# Question 1: Convexity and Logistic Regression

**Q1.1**: Derive the Hessian matrix for logistic regression.

Assume we have a logistic regression

$$l(\theta) = \sum_{i=1}^{m} [y_i \, log\,(h_\theta(x_i)) + (1 - y_i)\, log\,(1 - h_\theta(x_i))]\,.$$

Assume we have a hypothesis function

$$h_\theta(x_i) = \sigma(\omega^T x_i) = \sigma(z_i) = \frac{1}{1+e^{-z_i}}.$$

The loss function is then defined as:

$$l(\omega) = \sum_{i=1}^{m} - (y_i log\, \sigma(z_i) + (1 - y_i)\, log(1 - \sigma(z_i))).$$

Noting that,

$$1 - \sigma(z_i) = 1 - 1/(1 + e^{-z}) = e^{-z}/(1 + e^{-z}) = 1/(1 + e^{z}) = \sigma(-z).$$

and, $\frac{\partial}{\partial z}\sigma(z) = \frac{\partial}{\partial z}(1 + e^{-z})^{-1} = e^{-z}(1 + e^{-z})^{-2} = \frac{1}{1+e^{-z}} \bullet \frac{e^{-z}}{1+e^{-z}} = \sigma(z)(1 - \sigma(z)).$

The Hessian of the loss function is given by $\vec{\nabla}^2 l(\omega)$.

Let $l_i(\omega) =- y_i \, log\, \sigma(z_i) - (1 - y_i)\, log\,(1 - \sigma(z_i))$.

Using chain rule,

$$\frac{\partial log\, \sigma(z_i)}{\partial \omega^T} = \frac{1}{\sigma(z_i)} \bullet \frac{\partial \sigma(z_i)}{\partial \omega^T} = \frac{1}{\sigma(z_i)} \bullet \frac{\partial \sigma(z_i)}{\partial z_i} \bullet \frac{\partial z_i}{\partial \omega^T} = (1 - \sigma(z_i))x_i \bullet \frac{\partial log\,(1-\sigma(z_i))}{\partial \omega^T} = \frac{1}{1-\sigma(z_i)} \bullet \frac{\partial(1-\sigma(z_i))}{\partial \omega^T} =- \sigma(z_i)x_i$$

Now, we can have:

$$\vec{\nabla} l_i(\omega) = \frac{\partial l_i(\omega)}{\partial \omega^T} =- y_i x_i(1 - \sigma(z_i)) + (1 - y_i)x_i\sigma(z_i) = x_i(\sigma(z_i) - y_i).$$

For the Hessian matrix with m samples, we have $\vec{\nabla}^2 l(\omega) = \sum_{i=1}^{m} x_i x_i^T \sigma(z_i)(1 - \sigma(z_i))$.

We have $\sum_{i=1}^{m} x_i x_i^T = XX^T$ and in a diagonal matrix D such that $D_{ii} = \sigma(z_i)(1 - \sigma(z_i))$.

We can conclude that: $\vec{H}(\omega) = \vec{\nabla}^2 l(\omega) = XDX^T$.

*[Hessian of Logistic Function, StackExchange, 2019]*

**Q1.2**: Show that Hessian matrix is positive definite.

**Answer:** For positive definite, we need to prove that for any non-zero vector, v, the transpose V^T multiplied by the Hessian of v is greater than zero.

$$\mathbf{v}^T H \mathbf{v} = \mathbf{v}^T \left( \sum_{i=1}^{N} p_i(1-p_i)\mathbf{x}_i \mathbf{x}_i^T \right) \mathbf{v}$$

Which simplifies to:

$$\mathbf{v}^T H \mathbf{v} = \sum_{i=1}^{N} p_i(1-p_i)(\mathbf{v}^T \mathbf{x}_i)^2$$

Since the predicted probability of the positive class is always positive i.e 0 < p < 1 (probability is always between 0 and 1) and $(\mathbf{v}^T \mathbf{x}_i)^2$ (i.e, the squared inner product) is non-negative for at least one i (when v is non-zero), therefore, the Hessian matrix H is positive definite.

**Q1.3:** Is gradient descent guaranteed to find the absolute best solution when training a logistic regression model, or could it get stuck somewhere along the way? Justify your answer based on the previous parts.

**Answer:** Based on previous parts, (i.e, proving the positive definite of H) we are guaranteed that there exists a unique solution (global minimum) that minimizes the loss function. It also ensures that gradient descent will converge to the global minimum.

# Question 2: Confusion Matrix

**Q2.1:**

```
from sklearn.datasets import fetch_openml

mnist = fetch_openml('mnist_784')
```

```
[4] X_train, X_test, y_train, y_test = X[:30000], X[30000:], y[:30000], y[30000:]
```

**Q2.2:**

Why Standard Scaler is a Good Choice for Data Preprocessing:

- Nature of Data (Pixels): Pixel values in images typically range from 0 to 255. Standardizing the data to have a mean of 0 and a standard deviation of 1 ensures that each pixel contributes equally to the computation.

- Classifier (SGD): Stochastic Gradient Descent (SGD) is sensitive to the scale of the features. Features with larger scales can dominate the gradient updates, leading to suboptimal convergence. Standardizing the features helps in faster and more reliable convergence by ensuring that all features are on a similar scale.

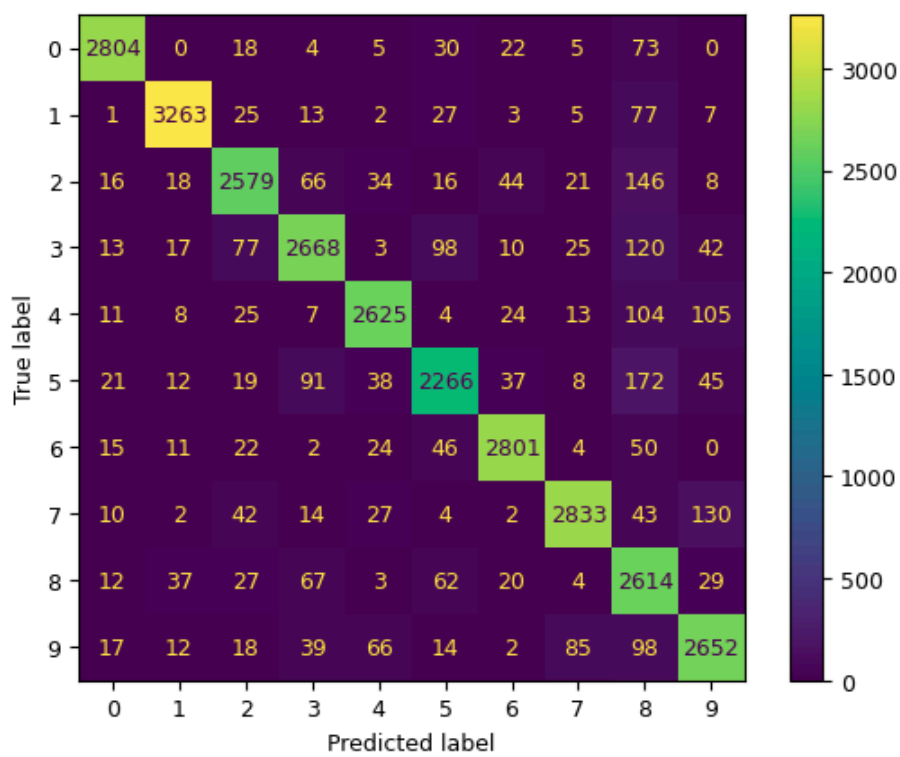
Comparison with Min-Max Scaling and No Processing:

| Min-Max Scaling | No Processing |
|---|---|
| This scaling technique transforms the data to a fixed range, usually [0, 1]. While this can be useful, it may not handle outliers well, leading to distorted scales for the majority of data points. Moreover, for algorithms like SGD, the range of the data can still affect convergence speed and stability. | Without any preprocessing, the raw pixel values (ranging from 0 to 255) can lead to large gradients and instability in the learning process, especially for algorithms like SGD. Features with larger values can dominate the learning process, leading to slower convergence and potentially poorer performance |

```
[6] from sklearn.preprocessing import StandardScaler

    scaler = StandardScaler()
    X_train_scaled = scaler.fit_transform(X_train.astype('float64'))
```

**Q2.3:**

```
from sklearn.metrics import ConfusionMatrixDisplay

y_train_pred = cross_val_predict(sgd_clf, X_train_scaled, y_train, cv=3)
plt.rc('font', size=9)
ConfusionMatrixDisplay.from_predictions(y_train, y_train_pred)
plt.show()
```




**Some notable misclassifications are:**

Digit 4 misclassified as digit 9 (85 times)
Digit 9 misclassified as digit 4 (66 times)

Digit 8 misclassified as digit 3 (67 times)
Digit 2 misclassified as digit 8 (146 times)
Digit 9 misclassified as digit 7 (98 times)
Digit 3 misclassified as digit 5 (120 times)

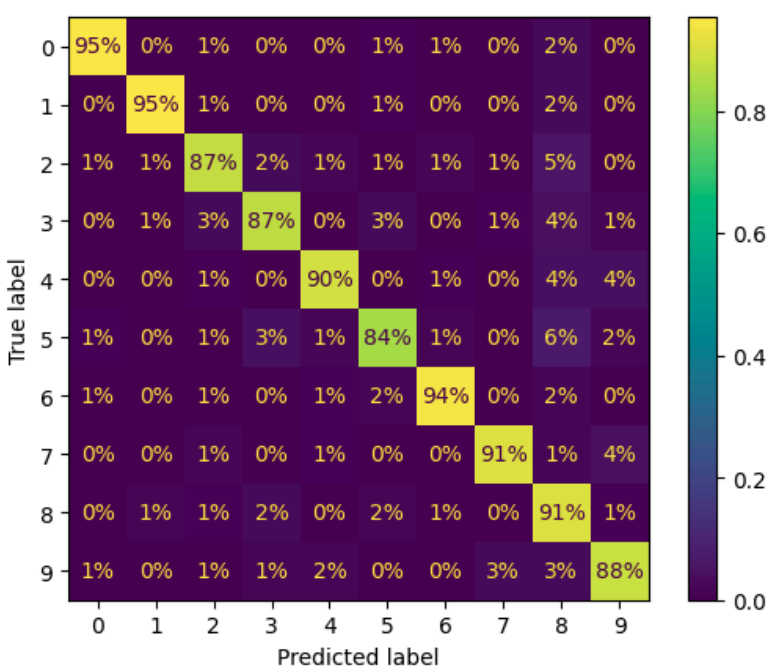
**Specific Correlations:**

There seems to be a noticeable correlation between digits 4 and 9, as they are frequently confused with each other.
Digits 8 and 3 also show a higher degree of misclassification.
Digits 2 and 8 are another pair that show a significant number of misclassifications.

**Q2.4:**

## TO DO: FILL THE BLANK LINE

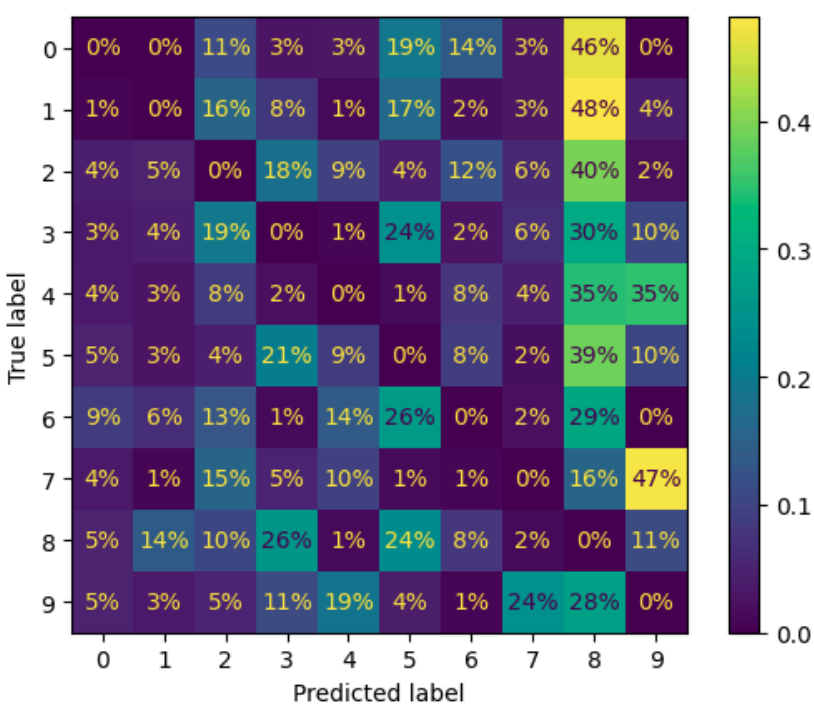
```
plt.rc('font', size=10)
ConfusionMatrixDisplay.from_predictions(
    y_train,
    y_train_pred,
    normalize='true',
    values_format=".0%",
)

plt.show()
```



Normalization divides every row by the sum of the row and converts into percentages. This will help provide an easier way to spot misclassifications.

**Q2.5:**

```
sample_weight = (y_train_pred != y_train)
plt.rc('font', size=10)
ConfusionMatrixDisplay.from_predictions(y_train, y_train_pred,
                                        sample_weight=sample_weight,
                                        normalize="true", values_format=".0%")
plt.show()
```




By introducing the sample weight, incorrect predictions become more obvious. This is useful for finding the patterns that the model is struggling with predicting. For example, the model seems to struggle with properly predicting 8s and a little bit of 9s as shown by the light colors in those columns.

# Question 3: Nearest Neighbors from Scratch

```python
def knn(data, query, k, distance_fn, choice_fn):
    neighbor_distances_and_indices = []

    # Calculate the distance between the query example and all the examples in the data
    for index, val in enumerate(data):
        distance = distance_fn(query, val[:-1])
        neighbor_distances_and_indices.append((distance, index))

    # Sort the distances and return the labels of the k nearest neighbors
    sorted_neighbors = sorted(neighbor_distances_and_indices, key=lambda x: x[0])

    # Pick the first k entries from the sorted collection
    k_nearest_distances_and_indices = sorted_neighbors[:k]

    # Get the labels of the selected k entries
    k_nearest_labels = [data[i][-1] for distance, i in k_nearest_distances_and_indices]

    # If regression (mean), if classification (mode)
    return choice_fn(k_nearest_labels)

def euclidean_distance(point1, point2):
   # function to return euclidean distance between point1 and point2
   point1 = np.array(point1)
   point2 = np.array(point2)
   square_diff_sum = np.sum((point1 - point2) ** 2)
   return np.sqrt(square_diff_sum)

def mean(labels):
   # function to return the mean of the labels.
   return sum(labels) / len(labels)

def mode(labels):
  # function to return the mode of the labels.
  label_counts = Counter(labels)
  max_count = max(label_counts.values())
  modes = [label for label, count in label_counts.items() if count == max_count]
  return modes
```

## Section II

```python
def sklearn_knn_regression(reg_data, reg_query):

    # Initialize the KNN regressor with 3 nearest neighbors
    knn_regressor = KNeighborsRegressor(n_neighbors=3)

    # Fit the model on the training data; use all but the last column as features and the last column as the target
    X_reg = reg_data[:, :-1]
    y_reg = reg_data[:, -1]
    knn_regressor.fit(X_reg, y_reg)

    # Predict the output for the provided query and return the first (and likely only) prediction
    skl_reg_prediction = knn_regressor.predict(reg_query)

    return skl_reg_prediction


def sklearn_knn_classification(clf_data, clf_query):

    # Initialize the KNN classifier with 3 nearest neighbors
    knn_classifier = KNeighborsClassifier(n_neighbors=3)

    # Fit the model on the training data; use all but the last column as features and the last column as the target
    X_clf = clf_data[:, :-1]
    y_clf = clf_data[:, -1]
    knn_classifier.fit(X_clf, y_clf)

    # Predict the class for the provided query and return the first (and likely only) prediction
    skl_clf_prediction = knn_classifier.predict(clf_query)

    return skl_clf_prediction
```

## Section III

```python
def weighted_mode(labels, weights):
    # Initialize a defaultdict to store the sum of weights for each label
    weight_sums = defaultdict(float)

    # Iterate through each label in the labels list and Sum the weights for each label and store in the defaultdict
    for label, weight in zip(labels, weights):
        weight_sums[label] += weight

    # Determine the label with the maximum sum of weights
    max_weighted_label = max(weight_sums, key=weight_sums.get)

    # Return the label that has the highest sum of weights
    return max_weighted_label

def knn_weighted(data, query, k, distance_fn, choice_fn, weights):

    # Calculate the distance between the query example and all the examples in the data.
    neighbor_distances_and_indices = []
    for index, val in enumerate(data):
        distance = distance_fn(query, val[:-1])
        neighbor_distances_and_indices.append((distance, index))

    # Sort the distances and return the labels of the k nearest neighbors
    neighbor_distances_and_indices.sort(key=lambda x: x[0])

    # Pick the first k entries from the sorted collection
    k_nearest_distances_and_indices = neighbor_distances_and_indices[:k]

    # Get the labels of the selected k entries
    k_nearest_labels = [data[i][1] for distance, i in k_nearest_distances_and_indices]

    # Get the weights (inverse of distances)
    k_nearest_distances = [distance for distance, index in k_nearest_distances_and_indices]
    weights = [1 / distance if distance != 0 else 1 for distance in k_nearest_distances]

    # Apply the weighted mode function and return the result
    return choice_fn(k_nearest_labels, weights)
```

```python
# # Load the Regression Data. The first index consists of age(feature) and the second index is the label.
data_path = '/content/drive/MyDrive/regression_data.csv'
reg_data = pd.read_csv(data_path).to_numpy()

reg_query = np.array([[55]])  # reshape to fit scikit-learn requirements

# Custom KNN Prediction
custom_reg_prediction = knn(reg_data, reg_query, k=3, distance_fn=euclidean_distance, choice_fn=mean)

# Scikit-learn KNN Regression
skl_reg_prediction = sklearn_knn_regression(reg_data, reg_query)

print("Custom KNN Regression Prediction:", custom_reg_prediction)
print("Scikit-learn KNN Regression Prediction:", skl_reg_prediction)
```

```
Custom KNN Regression Prediction: 128.24666666666667
Scikit-learn KNN Regression Prediction: [128.24666667]
```

```python
# Load the Classification Data. The first index consists of age(feature) and the second index is the label.
data_path = '/content/drive/MyDrive/classification_data.csv'
clf_data = pd.read_csv(data_path).to_numpy()

clf_query = np.array([[18]])  # reshape to fit scikit-learn requirements

# Custom KNN Prediction
custom_clf_prediction = knn(clf_data, clf_query, k=3, distance_fn=euclidean_distance, choice_fn=mode)

# Scikit-learn KNN Classification
skl_clf_prediction = sklearn_knn_classification(clf_data, clf_query)

print("Custom KNN Classification Prediction:", custom_clf_prediction)
print("Scikit-learn KNN Classification Prediction:", skl_clf_prediction)
```

```
Custom KNN Classification Prediction: [0]
Scikit-learn KNN Classification Prediction: [0]
```

```python
weights = {0: 1, 1: 2}

# Query for whether a 15-year-old likes pineapple or not. The classification should be 1 as this exact sample is present in the dataset
# but because of unbalanced dataset this will be predicted as class 0.

clf_query = np.array([[15]])  # reshape to fit scikit-learn requirements

# Custom KNN Prediction
custom_clf_prediction = knn(clf_data, clf_query, k=3, distance_fn=euclidean_distance, choice_fn=mode)

# Scikit-learn KNN Classification
skl_clf_prediction = sklearn_knn_classification(clf_data, clf_query)

print("Custom KNN Classification Prediction:", custom_clf_prediction)
print("Scikit-learn KNN Classification Prediction:", skl_clf_prediction)
```

```
Custom KNN Classification Prediction: [0]
Scikit-learn KNN Classification Prediction: [0]
```

```python
clf_prediction = knn_weighted(clf_data, clf_query, k=3, distance_fn=euclidean_distance, choice_fn=weighted_mode, weights=weights)
print("Prediction for weighted KNN:", clf_prediction)
```

```
Prediction for weighted KNN: 1
```

**Q3.6:** The KNeighborsRegressor and KNeighborsClassifier classes in Scikit-learn do not have a random_state parameter. This is because these algorithms are deterministic given a fixed set of data and query points. Since the algorithm only involves calculating distances and sorting, there is no inherent randomness in the operation of the algorithm itself. Therefore, there is no need for a random_state parameter to control any aspect of the algorithm.

In the context of KNN classification, ties can occur when there are equal numbers of neighbors from different classes within the k-nearest neighbors. For instance, if k=3 and the nearest neighbors are from classes 0, 1, and 1, class 1 would be chosen as it appears more frequently. However, if the neighbors are from classes 0, 0, 1, and 1, a tie occurs.

Scikit-learn handles ties in a deterministic way. When a tie occurs, the algorithm will choose the class that appears first in the sorted list of neighbors. This is why sorting by distances before selecting the top k neighbors ensures consistency in tie-breaking.

# Question 4: Decision Tree by Hand

Decision Tree by Hand:

For temperature:

sort:

15 cloudy
25 Sunny } 20 → Gini
28 cloudy
30 sunny } 26.5 → Gini
35 sunny } 29 → Gini
} 32.5 → Gini

Gini Impurity:

$1 - (\text{prob. of Yes})^2 - (\text{prob of NOs})^2$

We find the ~~of~~ Gini impurity for each of the averages:

$< 20$

True

| C | S |
|---|---|
| 1 | 0 |

False

| C | S |
|---|---|
| 1 | 3 |

$G_i: 1 - (0)^2 - (0)^2$

$= 0$

$1 - \left(\frac{1}{4}\right)^2 - \left(\frac{3}{4}\right)^2$

$1 - 0.0625 - 0.5625$

$= 0.375$

$< 26.5$

True

| C | S |
|---|---|
| 1 | 1 |

False

| C | S |
|---|---|
| 1 | 2 |

$1 - \left(\frac{1}{2}\right)^2 - \left(\frac{1}{2}\right)^2$

$= 0.5$

$1 - \left(\frac{1}{3}\right)^2 - \left(\frac{2}{3}\right)^2$

$= 0.442$

∴ ~~Gini~~ ~~If~~ weighted average:

$\frac{1}{5} \times 0 + \frac{4}{5} \times 0.375$

$G_{ii} = 0.3$

weighted average:

$\frac{2}{5} \times 0.5 + \frac{3}{5} \times 0.4422$

$G_{ii} = 0.465$

$\boxed{<29}$

True | False

| C | S |
|---|---|
| 2 | 1 |

| C | S |
|---|---|
| 0 | 2 |

$1 - (2/3)^2 - (1/3)^2$

$1 - (0.667)^2 - (0.33)^2$

$0.442$

$1 - 0 - 1$

$0$

Weighted average:

$3/5 \times 0.442 + 0$

$G_{(i)} = 0.265$

$\boxed{<32.5}$

True | False

| C | S |
|---|---|
| 2 | 2 |

| C | S |
|---|---|
| 0 | 1 |

$1 - (2/4)^2 - (2/4)^2$

$= 1 - (0.5)^2 - (0.5)^2$

$= 0.5$

$0$

Weighted average

$4/5 \times 0.5 + 0$

$G_{(i)} = 0.4$

CASE: Rained Yesterday:

$\boxed{\text{Rained Yesterday}}$

Yes

| C | S |
|---|---|
| 1 | 1 |

| C | S |
|---|---|
| 1 | 2 |

$1 - (1/2)^2 - (1/2)^2$

$1 - 0.25 - 0.25$

$= 0.5$

$1 - (1/3)^2 - (2/3)^2$

$1 - (0.33)^2 - (0.67)^2$

$1 - 0.1089 - 0.4489$

$= 0.4422$

Weighted average:

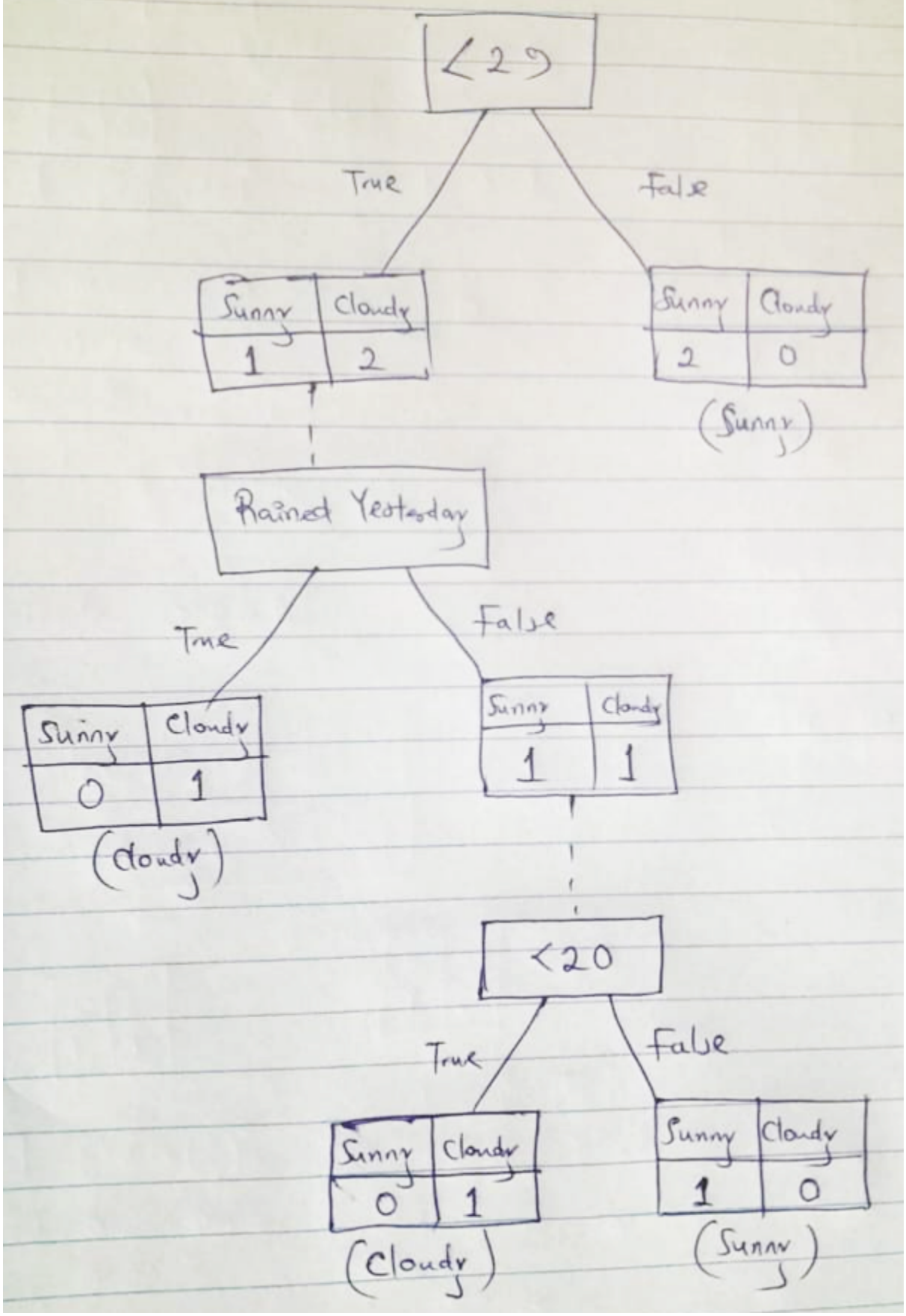
$2/5 \times 0.5 + 3/5 \times 0.4422$

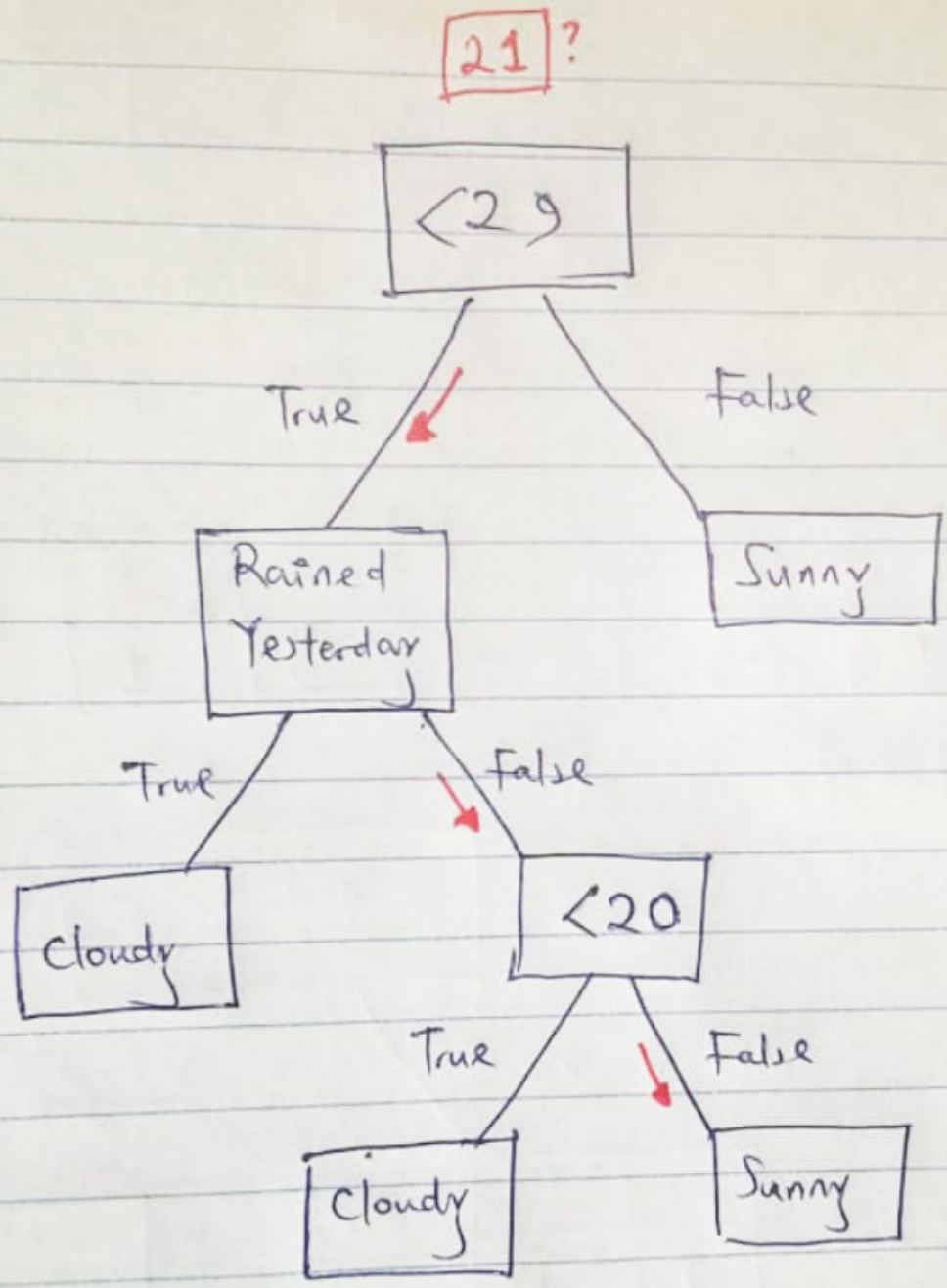
$0.2 + 0.2652$

$G_{(i)} = 0.4650$

Based on all Gini ($G_{(i)}$-$G_{(i)}$) impurities: 0.265 is the minimum, Hence, the first decision would be <29.

BEST SPLIT


<29
True
False
Sunny | Cloudy
1 | 2
Sunny | Cloudy
2 | 0
(Sunny)
Rained Yesterday
True
False
Sunny | Cloudy
0 | 1
(Cloudy)
Sunny | Cloudy
1 | 1
<20
True
False
Sunny | Cloudy
0 | 1
(Cloudy)
Sunny | Cloudy
1 | 0
(Sunny)




∴ For 21 ; it would be sunny.

# Question 5: Data Visualization and Model Selection

**Q5.1:**

## TODO: Print basic information

```python
# Load Basic information
heart_data.info()

heart_data.describe()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 549 entries, 0 to 548
Data columns (total 6 columns):
 #   Column   Non-Null Count  Dtype
---  ------   --------------  -----
 0   cp       549 non-null    int64
 1   ca       549 non-null    int64
 2   thalach  549 non-null    int64
 3   oldpeak  549 non-null    float64
 4   thal     549 non-null    int64
 5   target   549 non-null    int64
dtypes: float64(1), int64(5)
memory usage: 25.9 KB
```

| | cp | ca | thalach | oldpeak | thal | target |
|---|---|---|---|---|---|---|
| **count** | 549.000000 | 549.000000 | 549.000000 | 549.000000 | 549.000000 | 549.000000 |
| **mean** | 0.561020 | 1.078324 | 140.845173 | 1.492532 | 2.504554 | 0.094718 |
| **std** | 0.942901 | 1.039048 | 23.077379 | 1.288169 | 0.685138 | 0.293092 |
| **min** | 0.000000 | 0.000000 | 71.000000 | 0.000000 | 0.000000 | 0.000000 |
| **25%** | 0.000000 | 0.000000 | 125.000000 | 0.300000 | 2.000000 | 0.000000 |
| **50%** | 0.000000 | 1.000000 | 143.000000 | 1.200000 | 3.000000 | 0.000000 |
| **75%** | 1.000000 | 2.000000 | 160.000000 | 2.400000 | 3.000000 | 0.000000 |
| **max** | 3.000000 | 4.000000 | 195.000000 | 6.200000 | 3.000000 | 1.000000 |

**Q5.2:**

## TODO: Print the distribution of target variable

```python
value_counts = heart_data['target'].value_counts()

proportions = value_counts / len(heart_data)

print("Proportion of each unique value in the 'target' column in %:")
print(proportions * 100)
```

```
Proportion of each unique value in the 'target' column in %:
target
0    90.528233
1     9.471767
Name: count, dtype: float64
```

**Question**: Is the dataset imbalanced?

**Answer**: Yes, the dataset appears imbalanced because the proportion of value 0 (90.5%) is much higher than value 1 (9.5%)

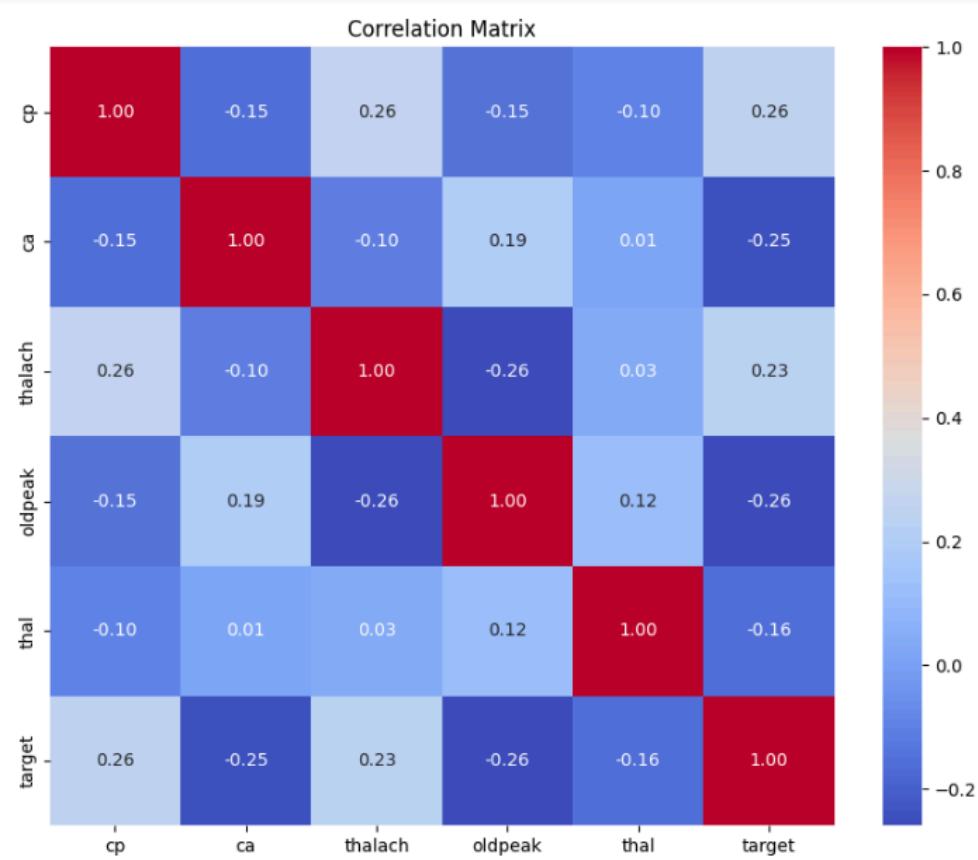
**Q5.3:**

```
cp         1.369990
ca         0.587301
thalach   -0.327264
oldpeak    0.790790
thal      -1.212341
target     2.775679
dtype: float64

Skewed Features:
 cp         1.369990
ca         0.587301
oldpeak    0.790790
thal      -1.212341
target     2.775679
dtype: float64
```

**Question:** Which of the features are skewed? Also, mention if they are left-skewed or right-skewed

**Answer: cp, ca, oldpeak & target** are **right-skewe**d while **thal** is **left-skewed**

## Q5.4:




```
Correlations with the Target Variable:
 cp         0.256380
ca         -0.246113
thalach     0.233654
oldpeak    -0.259604
thal       -0.156637
target      1.000000
Name: target, dtype: float64
```

**Question**: Is there a high correlation with features or with the target variable

**Answer**: There are no very high correlations (greater than ±0.7) either among the features themselves or between the features and the target variable. The moderate correlations (around ±0.25) with the target variable can be considered potentially important for predictive modeling but are not strong enough to indicate redundancy among the features.

## Q5.5:

**Question**: Why is this step important?

**Answer**: Since the target variable represents the outcome or response variable that we are trying to predict, separating the target column before scaling is important so that the information from the target variable doesn't affect the training process.

```python
#Report the mean and standard deviation of the scaled features
print("Mean of Scaled Features:\n", scaled_heart_data.mean())
print("\nStandard Deviation of Scaled Features:\n", scaled_heart_data.std())
```

```
Mean of Scaled Features:
 cp         2.912060e-17
ca        -2.264936e-17
thalach   -1.132468e-16
oldpeak    9.059743e-17
thal      -6.471245e-17
dtype: float64

Standard Deviation of Scaled Features:
 cp         1.000912
ca         1.000912
thalach    1.000912
oldpeak    1.000912
thal       1.000912
dtype: float64
```

**Q5.6:**

```
Shape of X_train: (439, 5)
Shape of X_test: (110, 5)
Shape of y_train: (439,)
Shape of y_test: (110,)
```

**Q5.7:**

**XGBoost_Set2** and **XGBoost_Set3** both have the highest testing accuracy (0.9636) and good performance, suggesting its best for the health data provided.

***Additional Insights:***

**KNN_Set1** : This model has high training and testing accuracy, suggesting it generalizes well. The confusion matrices indicate good performance but with some misclassifications of the minority class (class 1)

**KNN_Set2** : Slightly lower training accuracy compared to KNN_Set1, but same testing accuracy. It misclassifies more samples in the training set compared to KNN_Set1.

**KNN_Set3** : Very high training accuracy, indicating potential overfitting. Testing accuracy remains the same, but it performs slightly better on the minority class in the training set

**DecisionTree_Set1** : Moderate performance with reasonable training accuracy and slightly lower testing accuracy. Shows some misclassification, indicating it may not be capturing all patterns in the data.

**DecisionTree_Set2** : Higher training accuracy, improved performance on the minority class compared to Set1. Testing accuracy is good, but still misclassifies some minority class samples.

**DecisionTree_Set3** : High training accuracy, indicating potential overfitting. Testing accuracy is lower, with misclassifications in both classes.

**RandomForest_Set1** : Excellent performance with high training and testing accuracy. Balanced performance on minority class, showing robustness.

**RandomForest_Set2** : Similar to Set1, excellent performance with high accuracy and balanced confusion matrix.

**RandomForest_Set3** : Similar to Set1 and Set2, showing robustness across different hyperparameters.

**XGBoost_Set1** : Lower accuracy, particularly struggles with the minority class, indicating poor performance.

**XGBoost_Set2** : Excellent performance with high accuracy and better handling of the minority class compared to Set1.

**XGBoost_Set3** : Similar to Set2, excellent performance and robustness across different hyperparameters.

In summary, **XGBoost_Set2** and **XGBoost_Set3** both have the highest testing accuracy (0.9636) and good performance on the minority class. **RandomForest_Set1**, **Set2**, and **Set3** all have high testing accuracy (0.9545) and balanced performance, indicating robustness.

Contrastly, **XGBoost_Set1** has Low accuracy and poor handling of the minority class, while **DecisionTree_Set1** and **Set3** have lower testing accuracy and misclassifications.

**KNN_Set3**, **DecisionTree_Set2** might suggest potential overfitting because they shows very high training accuracy but slightly lower testing accuracy.

**Q5.8:**

Healthcare data is high-dimensional (with a large number of features) and multimodal (in different forms) - this complicates data integration and processing. Annotating such complex data, especially medical images, requires expert domain knowledge and is often time-consuming and costly.

In addition, biases might also exist in data collection, say for example, data from healthy people and unhealthy conduction the same test, or people who have genetic advantages over those who don't. These biases can lead to unfair and inaccurate models.

Lastly, publicly accessing healthcare data might pose a problem, since patient-doctor confidentiality is key. Healthcare data are not readily available for unauthorized ML enthusiasts to train and test.

# Question 6: The Curse of Dimensionality

**Q6.1:**

6.1

We have X & Y independent & uniformly distributed on [0,1]
and $Z = |X-Y|^2$

Expectation $\mathbb{E}[Z] = \mathbb{E}[|X-Y|^2]$

Doing a double integral :

$$= \int_0^1 \int_0^1 (x-y)^2 \, dx \, dy$$

Evaluating inner $\int_0^1 \Rightarrow$ $= \int_0^1 (x^2 - 2xy + y^2) \, dx$

$$= \int_0^1 x^2 dx - 2y \int_0^1 x \, dx + y^2 \int_0^1 dx$$

$$= \frac{1}{3} - y + y^2$$

Outer $\int_0^1 \Rightarrow$ $\int_0^1 \left(\frac{1}{3} - y + y^2\right) dy$

$$= \frac{1}{3}\int_0^1 dy - \int_0^1 y \, dy + \int_0^1 y^2 \, dy$$

$$= \frac{1}{3} - \frac{1}{2} + \frac{1}{3}$$

$$\boxed{\text{Expectation of } Z = \frac{1}{6}.}$$

For Variance of $z$

$$Var(z) = E[z^2] - (E[z])^2$$

$$E[z^2] = E[|X-Y|^4]$$

$$= \int_0^1 \int_0^1 (x-y)^4 \, dx \, dy$$

Inner integral =

$$\int_0^1 (x^4 - 4x^3y + 6x^2y^2 - 4xy^3 + y^4) \, dx$$

$$= \frac{1}{5} - y + 2y^2 - 2y^3 + y^4$$

outer integral = $\int_0^1 \left(\frac{1}{5} - y + 2y^2 - 2y^3 + y^4\right) dy$

$$= \frac{1}{5} - \frac{1}{2} + \frac{2}{3} - \frac{1}{2} + \frac{1}{5}$$

$$= \frac{2}{5} + \frac{2}{3} - 1$$

$$= \frac{6}{15} + \frac{10}{15} - \frac{15}{15}$$

$$= \frac{1}{15}$$

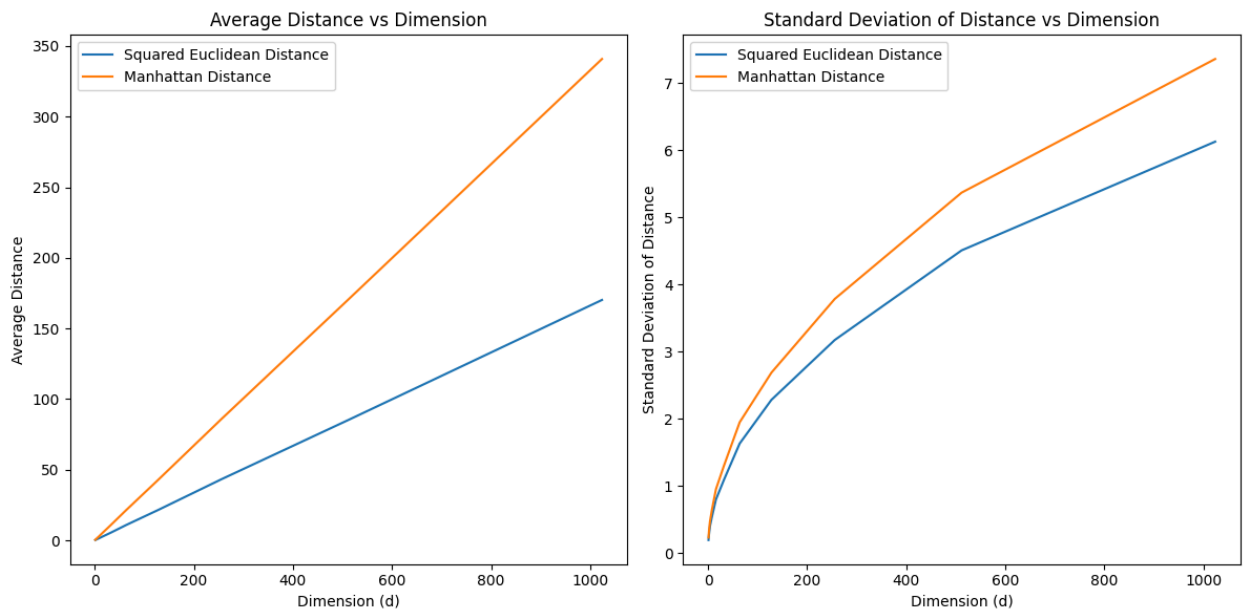
$\therefore$ Variance $(z) = E[z^2] - (E|z|)^2$

$$= \frac{1}{15} - \left(\frac{1}{6}\right)^2$$

$$= \frac{1}{15} - \frac{1}{36}$$

$$\boxed{= \frac{7}{180}}$$

**Q6.2:**




```python
import numpy as np
import matplotlib.pyplot as plt
from scipy.spatial.distance import pdist, squareform

# Define the dimensions
dimensions = [2**i for i in range(11)]
n_points = 100

# Initialize lists to store results
avg_squared_euclidean_distances = []
std_squared_euclidean_distances = []
avg_manhattan_distances = []
std_manhattan_distances = []

# Perform the calculations
for d in dimensions:
    # Sample 100 points from the unit cube [0, 1]^d
    points = np.random.rand(n_points, d)

    # Compute pairwise squared Euclidean distances
    squared_euclidean_distances = pdist(points, 'sqeuclidean')
    avg_squared_euclidean_distances.append(np.mean(squared_euclidean_distances))
    std_squared_euclidean_distances.append(np.std(squared_euclidean_distances))

    # Compute pairwise Manhattan distances
    manhattan_distances = pdist(points, 'cityblock')
    avg_manhattan_distances.append(np.mean(manhattan_distances))
    std_manhattan_distances.append(np.std(manhattan_distances))
```

```python
# Plot the results
plt.figure(figsize=(12, 6))

# Plot average distances
plt.subplot(1, 2, 1)
plt.plot(dimensions, avg_squared_euclidean_distances, label='Squared Euclidean Distance')
plt.plot(dimensions, avg_manhattan_distances, label='Manhattan Distance')
plt.xlabel('Dimension (d)')
plt.ylabel('Average Distance')
plt.title('Average Distance vs Dimension')
plt.legend()

# Plot standard deviation of distances
plt.subplot(1, 2, 2)
plt.plot(dimensions, std_squared_euclidean_distances, label='Squared Euclidean Distance')
plt.plot(dimensions, std_manhattan_distances, label='Manhattan Distance')
plt.xlabel('Dimension (d)')
plt.ylabel('Standard Deviation of Distance')
plt.title('Standard Deviation of Distance vs Dimension')
plt.legend()

plt.tight_layout()
plt.show()
```

**Q6.3:**

**Q6.4:**

**Q6.5:**

6.5

Given $K = 10$

and $n = 1000$

$$l^d \approx K/n$$

$$= \frac{10}{1000} = 0.01$$

$$l \approx (0.01)^{1/d}$$

for given $d \Rightarrow [10^0, 10^1, 10^2, 10^3, 10^4]$

for $d = 10^0$; $l \approx (0.01)^{1/1} = 0.01$

for $d = 10^1$; $l \approx (0.01)^{1/10} = 0.794$

for $d = 10^2$; $l \approx (0.01)^{1/100} = 0.955$

for $d = 10^3$; $l = (0.01)^{1/1000} = 0.998$

for $d = 10^4$; $l \approx (0.01)^{1/10000} = 0.9998$

As $d$ increases, the volume of the space increases exponentially, causing the nearest neighbors to be nearly as far away as the most distant points in lower dimensions. For very high dimensions, $l$ becomes very close to 1.

**Q6.6:**

In practice, increasing the number of samples is infeasible because the number of samples required would exponentially increase for every dimension. We then have issues of storage and longer processing times - making it less feasible in real life.

# References

1. Hessian of Logistic Function, StackExchange, 2019: https://stats.stackexchange.com/questions/68391/hessian-of-logistic-function