In [ ]:

```python
import os
import pandas as pd


data = pd.read_csv("../../data/data.csv")
```

In [ ]:

```python
data
```

Out[ ]:

	id	diagnosis	radius_mean	texture_mean	perimeter_mean	area_mean	sm
0	842302	M	17.99	10.38	122.80	1001.0	
1	842517	M	20.57	17.77	132.90	1326.0	
2	84300903	M	19.69	21.25	130.00	1203.0	
3	84348301	M	11.42	20.38	77.58	386.1	
4	84358402	M	20.29	14.34	135.10	1297.0	
...	...	...	...	...	...	...	
564	926424	M	21.56	22.39	142.00	1479.0	
565	926682	M	20.13	28.25	131.20	1261.0	
566	926954	M	16.60	28.08	108.30	858.1	
567	927241	M	20.60	29.33	140.10	1265.0	
568	92751	B	7.76	24.54	47.92	181.0	

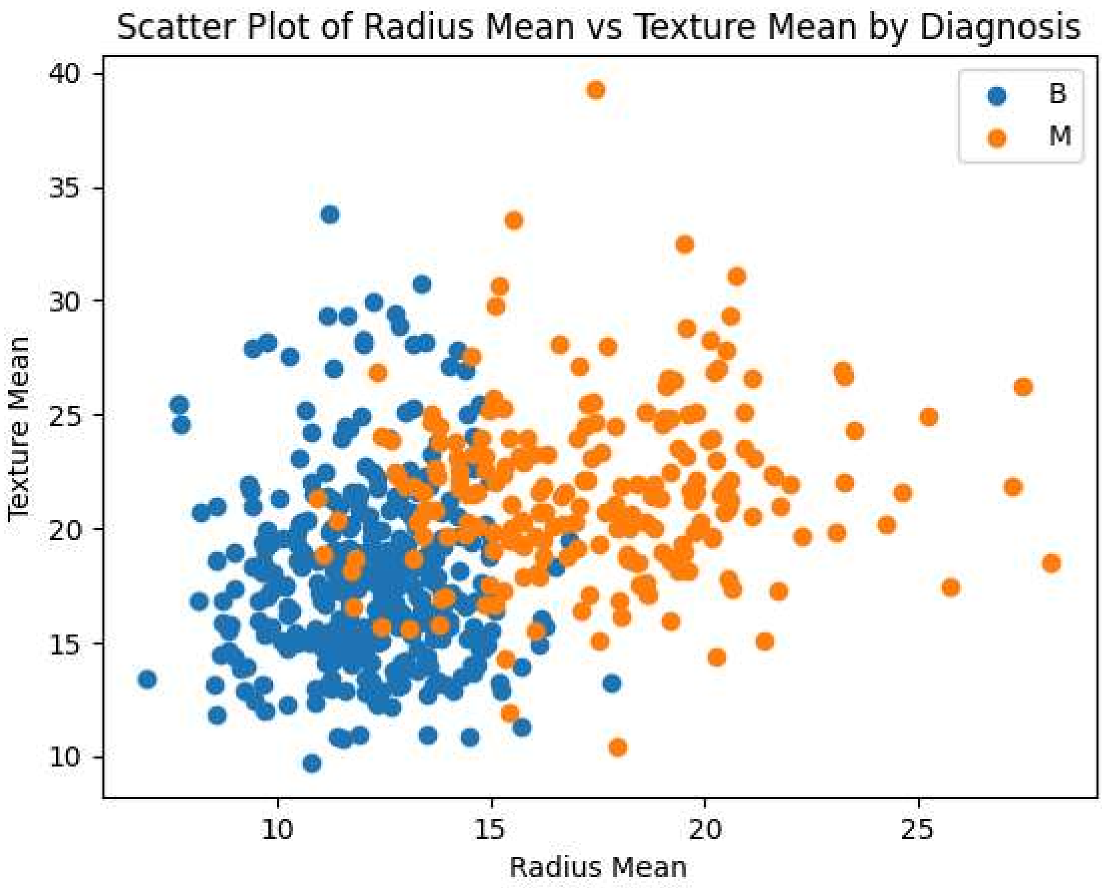
569 rows × 33 columns

In [ ]:

```python
import matplotlib.pyplot as plt

# Plotting different groups with different colors
for label, group in data.groupby('diagnosis'):
    plt.scatter(group['radius_mean'], group['texture_mean'], label=label)

plt.xlabel('Radius Mean')
plt.ylabel('Texture Mean')
plt.title('Scatter Plot of Radius Mean vs Texture Mean by Diagnosis')
plt.legend()
plt.show()
```



In [ ]:
```
from sklearn.neighbors import KNeighborsClassifier

knn = KNeighborsClassifier(n_neighbors=1)

knn.fit(data[['radius_mean', 'texture_mean']], data['diagnosis'])

# Creating new instances for prediction (supresses warning)
new_instances = pd.DataFrame({
    'radius_mean': [10, 40, 60],
    'texture_mean': [20, 50,60]
})
# Predicting classes for the given instances
predictions = knn.predict(new_instances)
predictions
```

Out[ ]:
```
array(['B', 'M', 'M'], dtype=object)
```

Evaluation

As part of an precise evaluation the following true classes were identified (gold standard). • f(i1) -> benign • f(i2) -> benign • f(i3) -> malignant

	Predicted Benign	**Predicted Malignant**
Actual Benign	True Negative (TN)	False Positive (FP)
Actual Malignant	False Negative (FN)	True Positive (TP)

Now we fill in the confusion matrix with the given true and predicted classes:

1. f(i1) -> True Negative (TN)
2. f(i2) -> False Positive (FP)
3. f(i3) -> True Positive (TP)

	Predicted Benign	**Predicted Malignant**
Actual Benign	1 (TN)	1 (FP)
Actual Malignant	0 (FN)	1 (TP)

(1+1)/(1+1+0+1) =0.67

- **Accuracy: 67%**

Manual Approach

The method should be done 'manually', but this would need 290,142 mathematic operations, taking up to 600 days of work. Therefore I used AI to develop a manual approach, that may help in understanding the algorithm better.

In [ ]:
```
import numpy as np

def euclidean_distance(point1, point2):
    """Calculate the Euclidean distance between two points."""
    return np.sqrt(np.sum((np.array(point1) - np.array(point2)) ** 2))

def k_nn_manual(X_train, y_train, query_point, k=1):
    """A simple k-NN algorithm implementation."""
    distances = []
    # Calculate distance from query point to all other points
    for idx, train_point in X_train.iterrows():
        distance = euclidean_distance(query_point, [train_point['radius_mean'],
        distances.append((distance, idx))

    # Sort distances and select the nearest k neighbors
    distances.sort()
    neighbors = distances[:k]

    # Collect the classes of these nearest neighbors
    classes = [y_train.iloc[neighbor[1]] for neighbor in neighbors]

    # Return the most common class among nearest neighbors
    return max(set(classes), key=classes.count)

# Sample Training Data
X_train_sample = data[['radius_mean', 'texture_mean']].iloc[:100]  # assuming a
y_train_sample = data['diagnosis'].iloc[:100]

# Query Points
query_points = [(10, 20), (40, 50), (60, 60)]

# Classify each query point
predictions = [k_nn_manual(X_train_sample, y_train_sample, query) for query in q
predictions
```

Out[ ]: ['B', 'M', 'M']

In [ ]:
```
import numpy as np
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from sklearn.preprocessing import LabelEncoder
from sklearn.preprocessing import StandardScaler
from tensorflow.keras.layers import Activation



# Data preparation
X = data[['radius_mean', 'texture_mean']].values
y = data['diagnosis'].values
encoder = LabelEncoder()
encoder.fit(y)
encoded_Y = encoder.transform(y)
print(encoded_Y) # check if encoding successful



# Scale features
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

'''
model = Sequential()
model.add(Dense(10, input_dim=2, activation='relu'))
model.add(Dense(1, activation='sigmoid'))

# Compile model
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy']
'''

model = Sequential()
model.add(Dense(10, input_dim=2, activation='linear'))
model.add(Dense(1, activation='linear'))

# Compile model
model.compile(loss='mse', optimizer='adam', metrics=['accuracy'])


# Fit the model
model.fit(X_scaled, encoded_Y, epochs=100, batch_size=10, verbose=0)

# Predicting classes for the new instances with scaled input
new_instances_scaled = scaler.transform(new_instances)
predictions = model.predict(new_instances_scaled)
predicted_classes = (predictions > 0.5).astype(int)
predicted_labels = encoder.inverse_transform(predicted_classes.flatten())

predicted_labels
```

```
[1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 0 1 1 1 1 1 1 1 1 0 1 0 0 0 0 0 1 1 0 1 1 0 0 0 0 1 0 1 1 0 0 0 0 1 0 1 1
 0 1 0 1 1 0 0 0 1 1 0 1 1 1 0 0 0 1 0 0 1 1 0 0 0 1 1 0 0 0 0 1 0 0 1 0 0
 0 0 0 0 0 0 1 1 1 0 1 1 0 0 0 1 1 0 1 0 1 1 0 1 1 0 0 1 0 0 1 0 0 0 0 1 0
 0 0 0 0 0 0 0 0 1 0 0 0 0 1 1 0 1 0 0 1 1 0 0 1 1 0 0 0 0 1 0 0 1 1 1 0 1
 0 1 0 0 0 1 0 0 1 1 0 1 1 1 1 0 1 1 1 0 1 0 1 0 0 1 0 1 1 1 1 0 0 1 1 0 0
 0 1 0 0 0 0 0 1 1 0 0 1 0 0 1 1 0 1 0 0 0 0 1 0 0 0 0 0 1 0 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 0 0 0 0 0 0 1 0 1 0 0 1 0 0 1 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0
 0 1 0 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 0 1 0 0 0 0 1 1 1 0 0
 0 0 1 0 1 0 1 0 0 0 1 0 0 0 0 0 0 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0 1 1 0 1 1
 1 0 1 1 0 0 0 0 0 1 0 0 0 0 0 1 0 0 0 1 0 0 1 1 0 0 0 0 0 0 1 0 0 0 0 0 0
 0 1 0 0 0 0 0 1 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 1 0 1 0 0 0 0 0 1 0 0
 1 0 1 0 0 1 0 1 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 1 0
 0 0 0 0 0 0 1 0 1 0 0 1 0 0 0 0 0 1 1 0 1 0 1 0 0 0 0 0 1 0 0 1 0 1 0 1 1
 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 1 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 1 1 1 1 1 1 0]
1/1 [==============================] - 0s 30ms/step
```

Out[ ]: `array(['B', 'M', 'M'], dtype=object)`

Manual as in example

(0.1/0.2)

H1 = (0.1 * 1.7) + (0.2 * 0.58) -1.12 = -0.834 C1(B) = (-0.834 * -1.14) + 0.22 = 1.17076 C2(M) = (-0.843 * 1.14) + 0.78 = - 1.17076 C1>C2->B

(0.4/0.5)

H1 = (0.4 * 1.7) + (0.5 * 0.58) -1.12 = -0.15 C1(B) = (-0.15 * -1.14) + 0.22 = 0.391 C2(M) = (-0.15 * 1.14) + 0.78 = 0.609 C2>C1->M

(0.6/0.6)

H1 = (0.6 * 1.7) + (0.6 * 0.58) -1.12 = 0.248 C1(B) = (0.248* -1.14) + 0.22 = -0.06272 C2(M) = (0.248 * 1.14) + 0.78 = 1.06272 C2>C1->M

from sklearn import datasets

2.2

(0.1/0.2)

Closest: [0.14/0.21] at 79°F Distance: ˆ f(i1) = (0.1 − 0.14)2 + (0.2 − 0.21)2 ≈ 0.04

> 79◦F
>
> ## (0.5/0.1)

> Closest: [0.51/0.12] at 83°F Distance: (0.5 − 0.51)2 + (0.1 − 0.12)2 ≈ 0.02
> 83◦F

(1.0/0.8)

Closest: (0.96/0.78) with 73°F Distance: (1.0 − 0.96)2 + (0.8 − 0.78)2 ≈ 0.05

> 73◦F

- **RMSE calculation**:
  - Squared errors:
    - (80 - 79)^2 = 1
    - (95 - 83)^2 = 144
    - (80 - 73)^2 = 49
  - Sum of squared errors: 1 + 144 + 49 = 194
  - Mean of squared errors: 64.67
  - RMSE: sqrt(64.67) => 8.04

RMSE => **8.04**.

2.3.3

[0.1/0.2]

H1=(0.1∗1.46)+(0.2∗−4.2)+12.69 = 11.996 y =(11.996∗5.49) +13.32 ≈ 79.2

> 79.2◦F
>
> ## [0.5/0.1]
>
> H1=(0.5∗1.46)+(0.1∗−4.2)+12.69 = 12.91 y =(12.91∗5.49) +13.32 ≈ 84.2

> 84.2◦F
>
> ## [1.0/0.8]
>
> H1=(1.0∗1.46)+(0.8∗−4.2)+12.69 = 10.7 y =(10.7∗5.49) +13.32 ≈ 72.1 72.1◦

- **RMSE**:
  - Squared errors:
    - (80 - 79.2)^2 = 0.64
    - (95 - 84.2)^2 = 116.64
    - (80 - 72.1)^2 = 62.41

- Sum of squared errors:0.64 + 116.64 + 62.41 = 179.6
- Mean of squared errors: 179.69/3=59.90
- RMSE:sqrt{59.90} => 7.74

7.74.