): 0.8120
Testing R-squared (mean): 0.7697 ± 0.0374
--------------------------------------------------
n_neighbors: 9, metric: euclidean, weights: distance
Training R-squared (mean): 0.9987
Testing R-squared (mean): 0.7764 ± 0.0360
--------------------------------------------------
n_neighbors: 3, metric: manhattan, weights: uniform
Training R-squared (mean): 0.8828
Testing R-squared (mean): 0.7557 ± 0.0361
--------------------------------------------------
n_neighbors: 3, metric: manhattan, weights: distance
Training R-squared (mean): 0.9987
Testing R-squared (mean): 0.7567 ± 0.0388
--------------------------------------------------
n_neighbors: 5, metric: manhattan, weights: uniform
Training R-squared (mean): 0.8448
Testing R-squared (mean): 0.7691 ± 0.0444
--------------------------------------------------
n_neighbors: 5, metric: manhattan, weights: distance
Training R-squared (mean): 0.9987
Testing R-squared (mean): 0.7701 ± 0.0433
--------------------------------------------------
n_neighbors: 7, metric: manhattan, weights: uniform
Training R-squared (mean): 0.8260
Testing R-squared (mean): 0.7710 ± 0.0390
```

```
n_neighbors: 7, metric: manhattan, weights: distance
Training R-squared (mean): 0.9987
Testing R-squared (mean): 0.7745 ± 0.0392
--------------------------------------------------
n_neighbors: 9, metric: manhattan, weights: uniform
Training R-squared (mean): 0.8097
Testing R-squared (mean): 0.7649 ± 0.0324
--------------------------------------------------
n_neighbors: 9, metric: manhattan, weights: distance
Training R-squared (mean): 0.9987
Testing R-squared (mean): 0.7735 ± 0.0342
--------------------------------------------------
```

2. Classification

```
Classification Report
              precision    recall  f1-score   support

           B       0.94      1.00      0.97       118
           M       1.00      0.90      0.95        70

    accuracy                           0.96       188
   macro avg       0.97      0.95      0.96       188
weighted avg       0.96      0.96      0.96       188


Confusion Matrix
 [[118   0]
 [  7  63]]

Accuracy Score 0.9627659574468085
```

Interpretation:

1. Regression

   KNN Performance: Best parameters found are n_neighbors: 9, metric: 'euclidean', and weights: 'distance'.

   Training vs. Testing: Training accuracy is very high (0.9983), but testing accuracy (R-squared: 0.8561) suggests potential overfitting.

   Mean Squared Error: High training (244239.55) and testing (289814.81) errors indicate that the model may not generalize well to new data.

   Refinement Needed: To improve generalization, consider adjusting model parameters or employing regularization techniques.

2. Classification

   Model Performance: High precision (0.94) and perfect recall (1.00) for benign cases (B); excellent precision (1.00) for malignant cases (M) but lower recall (0.90).

   Confusion Matrix: Correctly identifies 118 benign (TN) and 63 malignant cases (TP); misses 7 malignant cases (FN).

   Accuracy: Overall accuracy score of 0.9628, indicating strong performance in distinguishing between classes.

Improvement Needed: The model should enhance recall for malignant cases to reduce false negatives, critical for effective cancer detection.

Conclusion:

- **Model Effectiveness:** The breast cancer prediction model achieved **96.28%** accuracy and an F1-score of **0.95**, effectively identifying benign and malignant tumors. The medical insurance model reported a training R-squared of **99.83%** but only **85.61%** for testing, indicating potential overfitting.
- **Generalization Challenges:** While the breast cancer model excelled in precision and recall (1.00 for benign and malignant cases), it struggled with false negatives. The insurance model's moderate testing accuracy highlights the need for improved feature selection.
- **Clinical Implications:** The breast cancer model is a promising diagnostic tool, but enhancements are needed for detecting malignant cases. The insurance model requires better predictive accuracy for cost estimation.
- **Future Enhancements:** Both models can benefit from parameter tuning and additional features to improve performance and generalization, ensuring their practical applicability.