IFN580: Machine Learning

Assessment 2

Team Name: [Group 15]

Group No. [15]

Student Name	Student Id
Juhee Park	N11902809
Sarang Choi	N12010774
Hyojeong Chin	N12118303

	Student 1	Student 2	Student 3
Student 1	<100 %>	<100 %>	< 100%>
Student 2	<100 %>	<100 %>	<100 %>
Student 3	<100 %>	<100 %>	<100 %>

Replace the % contribution with an appropriate number if it is not an equal contribution.

Task 1: Dimensionality Reduction on Hydrogen Tweets Dataset

1.1 Outline the steps required to prepare the dataset for dimensionality reduction. Describe the preprocessing pipeline and justify the choices made.

The **hydrogen.csv** dataset requires additional preprocessing steps, which is the Boolean representation method. To convert the tweets into the binary vectors, splitting all tweets into words is necessary with the **split()** method, and a list should store the vocabulary of all unique words using **set()** and **union()** functions. Then, empty lists with the **append()** method and the syntax **if x in y:** can convert each tweet to a binary vector, which indicates the presence or absence of each word. Finally, the converted binary values can be computed by the Jaccard Coefficient to measure pairwise similarity between tweets and can also be applied to compute the **Euclidean distance** using **numpy.linalg.norm()** as well as perform dimensionality reduction.

However, the **tfidf_features.csv** dataset was selected to demonstrate the curse of dimensionality and perform dimensionality reduction, because additional preprocessing is not required. The TF-IDF vector embeddings are included as the feature values between 0 and 1 in this dataset. Therefore, it may produce more accurate results and can be directly applied to dimensionality reduction steps without preprocessing.

1.2 Demonstrate the curse of dimensionality by

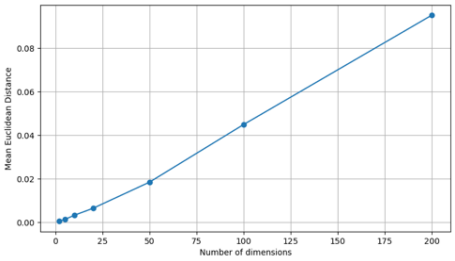
a. Incrementally increasing the number of dimensions in the feature space.

b. Visualising how the Euclidean distances between samples change as dimensionality grows.

The curse of dimensionality means issues that occur in high dimensional feature spaces because of the changes in data distance, spread, and density. The Euclidean distance can be used to demonstrate these issues in high dimensionality, since it can compute pairwise distances between all samples, and the distance results can expect the curse of dimensionality. Thus, two results show the changed tendency of Euclidean distances as dimensionality grows.

```
random_state=10
X_np = X.to_numpy()

dims = [2, 5, 10, 20, 50, 100, 200]
mean_dists = []

for d in dims:
    Xd = X_np[:, :d] # Select the first d features
    dists = []
    n = len(Xd) # Number of samples

    #Compute pairwise Euclidean distances
    for i in range(n): # Select the first sample
        for j in range(i + 1, n): # Select the second sample (bigger than i to avoid duplicates and self-comparisons)
            dist = np.linalg.norm(np.array(Xd[i]) - np.array(Xd[j])) #Euclidean Distance
            dists.append(dist)
    mean_dists.append(np.mean(dists))

mean_dists_df = pd.DataFrame({"Dimension": dims, "Mean Euclidean Distance": mean_dists})
print(mean_dists_df)
```

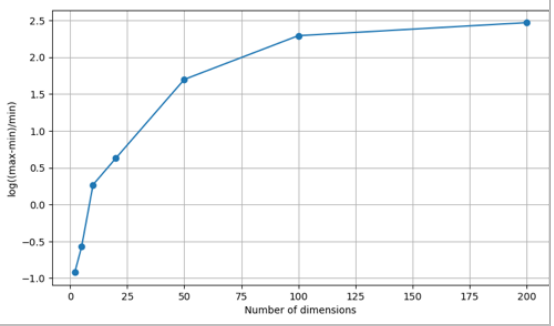
Fig 1.1 Mean Euclidean distance.



The first code computes the relationship between dimensionality and the mean Euclidean distance for the TF-IDF features. This dataset was converted to **X_np = X.to_numpy ()**, and the dimensions (**dims**) were defined from 2 to 200. The first **d** was only selected (**Xd = X_np[:, :d]**), and pairwise Euclidean distances were calculated for all upper-triangular pairs to avoid duplicates and self-comparisons (**for j in range (i + 1, n)**) using **np.linalg.norm()**. Finally, **mean_dists** stored the average of all Euclidean distances for each dimensionality. The **figure 1.1** shows a nearly linear increasing trend of mean Euclidean distance from 0.000487 to 0.095030 as dimensions grow.

```
random_state=10
X_np = X.to_numpy()

dims = [2, 5, 10, 20, 50, 100, 200]
log_dists = []

for d in dims:
    Xd = X_np[:, :d]
    n = len(Xd)

    # Initialise nearsest and farthest distances
    d_min = [float('inf')] * n # Smallest distance -> start from inf
    d_max = [0] * n # Largest distance -> start from 0

    for i in range(n):
        for j in range(i + 1, n):
            dist = np.linalg.norm(Xd[i] - Xd[j])

            # Update nearest distances
            if dist < d_min[i]: d_min[i] = dist
            if dist < d_min[j]: d_min[j] = dist

            # Update farthest distances
            if dist > d_max[i]: d_max[i] = dist
            if dist > d_max[j]: d_max[j] = dist

    d_min = np.array(d_min)
    d_max = np.array(d_max)

    valid = (d_min > 0) & (d_min < np.inf) # d_min must be > 0 except isolated, self distances and infinites
    d_min_valid = d_min[valid]
    d_max_valid = d_max[valid]

    log_dists.append(np.mean(np.log((d_max_valid - d_min_valid) / (d_min_valid))))

log_df = pd.DataFrame({"Dimension": dims, "log((max-min)/min)": log_dists})
print(log_df)
```

Fig 1.2 Distance concentration by dimensions



The logarithmic ratio between maximum and minimum Euclidean distances is calculated in this code logic. Based on the first code, the smallest and largest distances initialised for **d_min** and **d_max**, and **valid** filtered when **d_min** was 0 or **d_max** was infinite to prevent the invalid cases. Finally, computing the **log((d_max_valid – d_min_valid) / (d_min_valid))** can measure the relative contrast between the farthest and closest distances, which illustrate the pattern of distance concentration as dimensionality increases. As highlighted in the **fig 1.2**, the log values rapidly increase from 2 to 10 dimensions while it gradually grows after 100 dimensions.

c. Interpret your findings.

As dimensionality increases, the mean Euclidean distance grows approximately linearly, which demonstrates **distance inflation**, because distances between all samples generally increase. In addition, the **distance concentration** can be shown in the **figure 1.2**, which can cause all samples to have similar distances by reducing the pairwise distances between maximum and minimum. These effects illustrate the curse of dimensionality generated in high dimensionality feature spaces since the discriminative power of Euclidean distance is limited. Consequently, dimensionality reduction methods such as PCA and t-SNE help mitigate these issues by reducing dimensionality and restoring the structure of high dimensionality in low dimensions, since distance-based algorithms are difficult to distinguish between near and far samples in high feature space.

1.3 Perform dimensionality reduction using PCA

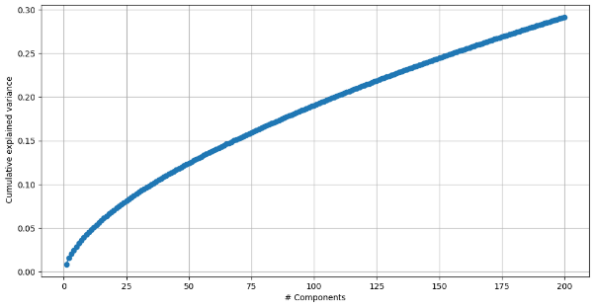
a. Determine the optimal number of principal components.

i. Explain the method you used.

ii. Include appropriate visualisations to support your answer.

```
from sklearn.decomposition import PCA
import matplotlib.pyplot as plt

comps_num = np.arange(1, 200+1)
comps_variance = []

for num_components in comps_num:
    pca = PCA(n_components=num_components, random_state=10)
    pca.fit(X.values)
    total_explained_variance = sum(pca.explained_variance_ratio_)
    comps_variance.append(total_explained_variance)

plt.figure(figsize=(12, 6))
plt.grid(True)
plt.plot(comps_num, comps_variance, marker='o')
plt.xlabel("# Components")
plt.ylabel(" Cumulative explained variance")
plt.show()
```

Fig 1.3 Cumulative explained variance



Principal Component Analysis (PCA) was applied to the TF-IDF feature matrix as a linear transformation method the high dimensionality data to low dimension spares. This method is to analyse how variance is distributed and maximised across principal components (PCs) and eliminate redundancy and lowering computational requirements. Finalising PCs can remove unnecessary features and be used to effective visualisations by reducing noise. The **sklearn.decomposition** function was used to compute the explained variance ratio to determine

the optimal number of dimensions from 1 to 200 components (**comps_num = np.arange(1, 200+1)**). The cumulative explained variance plotted to identify any elbow point, which can indicate an optimal dimensionality using **(pca.explained_variance_ratio_)**. However, the **figure 1.3** shows a gradually increasing pattern with any elbow point, and the cumulative explained variance reached only about 29% including the first 200 components. This result indicates that the variance in the TF-IDF features is spread through all dimensions due to their sparsity. Therefore, 50 components were finalised as the optimal number to achieve effective dimensionality reduction in the t-SNE step.

b. Report and interpret the total explained variance of the top 5 components.

```
top5_each = pca.explained_variance_ratio_[:5]
top5_total = sum(top5_each)

print("Top 5 each:", top5_each)
print("Top 5 explained variance:", top5_total)
```

```
Top 5 each: [0.00848973 0.0074027  0.0045459  0.00436033 0.00393847]
Top 5 explained variance: 0.028737132000712644
```

A small subset of components cannot capture most of the information, as shown by the 2.87 % the top 5 explained variance. Therefore, PCA mainly tend to reduce redundancy and prepare the data for non-linear dimensionality reduction method, which is t-SNE in the next task, since it is not sufficient for capturing non-linear relationships in TF-IDF data. Overall, the PCA results confirm that the TF-IDF data do not exhibit a low dimensional linear structure, and it supports the use of non-linear methods in subsequent analysis.

1.4 Apply t-SNE using default settings on the PCA-reduced data.

The t-Distributed Stochastic Neighbour Embedding (t-SNE) is nonlinear dimensionality reduction technique, which reduce high-dimensionality data to low-dimensionality space such as 2D or 3D keeping the similarity of the local data structure in high dimensionality.

```
from sklearn.decomposition import PCA

pca_tsne = PCA(n_components=50, random_state=10)
X_pca50 = pca_tsne.fit_transform(X.values)
print("Shape after PCA:", X_pca50.shape)
```

```
Shape after PCA: (2500, 50)
```

Although PCA was not sufficient to confirm the principal components as discussed in 1.3.b, the PCA-reduced data was used as the input for t-SNE (**pca_tsne**). This is because applying **X_pca50** helps t-SNE learn more effectively by retaining the important variance and reducing noise using the **fit_transform()** method.

a. Report the KL divergence values when reducing the dataset to 1, 2, and 3 components.

```
from sklearn.manifold import TSNE
random_state = 10

tsne_comps_num = np.arange(1, 3+1)
tsne_comps_divergence = []

for num_components in tsne_comps_num:
    tsne = TSNE(n_components=num_components, random_state=10)
    tsne.fit(X_pca50)
    tsne_comps_divergence.append(tsne.kl_divergence_)
    print(f"KL Divergence for {num_components}: {tsne.kl_divergence_}")
```

```
KL Divergence for 1: 2.082000494003296
KL Divergence for 2: 1.3181419372558594
KL Divergence for 3: 1.2136266231536865
```

The Kullback-Leibler (KL) Divergence measures the difference of two probability distributions and represents the divergence similarities between high-dimensional and low-dimensional neighbourhood. It is implemented in **sklearn.mainfold.TSNE** and computed during the **fit ()** process.

As a result, the KL divergence values were 2.08, 1.32, and 1.21 respectively for 1D, 2D, and 3D t-SNE embeddings. The reduction of the KL divergence from 1D to 2D indicates that the low-dimensional distribution becomes more similar to the original high-dimensional structure and preserve the local neighbourhood relationships. The divergence between 2D and 3D (1.214) shows a slight improvement in the 3-dimensional space. This means that t-SNE of over two dimensions sufficiently preserves a stable local structure with minimal information loss, while the 1-dimensional t-SNE

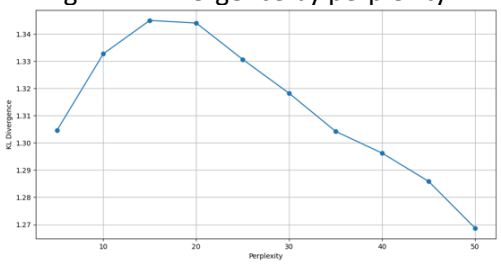
represent the greater information distortion.

b. State the perplexity value you selected. Explain how you determined its optimal value.

```
tsne_perplexity_num = np.arange(5, 50+1, 5)
tsne_perplexity_divergence = []

for perplexity in tsne_perplexity_num:
    tsne = TSNE(n_components=2, perplexity=perplexity, random_state=10)
    tsne.fit(X_pca50)
    tsne_perplexity_divergence.append(tsne.kl_divergence_)
    print(f"KL Divergence for perplexity of {perplexity}: {tsne.kl_divergence_}")
```

```
KL Divergence for perplexity of 5: 1.3044869899749756
KL Divergence for perplexity of 10: 1.3326548337936401
KL Divergence for perplexity of 15: 1.3449114561080933
KL Divergence for perplexity of 20: 1.3439373970031738
KL Divergence for perplexity of 25: 1.3305904865264893
KL Divergence for perplexity of 30: 1.3181419372558594
KL Divergence for perplexity of 35: 1.3041757345199585
KL Divergence for perplexity of 40: 1.2961950302124023
KL Divergence for perplexity of 45: 1.285843014717102
KL Divergence for perplexity of 50: 1.2686959505081177
```

Fig 1.4 K-Divergence by perplexity



The t-SNE uses the **perplexity** parameter to determine the effective number of neighbours for each sample pairs. **Perplexity** controls the balance between local and global data structure. Smaller perplexity values focus on local relationships, while larger values highlight the global structure. In this task, the perplexity parameter was arranged between 5 and 50 (**np.arrange(5, 1+50, 5)**) by keeping the t-SNE embedding in two dimensions (**n_components=2**). While the unstable KL-divergence is shown in the perplexity range between 5 and 15 by increasing the KL-divergence value, it decreases from 1.34 to 1.26 after a perplexity of 20. In particular, the perplexity between 30 and 50 have more stable KL-divergence value. This trend believes that higher perplexity values allow t-SNE to better capture global relationships without distorting local neighbourhoods. Therefore, 50 perplexity was finalised as the optimal balance between local and global structure preservation.

1.5 Comparative Visualisation

a. Visualise the top-3 components from PCA and t-SNE.

The visualisations show only one colour for all data points because the dimensionality reduction was applied to the **tfidf_features.csv** dataset, which does not include class label unlikely **hydrogen.csv** dataset. Nevertheless, these 3-dimensional scatter plots clearly demonstrate evident structural differences between PCA and t-SNE.

```
#PCA 3 Dimensionality
pca = PCA(n_components=3, random_state=10)
X_pca = pca.fit_transform(X.values)
print("Total explained variance:", sum(pca.explained_variance_ratio_))
```

```
Total explained variance: 0.02043717778654141
```

```
fig = plt.figure(1, figsize=(10, 10))
ax = fig.add_subplot(projection="3d")

scatter = ax.scatter(X_pca[:, 0], X_pca[:, 1], X_pca[:, 2], s=20)
ax.set(xlabel="PC1", ylabel="PC2", zlabel="PC3")
plt.show()
```

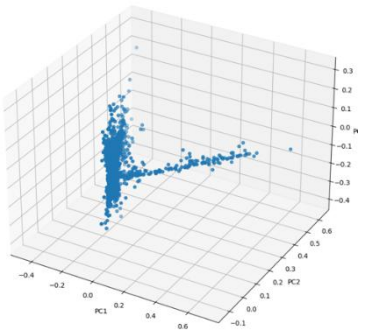


Fig 1.5 3D plot of PCA

The 3D plot includes the top 3 principal components such as PC1, PC2, and PC3. The total explained variance of these components is very low for only 2.0 %. As observed in the **fig 1.5**, the data points are linearly spread in a V-shaped pattern centred around 0. Also, it does not show a clear variance structure.

```
tsne_perplexity_num = np.arange(5, 50+1, 5)
tsne_perplexity_divergence = []

for perplexity in tsne_perplexity_num:
    tsne= TSNE(n_components=3, perplexity=perplexity, random_state=random_state)
    tsne.fit(X.values)
    tsne_perplexity_divergence.append(tsne.kl_divergence_)
    print(f"KL Divergence for perplexity of {perplexity}: {tsne.kl_divergence_}")
```

```
KL Divergence for perplexity of 5: 2.2953009605407715
KL Divergence for perplexity of 10: 2.5337114334106445
KL Divergence for perplexity of 15: 2.574293851852417
KL Divergence for perplexity of 20: 2.63037109375
KL Divergence for perplexity of 25: 2.6225898265838623
KL Divergence for perplexity of 30: 2.640794277191162
KL Divergence for perplexity of 35: 2.659586191177368
KL Divergence for perplexity of 40: 2.5672430992126465
KL Divergence for perplexity of 45: 2.5345921516418457
KL Divergence for perplexity of 50: 2.5091516971588135
```

```
#t-SNE 3 Dimensionality
tsne = TSNE(n_components=3, perplexity=50, random_state=random_state)
X_tsne = tsne.fit_transform(X.values)
print("KL Divergence with 3 components and perplexity of 50:", tsne.kl_divergence_)
```

```
KL Divergence with 3 components and perplexity of 50: 2.5091516971588135
```

```
fig = plt.figure(figsize=(10, 10))
ax = fig.add_subplot(projection="3d")
scatter = ax.scatter(X_tsne[:, 0], X_tsne[:, 1], X_tsne[:, 2], s=20)
ax.set(xlabel="C1"), ylabel="C2", zlabel="C3")
plt.show()
```

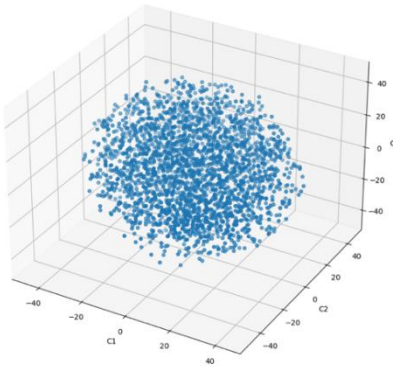


Fig 1.6 3D plot of t-SNE

Before generating the 3D t-SNE visualisation, the optimal perplexity was confirmed. As similar to the 2D case, a perplexity of 50 produced the lowest KL-divergence value for 2.51. As a result, the t-SNE visualisation (**fig 1.6**) with the optimal perplexity displays a more spherical and well-distributed clustering pattern than PCA.

b. Compare and explain the differences in feature representations generated by the two dimensionality reduction techniques.

A V-shaped pattern centred near zero is visualised in the PCA (**fig 1.5**). This is because the top three principal components explain only approximately 2 % of the total variance. In other words, PCA fails to represent the complex structure of the initial high-dimensional data. Since PCA can preserve only global linear variance, the local structure of the TF-IDF dataset tend to be largely lost. Consequently, the distances between samples become similar, and it cannot distinguish scattering or clustering patterns.

In contrast, the t-SNE clustered visualisation (**fig1.6**) shows a well-distributed spherical pattern by distributing closer with similar samples. In t-SNE dimensionality reduction, non-linear relationships can be preserved, and local neighbourhood are similarly maintained that PCA could not capture. Overall, t-SNE can perform low-dimensionality a more expressively and interpretably compared with PCA.

c. Highlight the strengths and limitations of each method in this context.

Both PCA and t-SNE have the strengths and limitations for dimensionality reduction. The PCA method provides a fast and interpretable linear transformation, although the PCA visualisation failed to represent the non-linear structure of the TF-IDF data, as it explained only a small portion of the total variance and lost local separability. In contrast, t-SNE effectively visualised the non-linear and complex structure by maintaining local similarities. Nevertheless, it has several limitations such as high computational cost, sensitivity to the perplexity parameter, and distortion of global relationships. Overall, t-SNE was more suitable for this task, as it visualised a well-distributed and interpretable clustering of the TF-IDF features.

Task 2: Clustering the Kick dataset

The Kick dataset is a refined subset of the data used in Assignment 1. It contains 41,409 observations, each representing a vehicle purchase made by the dealership.

Five key attributes relevant to the analysis are retained, while irrelevant and redundant variables have been removed for clarity. The variables are outlined in Table 2 of the assignment detail.

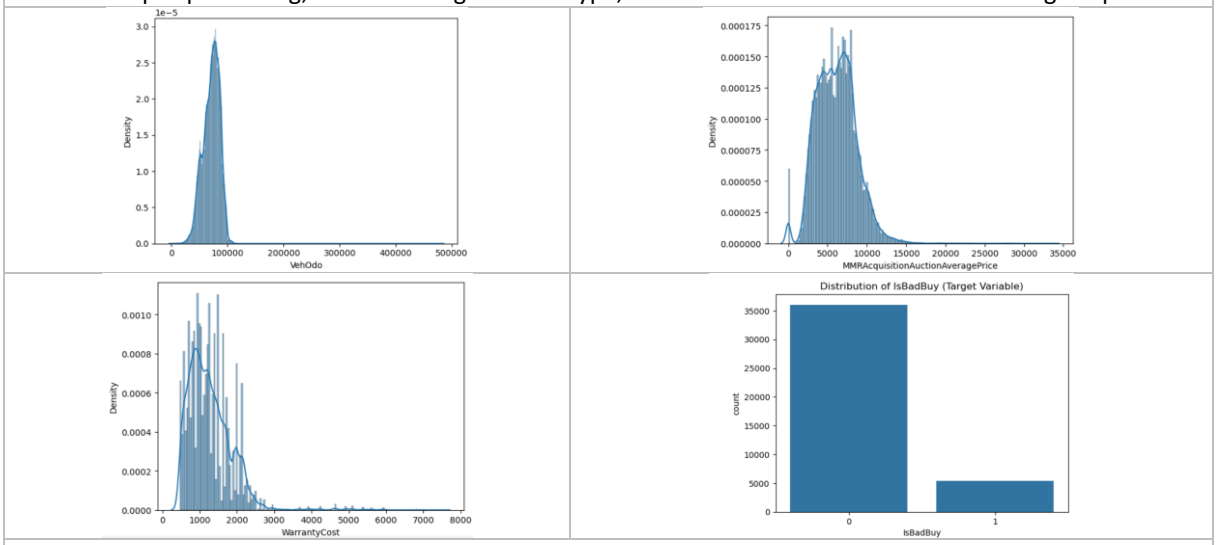
The dealership is particularly interested in profiling vehicles flagged as a kick (bad buy)- vehicles with significant issues that make them difficult or impossible to resell. To address this, perform clustering on the dataset and respond to the following tasks. Please include relevant screenshots where appropriate.

1. **Preprocessing**
   a. **Describe the preprocessing steps required before building the clustering model.**
   b. **Explain the importance of each step and justify your choices**.

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 41409 entries, 0 to 41408
Data columns (total 5 columns):
 #   Column                            Non-Null Count  Dtype
---  ------                            --------------  -----
 0   VehOdo                            41409 non-null  int64
 1   MMRAcquisitionAuctionAveragePrice  41409 non-null  int64
 2   Make                              41409 non-null  object
 3   WarrantyCost                      41409 non-null  int64
 4   IsBadBuy                          41409 non-null  int64
dtypes: int64(4), object(1)
memory usage: 1.6+ MB
```

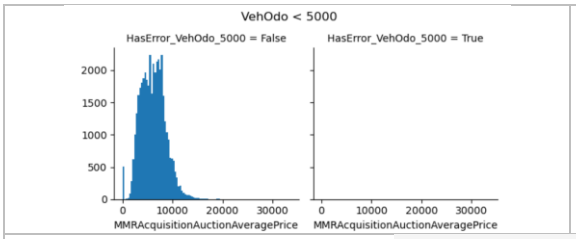
```
import seaborn as sns
import matplotlib.pyplot as plt

# Distribution of VehOdo
regdens_dist = sns.histplot(df['VehOdo'].dropna(), kde=True, stat="density",
kde_kws=dict(cut=3))
plt.show()
# Distribution of MMRAcquisitionAuctionAveragePrice
medhhinc_dist = sns.histplot(df['MMRAcquisitionAuctionAveragePrice'].dropna(), kde=True, stat="density",
kde_kws=dict(cut=3))
plt.show()
# Distribution of WarrantyCost
meanhhsz_dist = sns.histplot(df['WarrantyCost'].dropna(), kde=True, stat="density",
kde_kws=dict(cut=3))
plt.show()
#countplot of IsBadBuy(binary)
sns.countplot(x='IsBadBuy', data=df)
plt.title('Distribution of IsBadBuy (Target Variable)')
plt.show()
```

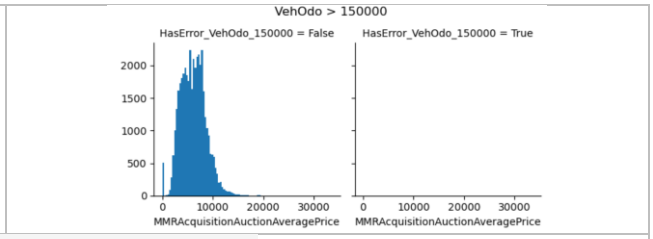
Before pre-processing, after checking the data type, the data distribution was checked using histplot.



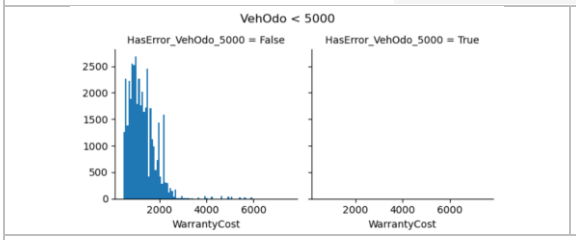
Before clustering, the following logic is used to check the correlation between data. Checking whether the outliers in the data correspond to errors or are correlated with other error values.

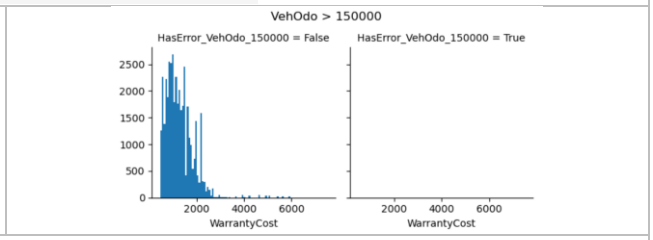
```
VehOdo_threshold = [1000,2500,5000]
for t in VehOdo_threshold:
    df[f'HasError_VehOdo_{t}'] = df['VehOdo'] < t
    g = sns.FacetGrid(df, col=f'HasError_VehOdo_{t}')
    g = g.map(plt.hist, 'MMRAcquisitionAuctionAveragePrice', bins=100)
    plt.suptitle(f"VehOdo < {t}",y=1.05)
    plt.show()

VehOdo_threshold = [110000,130000,150000]
for t in VehOdo_threshold:
    df[f'HasError_VehOdo_{t}'] = df['VehOdo'] > t
    g = sns.FacetGrid(df, col=f'HasError_VehOdo_{t}')
    g = g.map(plt.hist, 'MMRAcquisitionAuctionAveragePrice', bins=100)
    plt.suptitle(f"VehOdo > {t}",y=1.05)
    plt.show()
```






```
VehOdo_threshold = [1000,2500,5000]
for t in VehOdo_threshold:
    df[f'HasError_VehOdo_{t}'] = df['VehOdo'] < t
    g = sns.FacetGrid(df, col=f'HasError_VehOdo_{t}')
    g = g.map(plt.hist, 'WarrantyCost', bins=100)
    plt.suptitle(f"VehOdo < {t}",y=1.05)
    plt.show()

VehOdo_threshold = [110000,130000,150000]
for t in VehOdo_threshold:
    df[f'HasError_VehOdo_{t}'] = df['VehOdo'] > t
    g = sns.FacetGrid(df, col=f'HasError_VehOdo_{t}')
    g = g.map(plt.hist, 'WarrantyCost', bins=100)
    plt.suptitle(f"VehOdo > {t}",y=1.05)
    plt.show()
```



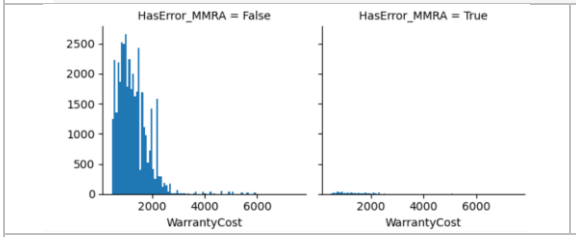


Check data quality based on VehOdo and compare the correlation distribution
1. Vehodo Low Distance Driving Criteria (1000 km, 2500 km, 5000 km)
-MMRA data distribution itself is zero or extremely small
-WarrantyCost distributed between 5000 and 3000, the same as the initial data

2. Vehodo based on High distance driving (110000 km, 1300 km, 1500,000 km)
-MMRA Data Distribution 0 or Extremely Low
-Warranty Cost data is distributed between 5000 and 3000, the same as the initial data

→ Outliers about extreme samples of VehOdo converge to almost zero. That is, there is no VehOdo error value enough to shake the distribution of MMRA and WarrantyCost, and especially for WarrantyCost, there is no significant correlation with VehOdo (The WarrantyCost does not increase significantly when the VehOdo increases). Therefore, the data quality can be trusted.

```
df['HasError_MMRA']=(df['MMRAcquisitionAuctionAveragePrice'] < 500)|(df['MMRAcquisitionAuctionAveragePrice'] > 20000)
g = sns.FacetGrid(df, col='HasError_MMRA')
g = g.map(plt.hist, 'WarrantyCost', bins=100)
plt.show()
```



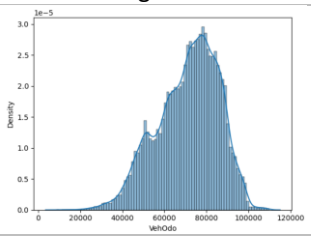
The Warranty Cost distribution between the extreme values of MMRA (less than 500, more than 20000) also shows only a few cases. With this data, abnormal patterns cannot be identified.
Therefore, the regularity between MMRA and WarrantyCost is difficult to determine

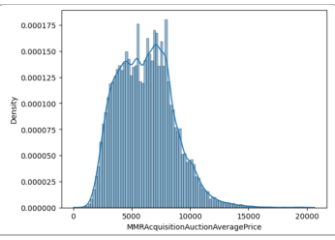
As a result of analysing the correlation between data, WarrantyCost does not have a significant correlation with VehOdo and MMRA, and it may be suspected that it is more related to other data. In addition, the data can be trusted by confirming that the outliers of VehOdo and MMRA do not significantly affect each data distribution.

```
print("Row VehOdo before dropping errorneous rows", len(df))
df = df[(df['VehOdo'] >= 5000)&(df['VehOdo'] <= 110000)]
print("Row VehOdo after dropping errorneous rows", len(df))

#MMRAcquisitionAuctionAveragePrice
print("Row MMRA before dropping errorneous rows", len(df))
df = df[(df['MMRAcquisitionAuctionAveragePrice'] >= 500)&(df['MMRAcquisitionAuctionAveragePrice'] <= 20000)]
print("Row MMRA after dropping errorneous rows", len(df))
```

```
Row VehOdo before dropping errorneous rows 41409
Row VehOdo after dropping errorneous rows 41401
Row MMRA before dropping errorneous rows 41401
Row MMRA after dropping errorneous rows 40887
```

Therefore, outliers beyond the identified range were removed, and the number of data points did not change significantly after removal, so it can be confirmed that the number of outliers itself was insignificant. Below is a histplot after removing outliers.





2. Clustering Model: using the selected attributes in Table 2 (excluding Make)

a. Specify which clustering algorithm you applied and explain why it was chosen.

b. List the attributes used in the clustering.

c. Determine the optimal number of clusters and explain the method used (e.g., elbow method, silhouette score).

d. Report the cluster centroids and interpret the characteristics of each cluster.

e. Discuss whether you applied normalisation or standardisation, and explain how this influenced the clustering results.

```
from sklearn.preprocessing import StandardScaler

# take 3 variables and drop the rest. copy the dataframe to avoid warnings later
df2 = df[['VehOdo', 'MMRAcquisitionAuctionAveragePrice', 'WarrantyCost']].copy() # convert df2 to matrix
X = df2.to_numpy()

scaler = StandardScaler()
X = scaler.fit_transform(X)
```

```
from sklearn.cluster import KMeans
for seed in [1, 5, 10, 42, 100]:
    model = KMeans(n_clusters=3, random_state=seed)
    model.fit(X)
    print(f"Seed {seed} -> Inertia: {model.inertia_}")
    print("Centroid locations:")
    for centroid in model.cluster_centers_:
        print(centroid)
```

One of the typical unsupervised learning methods is clustering, which we will use for data analysis. Therefore, IsBadBuy decides to be inappropriate for clustering to analyse random data because it is the 'ground truth' and has already been labelled. Therefore, it can be used for the extrinsic method for subsequent clustering data, but it is not suitable for the intrinsic method. Also, Make variables will be covered in Task 2.4.

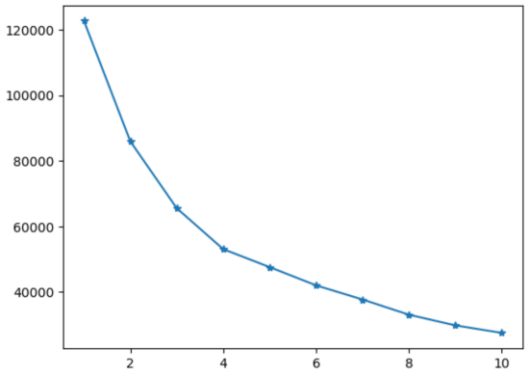
Since all three data sets to be used here are numerical variables, clustering is performed using the Kmeans algorithm. Since Kmeans uses Euclidean calculation based on distance, it is greatly affected if the units for each data are different. Therefore, Standardization was applied to prevent data distortion by unifying all data into categories between 0 and 1.

As a result of first checking the data variation according to the value of the random seed before clustering, it was confirmed that the change in the value of the inertia according to the random seed value was insignificant. Therefore, all subsequent random seeds are fixed at 42.

```
#Set a several n_clusters
for k in range(1,11):
    model = KMeans(n_clusters=k, random_state=42)
    model.fit(X)
    print("The number of cluster:", k)
    print("Sum of intra-cluster distance:", model.inertia_)
    print("Centroid locations:")
    for centroid in model.cluster_centers_:
        print(centroid)
    print("=====================================================")
```

```
The number of cluster: 1
Sum of intra-cluster distance: 122660.99999999991
Centroid locations:
[1.58508252e-17 1.63143346e-16 1.05822078e-16]
=====================================================
The number of cluster: 2
Sum of intra-cluster distance: 85910.21259236368
Centroid locations:
[0.69164676 0.01600045 0.52360581]
[-0.82581198 -0.01910421 -0.62517454]
=====================================================
The number of cluster: 3
Sum of intra-cluster distance: 65532.14714045293
Centroid locations:
[0.48212016 1.00911226 0.27442363]
[-1.01173558 -0.09671604 -0.67073662]
[ 0.66556459 -0.83221418  0.48328788]
=====================================================
The number of cluster: 4
Sum of intra-cluster distance: 53073.00696646654
Centroid locations:
[ 0.75758542 -0.31685243  1.59115073]
[-1.24896749 -0.00887809 -0.67095775]
[ 0.44248452 -0.75662446 -0.25520742]
[0.36561879 1.15625608 0.05108974]
=====================================================
The number of cluster: 5
Sum of intra-cluster distance: 47597.231430526066
Centroid locations:
[0.5366471  1.08218261 0.37098733]
[-1.05567916 -0.74387623 -0.61281472]
[ 0.61810957 -0.66875487 -0.22387297]
[-1.04101383  0.6778717  -0.62319959]
[ 0.76673054 -0.66761572  1.78341108]
=====================================================
```

```
# list to save the clusters and cost
clusters = []
inertia_vals = []
for k in range(1,11):
 # train clustering with the specified K
 model = KMeans(n_clusters=k, random_state=42)
 model.fit(X)

 # append model to cluster list
 clusters.append(model)
 inertia_vals.append(model.inertia_)

# plot the inertia vs K values
plt.plot(range(1,11), inertia_vals, marker='*')
plt.show()
```



After calculating the centroid and Intra-cluster distance according to the change in the number of K, the inertia tends to decrease as the number of K that outputs the value increases. After that, as a result of outputting the Elbow graph that visualises the inertia value, it can be seen that the reduction rate is rapidly eased at the point between the K values 2 and 4.

```
from sklearn.metrics import silhouette_score
print(clusters[1])
print("Silhouette score for k=2", silhouette_score(X, clusters[1].predict(X)))
print("=======================================")
print(clusters[2])
print("Silhouette score for k=3", silhouette_score(X, clusters[2].predict(X)))
print("=======================================")
print(clusters[3])
print("Silhouette score for k=4", silhouette_score(X, clusters[3].predict(X)))
```

```
KMeans(n_clusters=2, random_state=42)
Silhouette score for k=2 0.2819247346480283
=======================================
KMeans(n_clusters=3, random_state=42)
Silhouette score for k=3 0.293471625466365 8
=======================================
KMeans(n_clusters=4, random_state=42)
Silhouette score for k=4 0.286977196841481
```

The Elbow method has identified the point where the reduction rate is alleviated, but this method cannot prove a clear K. Therefore, by calculating the silhouette score within a specific range, we want to find the optimal K. The silhouette score proves the quality of clustering by calculating the ratio of the distance between inter-distances and intra-distances. The optimal K value in this data can be said to be 3 because it is 0.29, which is the highest at K=3 when calculating the silhouette score at K=2, 3, and 4.

```
# visualisation of K=3 clustering solution
model = KMeans(n_clusters=3, random_state=42)
model.fit(X)
# sum of intra-cluster distances
print("Sum of intra-cluster distance:", model.inertia_)
print("Centroid locations:")
for centroid in model.cluster_centers_:
 print(centroid)
```

```
Sum of intra-cluster distance: 65532.14714045293
Centroid locations:
[0.48212016 1.00911226 0.27442363]
[-1.01173558 -0.09671604 -0.67073662]
[ 0.66556459 -0.83221418  0.48328788]
```

The order of the data is VehOdo, MMRA acquisition average price, warranty cost, and since the data is standardized, the mean is 0 and the standard deviation is 1.
1st cluster: Given the higher MMRA relative to VehOdo, it can be inferred that vehicles are relatively expensive and have higher warranty costs
Second cluster: low VehOdo and MMRA, low WarrantyCosts, it can be inferred that low-value and low-burden vehicles
3rd cluster: High WarrantyCosts compared to high VehOdo and low MMRA can be inferred as high-risk vehicles

3. Using the model with the optimal number of clusters

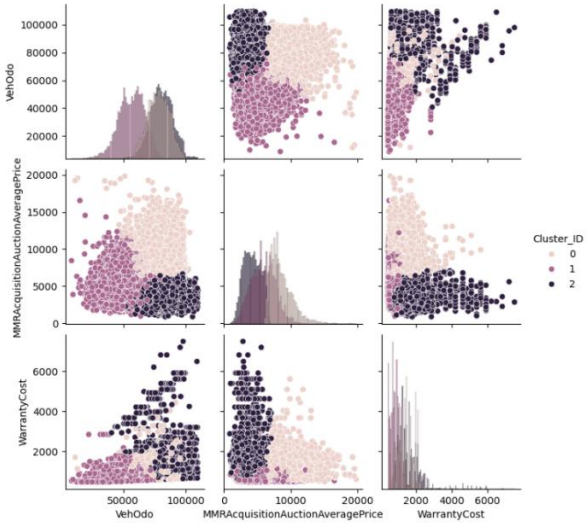
a. Visualise the clusters with a pairplot and interpret the observed patterns.

b. Assign a descriptive label to each cluster and briefly summarise the group's defining characteristics. (Hint: Use cluster distribution plots to support your explanation.)

```
y = model.predict(X)
df2['Cluster_ID'] = y
print("Cluster membership")
print(df2['Cluster_ID'].value_counts())

# pairplot
# added alpha value to assist with overlapping points
cluster_g = sns.pairplot(df2, hue='Cluster_ID', diag_kind='hist')
plt.show()
```

```
Cluster membership
Cluster_ID
1    14878
2    13455
0    12554
```

Before visualising the clustering result, the generated cluster is given an ID (labelled) and visualised as a pairplot.



Cluster_0(light pink)
: Medium VehOdo, high MMRA, low WarrantyCost
=>The price is high, but the Warranty Cost is low, which also means that the maintenance cost/risk is low. It can be seen that it is a highly valuable vehicle.
Cluster_1(pink)
: Low VehOdo, low to medium MMRA, low WarrantyCost
=>Warranty Cost is also low, so it is not a high-risk vehicle, and VehOdo and MMRA are also low, so it appears to be a low-cost model cluster
Cluster_2(purple)
: Highest VehOdo, Low MMRA, Widely Distributed WarrantyCost
Since the VehOdo is high and MMRA is low, it can be inferred that it is a low-value or old vehicle. However, since it is distributed in various ways on the Warranty Cost, it appears to be an old vehicle distributed in various ways according to the vehicle's maintenance status/risk.

4. Build a new clustering model that includes the Make variable. Use the best settings identified in Task 2 (e.g., scaling method, optimal K).

a. Indicate which clustering algorithm you applied and justify your choice.

```
df_4 = df[['VehOdo', 'MMRAcquisitionAuctionAveragePrice', 'WarrantyCost', 'Make']].copy()
print("Task 2.4. data:")
print(df_4.info())
print(df_4['Make'].value_counts())
```

```
Task 2.4. data:
<class 'pandas.core.frame.DataFrame'>
Index: 40887 entries, 0 to 41408
Data columns (total 4 columns):
 #   Column                           Non-Null Count  Dtype
---  ------                           --------------  -----
 0   VehOdo                           40887 non-null  int64
 1   MMRAcquisitionAuctionAveragePrice  40887 non-null  int64
 2   WarrantyCost                     40887 non-null  int64
 3   Make                             40887 non-null  object
dtypes: int64(3), object(1)
memory usage: 1.6+ MB
None
Make
CHEVROLET     9442
DODGE         7306
FORD          6362
CHRYSLER      5181
PONTIAC       2325
KIA           1288
SATURN        1236
NISSAN        1173
JEEP           979
HYUNDAI        953
SUZUKI         831
TOYOTA         653
MITSUBISHI     555
MAZDA          529
MERCURY        522
BUICK          411
GMC            340
HONDA          258
OLDSMOBILE     143
ISUZU           80
SCION           77
VOLKSWAGEN      72
LINCOLN         54
INFINITI        25
ACURA           19
MINI            19
SUBARU          16
CADILLAC        16
VOLVO           12
LEXUS           10
Name: count, dtype: int64
```

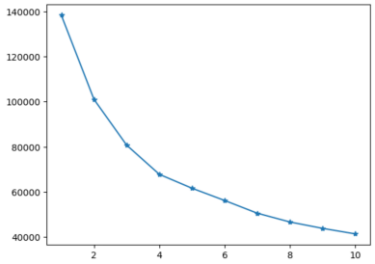
```
from sklearn.preprocessing import LabelEncoder
# convert string labels to numerical
le = LabelEncoder()
df_4['Make_encoded'] = le.fit_transform(df_4['Make'].values)
print(df_4[['Make', 'Make_encoded']].head(10))
df_4 = df_4[['VehOdo', 'MMRAcquisitionAuctionAveragePrice', 'WarrantyCost', 'Make_encoded']]
```

	Make	Make_encoded
0	DODGE	5
1	DODGE	5
2	CHRYSLER	4
3	CHEVROLET	3
4	DODGE	5
6	CHRYSLER	4
7	PONTIAC	22
8	PONTIAC	22
9	CHRYSLER	4
10	DODGE	5

```
# convert df to matrix
X = df_4.to_numpy()
# scaling
scaler = StandardScaler()
X = scaler.fit_transform(X)
```

Checking the data type. Categorical data is converted into discrete numeric variables through label encoding, and the remaining numerical data is unified between 0-1 through scaling.

```
from kmodes.kmodes import KModes
from kmodes.kprototypes import KPrototypes
# list to save the clusters and cost
clusters = []
cost_vals = []
# this process is computationally expensive and may take some time
for k in range(1,11):
    # train clustering with the specified K
    model = KPrototypes(n_clusters=k, random_state=42, n_jobs=-1)
    model.fit_predict(X, categorical=[3])

    # append model to cluster list
    clusters.append(model)
    cost_vals.append(model.cost_)
# plot the cost vs K values
plt.plot(range(1,11), cost_vals, marker='*')
plt.show()
```



The Kmeans algorithm cannot be used because the Make variable is categorical data. Therefore, this clustering uses the Kprototype algorithm that can handle both numerical and categorical variables. Kprototype cannot be calculated through inertia, but the cost (the sum of distances from each point to the center of the cluster) can be calculated. Through this logic, as a result of outputting an Elbow graph for the range of K values 1 to 10, the decrease is sharply reduced at K values 3 or 4.

```
X_num = [[row[0], row[1], row[2]] for row in X] # Variables of X with numeric datatype
X_cat = [[row[3]] for row in X] # variables of X with categorical datatype
#K=3
model = clusters[2] # cluster[1] holds the K-prtotypes model with K=3
# Calculate the Silhouette Score for the numeric and categorical variables seperately
silScoreNums = silhouette_score(X_num, model.fit_predict(X, categorical=[2]),
metric='euclidean')
print("Silscore for numeric variables:", silScoreNums)
silScoreCats = silhouette_score(X_cat, model.fit_predict(X, categorical=[2]),
metric='hamming')
print("Silscore for categorical variables:", silScoreCats)
# Average the silhouette scores
silScore = (silScoreNums + silScoreCats) / 2
print("The avg silhouette score for k=2:", silScore)


#K=4
model = clusters[3] # cluster[1] holds the K-prtotypes model with K=4
# Calculate the Silhouette Score for the numeric and categorical variables seperately
silScoreNums = silhouette_score(X_num, model.fit_predict(X, categorical=[3]),
metric='euclidean')
print("Silscore for numeric variables:", silScoreNums)
silScoreCats = silhouette_score(X_cat, model.fit_predict(X, categorical=[3]),
metric='hamming')
print("Silscore for categorical variables:", silScoreCats)
# Average the silhouette scores
silScore = (silScoreNums + silScoreCats) / 2
print("The avg silhouette score for k=4:", silScore)
```

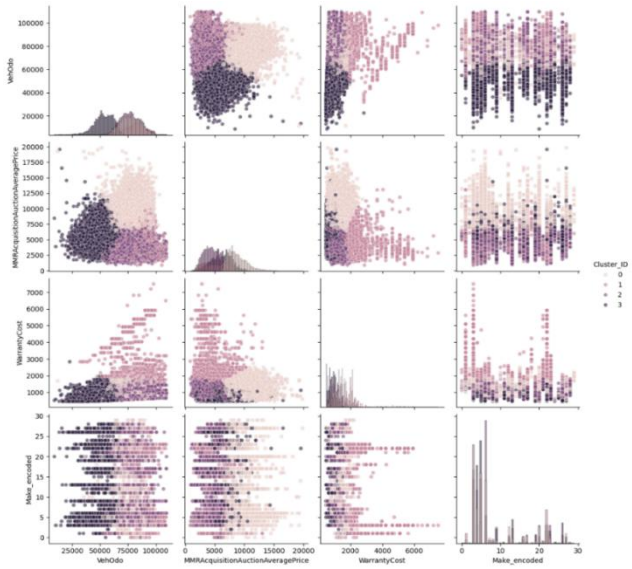
```
Silscore for numeric variables: 0.12521039343874277
Silscore for categorical variables: 0.052695189761783556
The avg silhouette score for k=2: 0.08895279160026316
Silscore for numeric variables: 0.28448032413066 3
Silscore for categorical variables: -0.044770784098887316
The avg silhouette score for k=4: 0.11985477001588783
```

Since it is a mixture of numerical and categorical variables, the silhouette score calculation method is also different from that of Kmeans. The silhouette score of a numerical variable is calculated through the Euclidean calculation method, and the silhouette score of a categorical variable is calculated through the Hamming calculation method to obtain the average of the two. As a result, it was confirmed that K=4 was the optimal value in the dataset including the Make variable.

b. Compare and contrast the interpretation of this model with the earlier model from Task 2.

```
#Optical K=4
import seaborn as sns
import matplotlib.pyplot as plt
model = clusters[3]
y=model.fit_predict(X, categorical=[3])
df_4['Cluster_ID'] = y
# how many records are in each cluster
print("Cluster membership")
print(df_4['Cluster_ID'].value_counts())
# pairplot the cluster distribution.
cluster_g = sns.pairplot(df_4, hue='Cluster_ID', diag_kind='hist',
                         height=3,
                         aspect=1,
                         plot_kws={'alpha': 0.6, 's': 30})
plt.show()
```

```
Cluster membership
Cluster_ID
2    12779
3    11340
0    10627
1     6141
Name: count, dtype: int64
```



After labelling the clusters with K=4, the optimal K found above, we visualised them as pairplots

The biggest difference from the pairplot in Task 2.3. is that one more Make variable has been added, and the cluster has been changed according to the manufacturer, and the total clusters configurations have been changed as 4. Previous models were relatively clearly classified only by VehOdo, MMRAcquisitionAveragePrice, and WarrantyCost, but this model shows more complex and detailed results. In particular, it is possible to check the distribution of WarrantyCost/MMRA according to a specific manufacturer.

1. Cluster_0: Medium drive, high middle price, low deposit, manufacturer relatively diverse
   => It is an expensive vehicle, a cluster of low-risk vehicles, and manufacturers are distributed in various ways.
2. Cluster_1: High VehOdo, low MMRA , high WarrantyCost, manufacturer also varies
   =>It is a cluster of low-cost old (high-driving) vehicles or high-risk vehicles, and it can be seen that some manufacturer vehicles have noticeably high WarrantyCost.
3. Cluster_2: VehOdo medium-high, low MMRA, low WarrantyCost
   => The VehOdo belongs to the medium or high group, but it is an economical cluster of vehicles with low risk and low MMRA. Manufacturers are also relatively diverse.
4. Cluster_3: Low VehOdo, medium MMRA, low WarrantyCost, distribution by specific brand
   =>Although the VehOdo is low and the risk is low, the MMRA is distributed halfway, and the distribution according to a specific manufacturer can be checked, indicating that it is a cluster affected by the manufacturer.

c. Explain how decision-makers in the dealership could apply the findings of this study in practical contexts.

The above results allow dealers to target specific consumer groups, i.e., the above data allows them to introduce and sell products to each consumer group.
Cluster_0 can offer customers looking for relatively new cars and reliability-focused products, Cluster_1 offers customers looking for low-cost models that are poor at driving or not used frequently. And some high-deposit manufacturers can be sold by offering additional benefits after enough explanation.
Cluster_2 can promote its products by targeting customers looking for economic models with relatively high VehOdo but low risk and cost, and Cluster_3's vehicles are manufacturer-affected clusters, so it would be more appropriate to target customers looking for specific manufacturers.

Thus, data clustering can help make promotions/sales a little easier and more efficient, and can also be a way to increase customer satisfaction with your purchase.

Task 3: Time Series Prediction using LSTM on the AgriWebb dataset

3.1. Detail the preprocessing steps required to prepare the dataset for time series forecasting

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 3691 entries, 0 to 3690
Data columns (total 10 columns):
 #   Column              Non-Null Count  Dtype
---  ------              --------------  -----
 0   PADDOCK_ID          3691 non-null   object
 1   OBSERVATION_DATE    3691 non-null   object
 2   TSDM                3691 non-null   float64
 3   15D_AVG_DAILY_RAIN  3691 non-null   float64
 4   15D_AVG_MAX_TEMP    3691 non-null   float64
 5   15D_AVG_MIN_TEMP    3691 non-null   float64
 6   15D_AVG_RH_TMAX     3691 non-null   float64
 7   15D_AVG_RH_TMIN     3691 non-null   float64
 8   15D_AVG_EVAP_SYN    3691 non-null   float64
 9   15D_AVG_RADIATION   3691 non-null   float64
dtypes: float64(8), object(2)
memory usage: 288.5+ KB
```

Table 3.1. Original dataset

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 3691 entries, 0 to 3690
Data columns (total 10 columns):
 #   Column              Non-Null Count  Dtype
---  ------              --------------  -----
 0   PADDOCK_ID          3691 non-null   object
 1   OBSERVATION_DATE    3691 non-null   datetime64[ns]
 2   TSDM                3691 non-null   float64
 3   15D_AVG_DAILY_RAIN  3691 non-null   float64
 4   15D_AVG_MAX_TEMP    3691 non-null   float64
 5   15D_AVG_MIN_TEMP    3691 non-null   float64
 6   15D_AVG_RH_TMAX     3691 non-null   float64
 7   15D_AVG_RH_TMIN     3691 non-null   float64
 8   15D_AVG_EVAP_SYN    3691 non-null   float64
 9   15D_AVG_RADIATION   3691 non-null   float64
dtypes: datetime64[ns](1), float64(8), object(1)
memory usage: 288.5+ KB
```

Table 3.2. Dataset after preprocessing

```
df.groupby("PADDOCK_ID").size()

PADDOCK_ID
352a2a92802487c32fa051f18f0fd7c53ef4b517d562a78369c7f8c679c6ac60    195
44f7088d1e8d42e054cd25c74d4bafee2059788c5ac8273a41af5d197c187ff8    195
56f79406af9de994603912c9f3a883d19502cdcd26840529370b7eb3f0044fee    193
5c4c28a9b5e43c10c32f7e2d905672fe9f9225b6ff9c22d3f228de17dfec4008    193
6b294fede7561d76c0b2592b9cfc8c131e8400860df9edc319981adb1afd1e0b    193
8191f02814bdec2922fd5287d8e55410e0fd78e2b195b47bdb05830ad71cb0cb    195
8714b98a7839f7e8e139a79e8dbff0ced6b5f68c476d58cfa80c4fee1b02f084    195
a649402b3679b6bc9ef3556484c74b3b212db6e121e11d14b530c877411de9a9    195
ab55bcd82ea3bedbf39af4eaf9b2c6387ce1d5345fa552d25a7357c10cb689ba    195
ad87a9f82ee7fb99ddfd4f63e8065925e8411b9e904cd6d4729de5cdee217fa8    195
```

Around 194 is the total sequence lengths. Therfore, we will exclude the paddocks with shorter than this lengths.

In time series prediction, datetime is an essential dataset to create a prediction model. However, **OBSERVATION_DATE** in the original dataset has an **object** data type, not **datetime** type, according to Table 3.1. The data type of this variable should be converted to a datetime and sorting by values, using `df['OBSERVATION_DATE'] = pd.to_datetime(df['OBSERVATION_DATE'])` and `df = df.sort_values(['PADDOCK_ID','OBSERVATION_DATE']).reset_index(drop=True)` codes. Furthermore, paddocks with shorter sequences must firstly be excluded before standardising sequence lengths across paddocks. We created two functions, named **create_sequences** and **data_prep**, for this task. These functions have the role for all sequences to pad the same lengths to ensure handling inputs of varying original lengths, maintaining same input dimensions.

```
# create_sequences function
def create_sequences(sequence, lookback, forecast_horizon, target_col):
    T, num_features = sequence.shape
    X, y, lengths = [], [], []
    pad_vector = np.zeros((lookback, num_features))

    # Fixed-Length lookback with pre-padding
    for t in range(1, T - forecast_horizon + 1):
        context = sequence[:t]
        if len(context) > lookback:
            context = context[-lookback:]

        padded_context = pad_vector.copy()
        padded_context[-len(context):] = context

        X.append(padded_context)
        y.append(sequence[t:t + forecast_horizon, target_col])
        lengths.append(min(len(context), lookback))

    return np.array(X), np.array(y), lengths
```

This function standardises the sequence lengths across paddocks and separate time series data to input(X) and target(y) for LSTM learning.
The codes in orange box mean if context of paddock is longer than the fixed lookback, paddock will be remained recent lookback. Additionally, paddock with shorter sequences will be replaced with zero.

```
def data_prep(df, feature_columns, lookback, test_steps, target_col):
    # prepare to store all training data
    X_all, y_all = [], []
    location_ids = [] # to track which location each sample comes from
    test_data = [] # to store test data for each location
    train_data = []
    lengths_all = []

    # Fit a global scaler
    all_train_values = []
    for _, group in df.groupby("PADDOCK_ID"):
        feature_values = group[feature_columns].values

        if len(feature_values) > lookback + test_steps:
            all_train_values.append(feature_values[:-test_steps])
    all_train_values = np.vstack(all_train_values)

    global_scaler = MinMaxScaler()
    global_scaler.fit(all_train_values)

    for location_id, group in df.groupby("PADDOCK_ID"):
        feature_values = group[feature_columns].values

        if len(feature_values) <= 194:
            continue

        # split and scale
        train_sample = global_scaler.transform(feature_values[:-test_steps])
        test_sample = global_scaler.transform(feature_values[-test_steps:])

        train_data.append((location_id, train_sample))
        test_data.append((location_id, test_sample, global_scaler))

        # prepare LSTM sequence data for training
        X_location, y_location, lengths = create_sequences(train_sample, lookback, test_steps, target_col)

        # append to the overall dataset
        X_all.append(X_location)
        y_all.append(y_location)
        lengths_all.append(lengths)

        # store location ID for tracking
        location_ids.extend([location_id] * len(y_location))

    # concatenate all locations' training data for model training
    X_all = np.concatenate(X_all, axis=0)
    y_all = np.concatenate(y_all, axis=0)
    lengths_all = np.concatenate(lengths_all, axis=0)

    X_all = X_all.reshape((X_all.shape[0], X_all.shape[1], X_all.shape[2]))

    return(torch.Tensor(X_all), torch.Tensor(y_all),
           torch.Tensor(lengths_all), train_data, test_data)
```

This function separates each location, splitting the data into train and test sets, which integrate the entire dataset.
The paddock with shorter than overall sequence lengths (194) will be excluded by codes

Moreover, the prediction models can ensure the last five timesteps of each paddock for test by defining **test_steps = 5** of **data_prep()** function.

```
# ensure the last 5 timesteps of each paddock for test: test_steps=5

lookback = 5
test_steps = 5
target_col = 0
X_5, y_5, lengths_5, train_d_5, test_d_5 = data_prep(df, ['TSDM'], lookback, test_steps, target_col)

print("Shape of input data after sequence creation:", X_5.shape)
print("Shape of targets after sequence creation:", y_5.shape)
```

```
Shape of input data after sequence creation: torch.Size([2220, 5, 1])
Shape of targets after sequence creation: torch.Size([2220, 5])
```

3.2. Build a Univariate LSTM Model (Lookback = 5, Predict = 5) – Model 1, 2

Four functions were created to build, train, and evaluate the LSTM Models conveniently.

```
class MyLSTMNet(nn.Module):
    def __init__(self, num_features, hidden_layer_size, num_layers, output_size, dropout_prob):
        super().__init__()
        self.lstm = nn.LSTM(
            input_size=num_features,
            hidden_size=hidden_layer_size,
            num_layers=num_layers,
            batch_first=True)
        self.dropout = nn.Dropout(dropout_prob)
        self.fc = nn.Linear(hidden_layer_size, output_size)

    def forward(self, data, lengths):
        packed_data = pack_padded_sequence(data, lengths.cpu(), batch_first=True, enforce_sorted=False)

        # Run through LSTM
        packed_output, (hn, cn) = self.lstm(packed_data)

        # Use the last layer's hidden state
        last_hidden = hn[-1]

        # apply dropout and final linear layer
        out = self.dropout(last_hidden)
        out = self.fc(out)
        return out
```

- MyLSTMNet(): to build the LSTM model by defining the hyperparameters of model.

```
# visualisation of train loss
def vis_train_loss(train_loss_history, val_loss_history):
    epochs = range(0, n_epochs, 100)
    plt.plot(epochs, train_loss_history, label='Training Loss')
    plt.plot(epochs, val_loss_history, label='Validation Loss')
    plt.xlabel('Epochs')
    plt.ylabel('Loss')
    plt.title('Loss Convergence')
    plt.legend()
    plt.grid()
    plt.show()
```

- vis_train_loss(): to visualise the training loss and validation loss history.

```
def pred_eval(model, X, y, lengths, train_d, test_d, lookback, target_col):
    model.eval()
    with torch.no_grad():
        train_preds = model(X, lengths)
        print("Training RMSE:", root_mean_squared_error(y.flatten().tolist(), train_preds.flatten().tolist()))
        print("Training R2:", r2_score(y.flatten().tolist(), train_preds.flatten().tolist()))

        X_test = []
        y_test = []
        lengths_test = []

        for count, (location_id, test_values, scaler) in enumerate(test_d):
            train_values = train_d[count][1]
            X_test.append(train_values[-lookback:])
            y_test.append(test_values[:, target_col])

            # append the actual lengths (just like the training phase)
            lengths_test.append(len(train_values[-lookback:]))

        X_test = torch.Tensor(np.array(X_test))
        y_test = torch.Tensor(np.array(y_test))
        lengths_test = torch.Tensor(lengths_test).long()
        test_preds = model(X_test, lengths_test)

        print("Test RMSE:", root_mean_squared_error(y_test.flatten().tolist(), test_preds.flatten().tolist()))
        print("Test R2:", r2_score(y_test.flatten().tolist(), test_preds.flatten().tolist()))

        plt.figure(figsize = (10, 6))
        plt.plot(y_test.flatten().tolist(), label="Expected Value")
        plt.plot(test_preds.flatten().tolist(), label="Predicted Value")
        plt.grid()
        plt.legend(fontsize=10)
        plt.tight_layout()
        plt.show()
```

pred_eval(): to evaluate model using RMSE and R^2 to compute the root mean square error of model's predictions.

snippet 1: train_prediction_model

```
def train_predict_model(model, n_epochs, lr, X_all, y_all, lengths, validation_split=0.2):
    batch_size = 32

    # split data into train and validation sets
    dataset = TensorDataset(X_all, y_all, lengths)
    val_size = int(len(dataset) * validation_split)
    train_size = len(dataset) - val_size
    train_set, val_set = random_split(dataset, [train_size, val_size])

    train_loader = DataLoader(train_set, batch_size=batch_size, shuffle=True)
    val_loader = DataLoader(val_set, batch_size=batch_size, shuffle=False)

    loss_fn = nn.MSELoss()
    optimizer = optim.Adam(model.parameters(), lr=lr)

    print(f"The model has {sum(p.numel() for p in model.parameters() if p.requires_grad)} trainable parameters")

    train_loss_history = []
    val_loss_history = []

    best_val_loss = float('inf')
    best_model_state = None
```

- train_predict_model(): to train model and implement prediction using epoch.

snippet 2: train_prediction_model

```
    for epoch in range(n_epochs):
        model.train()
        for X_batch, y_batch, lengths_batch in train_loader:
            y_pred = model(X_batch, lengths_batch)
            loss = loss_fn(y_pred, y_batch)
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

        # validation check every 100 epochs
        if epoch % 100 == 0:
            model.eval()
            with torch.no_grad():
                train_preds = model(X_all[train_set.indices], lengths[train_set.indices])
                train_loss = loss_fn(train_preds, y_all[train_set.indices]).item()

                val_preds = model(X_all[val_set.indices], lengths[val_set.indices])
                val_loss = loss_fn(val_preds, y_all[val_set.indices]).item()

                print(f"Epoch {epoch+1}: train loss {train_loss:.4f}, val_loss {val_loss:.4f}")

                train_loss_history.append(train_loss)
                val_loss_history.append(val_loss)

                # save best model
                if val_loss < best_val_loss:
                    best_val_loss = val_loss
                    best_model_state = model.state_dict()

    # restore best model state
    if best_model_state is not None:
        model.load_state_dict(best_model_state)

    return train_loss_history, val_loss_history, model
```

a. Report the optimal hyperparameters and describe how they were tuned.

1	```# Univariate LSTM model (lookback=5, predict=5) num_features = X_5.shape[2] hidden_layer_size = 10 output_size = test_steps num_layers = 2 dropout_prob = 0.2 model_lstm_5 = MyLSTMNet(num_features, hidden_layer_size, num_layers, output_size, dropout_prob) print(model_lstm_5) print("===") # training the Univariate LSTM Model n_epochs = 201 lr = 0.001 train_loss_history_5, val_loss_history_5, model_lstm_5 = train_predict_model(model_lstm_5, n_epochs, lr, X_5, y_5, lengths_5)```	num_features = X_5.shape[2] hidden_layer_size = **10** num_layers = **2** dropout_prob = 0.2 n_epochs = 201 lr = 0.001

2	``` # Univariate LSTM model (lookback=5, predict=5) num_features = X_5.shape[2] hidden_layer_size = 15 output_size = test_steps num_layers = 1 # to check more simply and reduce overfiting risk dropout_prob = 0.2 model_lstm_5 = MyLSTMNet(num_features, hidden_layer_size, num_layers, output_size, dropout_prob) print(model_lstm_5) print("==") # training the Univariate LSTM Model n_epochs = 201 lr = 0.001 train_loss_history_5, val_loss_history_5, model_lstm_5 = train_predict_model(model_lstm_5, n_epochs, lr, X_5, y_5, lengths_5) ```	num_features = X_5.shape[2] hidden_layer_size = **15** num_layers = **1** dropout_prob = 0.2 n_epochs = 201 lr = 0.001

Model 1 utilises 10 hidden layers and 2 layers of LSTM, whereas the hidden layer was increased to 15 layers and LSTM layer is reduced to 1 layer in model 2 to improve representation and to reduce overfitting risk.

b. State the training and testing accuracies.

The training and testing accuracies were evaluated using RMSE and R^2 since this model addresses time series data and predicts a numeric value.

- **RMSE**: the average magnitude of prediction errors. Lower RMSE indicates higher accuracy of prediction.
- $\boldsymbol{R^2}$: how much of the variance in the target variable. Higher R^2 value means a better model.

Model 1	Model 2
MyLSTMNet((lstm): LSTM(1, 10, num_layers=2, batch_first=True) (dropout): Dropout(p=0.2, inplace=False) (fc): Linear(in_features=10, out_features=5, bias=True)) == The model has 1455 trainable parameters Epoch 1: train loss 0.0685, val_loss 0.0735 Epoch 101: train loss 0.0229, val_loss 0.0233 Epoch 201: train loss 0.0226, val_loss 0.0232	MyLSTMNet((lstm): LSTM(1, 15, batch_first=True) (dropout): Dropout(p=0.2, inplace=False) (fc): Linear(in_features=15, out_features=5, bias=True)) == The model has 1160 trainable parameters Epoch 1: train loss 0.0522, val_loss 0.0512 Epoch 101: train loss 0.0252, val_loss 0.0237 Epoch 201: train loss 0.0232, val_loss 0.0218

Univariate LSML model	RMSE(Training)	R2(Training)	RMSE(Test)	R2(Test)
1	0.1506	0.5225	0.19995	-0.0349
2	0.1515	0.5175	0.1781	0.1791

The model 1 has stable training accuracy, but R-squared of test is extremely low (-0.0349). This result means overfitting and poor generalization. Model 2 shows improved training and testing accuracy, increasing R^2 and decreasing RMSE compared to Model 1. Therefore, Model 2 has a more optimal hyperparameter than Model 1.

c. Plot and interpret the loss function convergence during training

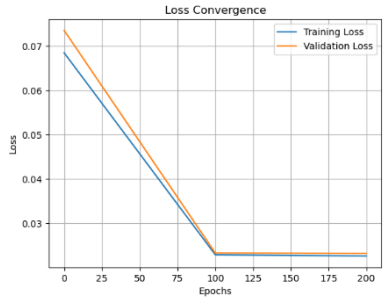


Fig 3.1. loss convergence of model 1

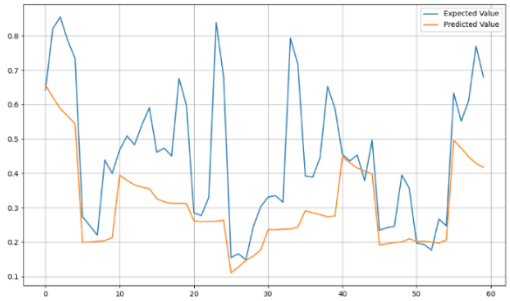


Fig 3.2. Prediction plot of model 1

According to Fig 3.1, both training and validation loss decrease and stabilise around epoch 100. Furthermore, Fig 3.2. shows comparison between expected and predicted values on test dataset using RMSE and R^2. This prediction plot indicates that the predicted value cannot capture the expected value. In addition, the general trend is being followed, but this model cannot represent extreme changes in values.

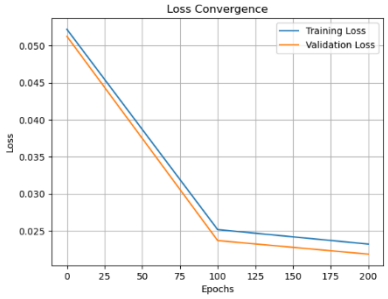


Fig 3.3. loss convergence of model 2

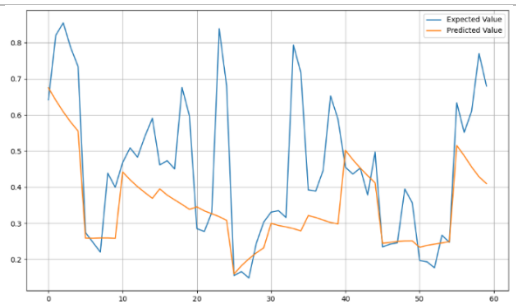


Fig 3.4. Prediction plot of model 2

Training loss and validation loss of model 2 also started to saturate approximately epoch 100. However, the loss value of model 2 is lower than that of model 1. Additionally, Fig 3.4. also shows that this model can follow the general trend but cannot capture extreme values.

The model cannot accurately predict many of the more extreme values because there are several reasons related to model architecture, the data, and the usage of only a single variable.

3.3. Build another Univariate LSTM Model (Lookback = 10, Predict = 5) – Model 3

a. Describe changes made to the model architecture to handle the larger lookback.

```
# Univariate LSTM Model (lookback=10, Predict=5)
lookback = 10
test_steps = 5
target_col = 0
X_10, y_10, lengths_10, train_d_10, test_d_10 = data_prep(df, ['TSOM'], lookback, test_steps, target_col)

print("Shape of input data after sequence creation:", X_10.shape)
print("Shape of targets after sequence creation:", y_10.shape)
print("=============================================================")

num_features = X_10.shape[2]
hidden_layer_size = 15
output_size = test_steps
num_layers = 2
dropout_prob = 0.2 # bigger number can make underfitting.
model_lstm_10 = MyLSTMNet(num_features, hidden_layer_size, num_layers, output_size, dropout_prob)

print(model_lstm_10)
print("=============================================================")

n_epochs = 201
lr = 0.001
train_loss_history_10, val_loss_history_10, model_lstm_10 = train_predict_model(model_lstm_10, n_epochs, lr, X_10, y_10, lengths_10)
print( )
vis_train_loss(train_loss_history_10, val_loss_history_10)

print( )
lookback = 10
target_col = 0
pred_eval(model_lstm_10, X_10, y_10, lengths_10, train_d_10, test_d_10, lookback, target_col)
```

We set the hyperparameter:
Lookback = 10
num_features = X_10.shape[2]
hidden_layer_size = 15
num_layers = 2
dropout_prob = 0.2
n_epochs = 201
lr = 0.001

b. Report the training and testing accuracies.

```
MyLSTMNet(
  (lstm): LSTM(1, 15, num_layers=2, batch_first=True)
  (dropout): Dropout(p=0.2, inplace=False)
  (fc): Linear(in_features=15, out_features=5, bias=True)
)
=============================================================
The model has 3080 trainable parameters
Epoch 1: train loss 0.0466, val_loss 0.0457
Epoch 101: train loss 0.0223, val_loss 0.0231
Epoch 201: train loss 0.0192, val_loss 0.0214
```

The training set has high accuracy (0.5872), but testing set shows low accuracy (0.3614), evaluating by RMSE, and R^2. This means low prediction performance and overfitting. The model can learn more data, but it can decrease performance because when lookback window is increased, model complexity and noise can increase.

RMSE(Training)	R2(Training)	RMSE(Test)	R2(Test)
0.1401	0.5872	0.1571	0.3614

c. Visualise and interpret the training loss convergence

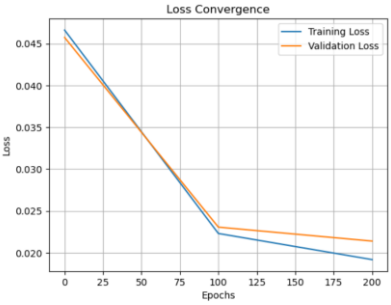


Fig 3.5. Loss convergence of model 3

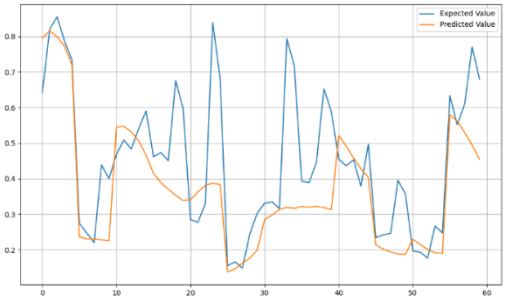


Fig 3.6. Prediction plot of model 3

Fig 3.5. shows that the gap between the training loss and validation loss increases. This situation may lead to overfitting. According to Fig 3.6, this graph shows that the model can predict the values following general trend, but still fails to predict the extreme values.

3.4. Build a Multivariate LSTM Model (TSDM + climate attributes) – Model 4

a. Describe the network architecture and modifications required to handle multiple inputs and flexible lookbacks.

```python
def create_sequences(sequence, lookback, forecast_horizon, target_col, pad_value=0.0):
    T, num_features = sequence.shape
    X, y, lengths = [], [], []

    if lookback > 0:
        pad_vector = np.zeros((lookback, num_features))

        for t in range(1, T - forecast_horizon + 1):
            context = sequence[:t]
            if len(context) > lookback:
                context = context[-lookback:]
            elif len(context) == 0:
                continue  # to resolve null context problem

            padded_context = pad_vector.copy()
            padded_context[-len(context):] = context

            X.append(padded_context)
            y.append(sequence[t:t + forecast_horizon, target_col])
            lengths.append(min(len(context), lookback))

        return np.array(X), np.array(y), lengths
    else:
        for t in range(1, T - forecast_horizon + 1):
            context = torch.tensor(sequence[:t], dtype=torch.float32)

            lengths.append(t)

            X.append(context) # No manual padding
            y.append(torch.tensor(sequence[t:t + forecast_horizon, target_col], dtype=torch.float32))

        X_padded = pad_sequence(X, batch_first=True, padding_value=pad_value)
        y_tensor = torch.stack(y)

        return X_padded.numpy(), y_tensor.numpy(), lengths
```

- For handling flexible lookbacks (no restriction), create_sequences() should be modified to separate the case which the lookback window is 0.
- If lookback was set to 0, this function uses the full history to each time step.
- Furthermore, when lookback is zero, each paddock has different sequence lengths. Thus, we use the pad_sequence() of PyTorch to match the sequence lengths.
- After that, pad_sequence return the tensor, which should be changed to numpy() for matching.

The multivariate LSTM Model, which added TSDM and climate attributes, was built utilising these functions.

```python
lookback = 0 # mean no restriction of lookback
test_steps = 5
target_col = 0

climate_features = ['TSDM','15D_AVG_DAILY_RAIN', '15D_AVG_MAX_TEMP', '15D_AVG_MIN_TEMP',
                    '15D_AVG_RH_TMAX', '15D_AVG_RH_TMIN','15D_AVG_EVAP_SYN', '15D_AVG_RADIATION']

X_f, y_f, lengths_f, train_d_f, test_d_f = data_prep(df, climate_features, lookback, test_steps, target_col)

print("Shape of input data after sequence creation:", X_f.shape)
print("Shape of targets after sequence creation:", y_f.shape)
print("============================================================")

num_features = X_f.shape[2]
hidden_layer_size = 20 # because of multivariate - need to increase hidden layer size
output_size = test_steps
n_epochs = 201 # to reduce running time
lr = 0.001
num_layers = 2
dropout_prob = 0.2

model_lstm_f = MyLSTMNet(num_features, hidden_layer_size, num_layers, output_size, dropout_prob)
print(model_lstm_f)
print("============================================================")

train_loss_history_f, val_loss_history_f, model_lstm_f = train_predict_model(model_lstm_f, n_epochs, lr, X_f, y_f, lengths_f)
print( )
vis_train_loss(train_loss_history_f, val_loss_history_f)
print( )
pred_eval(model_lstm_f, X_f, y_f, lengths_f, train_d_f, test_d_f, lookback, target_col)
```

lookback = 0 (for flexible lookback setting)
test_steps = 5
target_col = 0
climate_features variable: a list of eight features per time step.
num_features = X_f.shape[2]
hidden_layer_size = 20

- The size of hidden layer increased because of the increase in feature variables.

n_epochs = 201
lr = 0.001
num_layers = 2
dropout_prob = 0.2

- To reduce the risk of overfitting

```
MyLSTMNet(
  (lstm): LSTM(8, 20, num_layers=2, batch_first=True)
  (dropout): Dropout(p=0.2, inplace=False)
  (fc): Linear(in_features=20, out_features=5, bias=True)
)
============================================================
The model has 5865 trainable parameters
Epoch 1: train loss 0.0446, val_loss 0.0417
Epoch 101: train loss 0.0063, val_loss 0.0072
Epoch 201: train loss 0.0036, val_loss 0.0048
```

b. Visualise and interpret the loss function convergence.

RMSE(Training)	R2(Training)	RMSE(Test)	R2(Test)
0.0622	0.9185	0.1856	0.1083

This table indicates that the multivariate model has high accuracy in training set (RMSE = 0.06224, R2 = 0.9185). In contrast, there is lower accuracy (RMSE = 0.1856, R2 = 0.1083) in the testing set, which lead to overfitting.

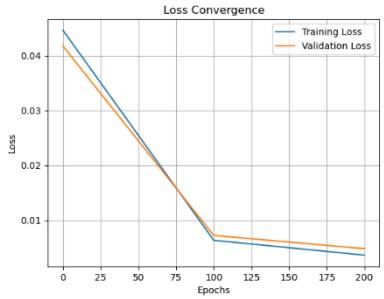


Fig 3.7. loss convergence of model 4

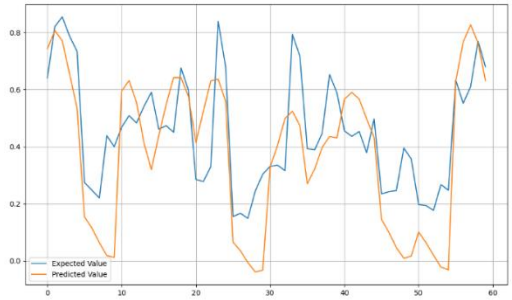


Fig 3.8. prediction plot of model 4

Fig 3.7. shows that training loss and validation loss stably converged around epoch 100. The gap between training and validation loss was small, so this situation can mean that training and testing data have consistent learning performance. However, after around 85 epoch, validation loss is more than training loss, which indicates overfitting tendency. Predicted values followed the general trend, and the model predicts the extreme value as more small values in Fig 3.8. Thus, extreme value cannot be predicted accurately.

3.5. Compare all three models (two univariate, one multivariate)

a. Report training and testing results using RMSE and R2 metrics.

Model	Lookback	RMSE(Training)	R2(Training)	RMSE(Test)	R2(Test)
model 2	5	0.1515	0.5175	0.1781	0.1791
model 3	10	0.1401	0.5872	0.1571	0.3614
model 4	0	0.0622	0.9185	0.1856	0.1083

b. Discuss the comparative performance of the models and their practical usefulness in forecasting pasture growth.

Model 3 (univariate, lookback =10) has high performance with the best balance between training and testing set. Furthermore, Model 3 gets the highest R2 (0.3614) in testing set, which means the best generalisation performance. However, Model 2 (univariate, lookback = 5) shows stable learning performance but less accuracy (R2: 0.1792). Finally, Model 4 (multivariate) shows high learning performance in training set but strong overfitting. Therefore, Model 3 can be applied to the practical usefulness in forecasting pasture growth, providing reliable prediction and accuracy without overfitting.

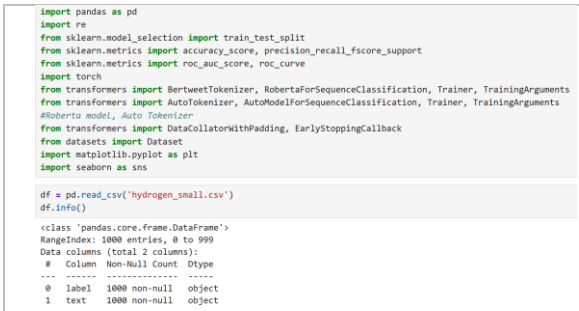
Task 4: Hydrogen Tweet Classification with BERT

The Hydrogen_small dataset (hydrogen_small.csv) is a subset of the dataset introduced in Task 1, containing 1,000 tweets that include the term hydrogen. However, not all of these tweets relate to hydrogen energy. To identify truly relevant content, a classification model is required to distinguish between energy-related and unrelated tweets.

Each tweet in this dataset has been manually labelled as either Relevant (hydrogen energy–related) or Irrelevant. Table 1 outlines the dataset attributes. Alongside the raw tweets (hydrogen_small.csv), you are also provided with tfidf_features_small.csv, which contains TF–IDF embeddings of the tweets. These can be used as input for baseline machine learning models.

Your task is to build and evaluate BERT-based classification models to automatically detect relevant tweets. These models will help stakeholders monitor public discourse and emerging trends in hydrogen energy. Include appropriate outputs, plots, and screenshots where required.

1. **Describe the preprocessing steps required to prepare the Hydrogen dataset for BERT-based classification.**



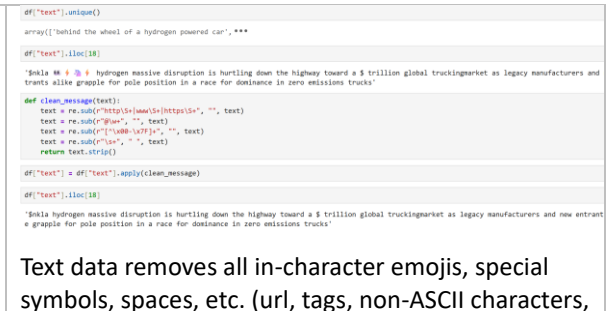
```
import pandas as pd
import re
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, precision_recall_fscore_support
from sklearn.metrics import roc_auc_score, roc_curve
import torch
from transformers import BertweetTokenizer, RobertaForSequenceClassification, Trainer, TrainingArguments
from transformers import AutoTokenizer, AutoModelForSequenceClassification, Trainer, TrainingArguments
#Roberta model, Auto Tokenizer
from transformers import DataCollatorWithPadding, EarlyStoppingCallback
from datasets import Dataset
import matplotlib.pyplot as plt
import seaborn as sns
```

```
df = pd.read_csv('hydrogen_small.csv')
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1000 entries, 0 to 999
Data columns (total 2 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   label   1000 non-null   object
 1   text    1000 non-null   object
```

Import all required modules (data split, evaluation, model, Tokeniser, trainer, visualisation module, etc.)

Check data types and the total number of data.



```
df["text"].unique()
```

```
array(['behind the wheel of a hydrogen powered car',***
```

```
df["text"].iloc[18]
```

```
'$nkla 🏎 ⚡ 🐘 ⚡ hydrogen massive disruption is hurtling down the highway toward a $ trillion global truckingmarket as legacy manufacturers and
trants alike grapple for pole position in a race for dominance in zero emissions trucks'
```

```
def clean_message(text):
    text = re.sub(r"http\S+|www\S+|https\S+", "", text)
    text = re.sub(r"@\w+", "", text)
    text = re.sub(r"[^\x00-\x7F]+", "", text)
    text = re.sub(r"\s+", " ", text)
    return text.strip()
```

```
df["text"] = df["text"].apply(clean_message)
```

```
df["text"].iloc[18]
```

```
'$nkla hydrogen massive disruption is hurtling down the highway toward a $ trillion global truckingmarket as legacy manufacturers and new entrant
e grapple for pole position in a race for dominance in zero emissions trucks'
```

Text data removes all in-character emojis, special symbols, spaces, etc. (url, tags, non-ASCII characters, non-English characters, removal of back-and-forth spaces, and conversion of continuous spaces into single spaces)

```
df["label"].unique()
```

```
array(['Relevant', 'Irrelevant'], dtype=object)
```

Label data is also mapped to Boolean.

```
df["label"] = df["label"].map({
    'Irrelevant': 0, # Negative = 0
    'Relevant': 1 # Positive = 1
})
```

```
df["label"].unique()
```

```
array([1, 0])
```

2. **Fine-tune at least two pre-trained BERT models for binary tweet classification.**
   a. **State which pre-trained BERT models you selected and explain why?**
   b. **Describe the data split strategy used for training and testing?**
   c. **Specify the loss function applied and explain its suitability for this task?**

Bertweet Model	**Roberta Model**

```
model_name1 = "vinai/bertweet-base"
tokenizer1 = BertweetTokenizer.from_pretrained(model_name1)
```

```
emoji is not installed, thus not converting emoticons or emojis into text. Install emoji: pip3 i
```

```
# Function that is applied to all samples in the dataset.

def tokenize_bertweet(batch):
 # We set truncation=True to truncate (cut off) messages that are too long.
 # NOTE: Not all models require this, you may get a warning indicating that it has no effect.
 # Padding is set to True if the model requires a fixed sequence length.
    return tokenizer1(batch['text'], truncation=True, padding=True)
# Apply to both the training and testing datasets.
# We set batched to True which can enable parallel processing, however on my machine I found
# it did not scale to a greater number of threads.
train_ds_bertweet = train_ds.map(tokenize_bertweet, batched=True)
test_ds_bertweet = test_ds.map(tokenize_bertweet, batched=True)
```

```
Map:   0%|          | 0/700 [00:00<?, ? examples/s]
Asking to truncate to max_length but no maximum length is provided and the model has no predefi
Map:   0%|          | 0/300 [00:00<?, ? examples/s]
```

The Bertweet model is a language model trained based on RoBERTa. Since it has been pre-trained with more than 800 million English tweets, it can be said to be a pre-learning model that can better understand Twitter data such as this data type. Bertweet Tokeniser is a Tokeniser dedicated to Bertweet models and specialises in Twitter language processing.

```
model_name2  = 'roberta-base'
tokenizer2 = AutoTokenizer.from_pretrained(model_name2)
```

```
def tokenize_roberta(batch):
    return tokenizer2(batch['text'], truncation=True, padding=True)

train_ds_roberta = train_ds.map(tokenize_roberta, batched=True)
test_ds_roberta = test_ds.map(tokenize_roberta, batched=True)
```

```
Map:   0%|          | 0/700 [00:00<?, ? examples/s]
Map:   0%|          | 0/300 [00:00<?, ? examples/s]
```

```
train_ds
```

```
Dataset({
    features: ['text', 'label'],
    num_rows: 700
})
```

The RoBERTa model is a pre-trained model based only on vast amounts of raw English text. In other words, it is a model that is not specialised like Bertweet, but is trained with larger data and is stronger for general sentence and context identification. It has the advantage of being able to utilise a lot of publicly available data because humans have not labelled themselves.
The AutoTokenizer automatically calls the fitted Tokeniser according to the model structure.

```
model1 = RobertaForSequenceClassification.from_pretrained(
    model_name1,
    num_labels=df["label"].nunique(),
    problem_type="single_label_classification")
```

```
model2 = AutoModelForSequenceClassification.from_pretrained(
    model_name2,
    num_labels=df["label"].nunique(),
    problem_type="single_label_classification")
```

The loss function used in the task is Cross-Entropy Loss. This function calculates the difference between the value predicted by the model and the actual correct answer to obtain the loss, and the closer the prediction is to the actual class probability, the smaller the loss. The learning goal of this model is to minimise loss. In this task, two classes were called: RobertaForSequenceClassification and AutoModelForSequenceClassification. The difference between the two is that Roberta classification specialises in the RoBERTa model, and AutoModel classification automatically calls the suitable model. The way these two classes calculate the loss is the same. According to the official document of the Hugging Face, CrossEntropyLoss is automatically applied as a loss function for classification when config.num_labels>1, and when config.num_labels==1, Mean-SquareLoss for regression is applied. When looking at the code, since a single label is used (num_labels=2), CrossEntropyLoss is automatically applied.

```
X = df["text"].values
y = df["label"].values
random_state = 42

X_train, X_test, y_train, y_test = train_test_split(X,y,
stratify=y, test_size=0.3, random_state=random_state)
print("Training set size:", len(X_train))
print("Testing set size:", len(X_test))
```

```
Training set size: 700
Testing set size: 300
```

```
train_df = pd.DataFrame({"text": X_train, "label": y_train})
test_df = pd.DataFrame({"text": X_test, "label": y_test})
train_ds = Dataset.from_pandas(train_df)
test_ds = Dataset.from_pandas(test_df)

print("Train dataset:", train_ds)
print("Test dataset:", test_ds)
```

```
Train dataset: Dataset({
    features: ['text', 'label'],
    num_rows: 700
})
Test dataset: Dataset({
    features: ['text', 'label'],
    num_rows: 300
})
```

As mentioned in Week 4 Practical, the data were split at a 70:30 ratio according to the machine learning standard procedure. 70% of the data is used for learning, and it is the amount of data that can be sufficiently learned. The remaining 30% is a set of tests to be used for evaluation, not used for learning, and then used to check whether the model is overfitting with the training data. In addition, the stratify=y option guaranteed a balance between training/test data.
In addition, subsampling was not done because the number of data points was not vast.

d. Outline the network architecture and the hyperparameters chosen (e.g., learning rate, batch size, epochs).

Bertweet

```
EarlyStopping_model1 = RobertaForSequenceClassification.from_pretrained(
 model_name1,
 num_labels=df["label"].nunique(),
 problem_type="single_label_classification")
EarlyStopping_model1.train()
EarlyStopping_training_args = TrainingArguments(
 output_dir="./results",
 num_train_epochs=10,
 per_device_train_batch_size=16,
 per_device_eval_batch_size=64,
 eval_strategy="epoch",
 save_strategy="epoch",
 learning_rate=1e-5,
 weight_decay=0.01,
 logging_dir="./logs",
 logging_steps=10,
 # Added for early stopping.
 metric_for_best_model = "loss",
 load_best_model_at_end = True
)
EarlyStopping_trainer1 = Trainer(
 model=EarlyStopping_model1,
 args=EarlyStopping_training_args,
 train_dataset=train_ds_bertweet,
 eval_dataset=test_ds_bertweet,
 processing_class=tokenizer1,
 data_collator=DataCollatorWithPadding(tokenizer1),
 compute_metrics=compute_metrics,
 callbacks = [EarlyStoppingCallback(early_stopping_patience=3)],
)
EarlyStopping_trainer1.train()
```

[264/440 03:42 < 02:29, 1.18 it/s, Epoch 6/10]

Epoch	Training Loss	Validation Loss	Accuracy	Precision	Recall	F1
1	0.565500	0.456672	0.893333	0.858696	0.963415	0.908046
2	0.295200	0.243400	0.936667	0.919075	0.969512	0.943620
3	0.263100	0.242222	0.916667	0.888268	0.969512	0.927114
4	0.174200	0.272386	0.910000	0.874317	0.975610	0.922190
5	0.145800	0.288900	0.916667	0.883978	0.975610	0.927536
6	0.058500	0.290545	0.923333	0.893855	0.975610	0.932945

In the case of Bertweet, as a result of combining several hyperparameters, the F1 score was the highest when epoch 10, train batch size 16, evaluation batch size 64, and learning rate was 1e-5. The validation loss was no longer improved, so it ended early at 6 epoch, and the average loss was 0.2555. Learning proceeded for a total of 222.7 seconds and 264 steps.

```
TrainOutput(global_step=264, training_loss=0.2555245564065196, metrics={'train_runtime': 222.7097, 'train_samples_per_second': 31.431, 'train_steps_per_second': 1.976, 'total_flos': 161874965700000.0, 'train_loss': 0.2555245564065196, 'epoch': 6.0})
```

RoBERTa

```
EarlyStopping_model2 = AutoModelForSequenceClassification.from_pretrained(
 model_name2,
 num_labels=df["label"].nunique(),
 problem_type="single_label_classification")
EarlyStopping_model2.train()
EarlyStopping_training_args = TrainingArguments(
 output_dir="./results",
 num_train_epochs=10,
 per_device_train_batch_size=16,
 per_device_eval_batch_size=64,
 eval_strategy="epoch",
 save_strategy="epoch",
 learning_rate=2e-5,
 weight_decay=0.01,
 logging_dir="./logs",
 logging_steps=10,
 # Added for early stopping.
 metric_for_best_model = "loss",
 load_best_model_at_end = True
)
EarlyStopping_trainer2 = Trainer(
 model=EarlyStopping_model2,
 args=EarlyStopping_training_args,
 train_dataset=train_ds_roberta,
 eval_dataset=test_ds_roberta,
 processing_class=tokenizer2
    ,
 data_collator=DataCollatorWithPadding(tokenizer2),
 compute_metrics=compute_metrics,
 callbacks = [EarlyStoppingCallback(early_stopping_patience=3)],
)
EarlyStopping_trainer2.train()
```

[176/440 02:31 < 03:50, 1.15 it/s, Epoch 4/10]

Epoch	Training Loss	Validation Loss	Accuracy	Precision	Recall	F1
1	0.257100	0.177289	0.936667	0.928994	0.957317	0.942943
2	0.137200	0.260667	0.920000	0.897727	0.963415	0.929412
3	0.066400	0.329329	0.913333	0.879121	0.975610	0.924855
4	0.072000	0.414834	0.916667	0.879781	0.981707	0.927954

RoBERTa also tested several hyperparameters, and as a result, the F1 score was the highest when epoch 10, train batch size 16, evaluation batch size 64, and learning rate was 2e-5. The validation loss was no longer improved, so it ended early at 4 epochs, and the average loss was 0.225. Learning took place for a total of 152.0 seconds and 176 steps.

```
TrainOutput(global_step=176, training_loss=0.22599351016635244, metrics={'train_runtime': 152.0314, 'train_samples_per_second': 46.043, 'train_steps_per_second': 2.894, 'total_flos': 106477755216000.0, 'train_loss': 0.22599351016635244, 'epoch': 4.0})
```

e. Report training and testing accuracy for each model.

Bertweet	RoBERTa

```
# Switch the model to evaluation mode, disabling dropout etc layers.
model1.eval()
# Evaluate the datasets.
train_results_bertweet = EarlyStopping_trainer1.evaluate(train_ds_bertweet)
test_results_bertweet = EarlyStopping_trainer1.evaluate(test_ds_bertweet)
```

```
/opt/anaconda3/lib/python3.12/site-packages/torch/utils/data/dataloader.py:684: UserWarning
MPS now, then device pinned memory won't be used.
  warnings.warn(warn_msg)
/opt/anaconda3/lib/python3.12/site-packages/torch/utils/data/dataloader.py:684: UserWarning
MPS now, then device pinned memory won't be used.
  warnings.warn(warn_msg)
```

```
def display_evaluation(setname_bertweet, results_bertweet):
 print(f"{setname_bertweet} Set Accuracy:", round(results_bertweet["eval_accuracy"], 3))
 print(f"{setname_bertweet} Set Precision:", round(results_bertweet["eval_precision"], 3))
 print(f"{setname_bertweet} Set Recall:", round(results_bertweet["eval_recall"], 3))
 print(f"{setname_bertweet} Set F1 score:", round(results_bertweet["eval_f1"], 3))
display_evaluation("Training", train_results_bertweet)
display_evaluation("Testing", test_results_bertweet)
```

```
Training Set Accuracy: 0.956
Training Set Precision: 0.931
Training Set Recall: 0.992
Training Set F1 score: 0.961
Testing Set Accuracy: 0.917
Testing Set Precision: 0.888
Testing Set Recall: 0.97
Testing Set F1 score: 0.927
```

```
model2.eval()
# Evaluate the datasets.
train_results_roberta = EarlyStopping_trainer2.evaluate(train_ds_roberta)
test_results_roberta = EarlyStopping_trainer2.evaluate(test_ds_roberta)
```

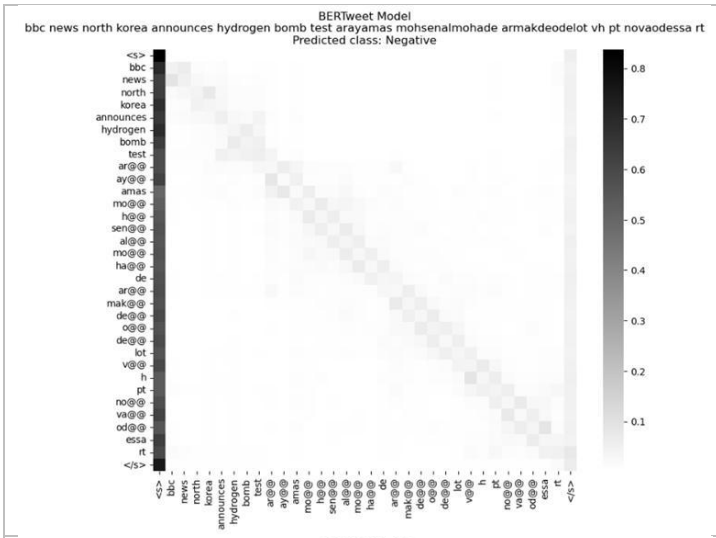
```
/opt/anaconda3/lib/python3.12/site-packages/torch/utils/data/dataloader.py:684: UserWarni
MPS now, then device pinned memory won't be used.
  warnings.warn(warn_msg)
/opt/anaconda3/lib/python3.12/site-packages/torch/utils/data/dataloader.py:684: UserWarni
MPS now, then device pinned memory won't be used.
  warnings.warn(warn_msg)
```

```
def display_evaluation(setname_roberta, results_roberta):
 print(f"{setname_roberta} Set Accuracy:", round(results_roberta["eval_accuracy"], 3))
 print(f"{setname_roberta} Set Precision:", round(results_roberta["eval_precision"], 3))
 print(f"{setname_roberta} Set Recall:", round(results_roberta["eval_recall"], 3))
 print(f"{setname_roberta} Set F1 score:", round(results_roberta["eval_f1"], 3))
display_evaluation("Training", train_results_roberta)
display_evaluation("Testing", test_results_roberta)
```

```
Training Set Accuracy: 0.939
Training Set Precision: 0.943
Training Set Recall: 0.945
Training Set F1 score: 0.944
Testing Set Accuracy: 0.937
Testing Set Precision: 0.929
Testing Set Recall: 0.957
Testing Set F1 score: 0.943
```

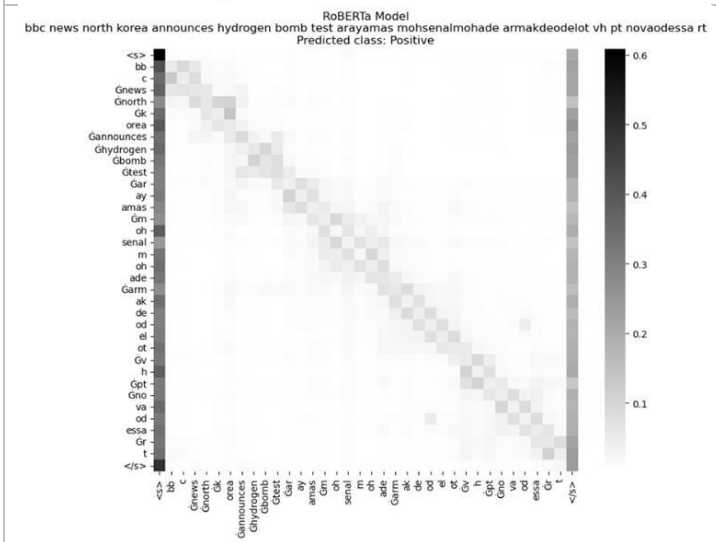
While the learning accuracy of the Bertweet model is slightly higher than that of the RoBERTa model, the test performance and F1 score show that the RoBERTa model is slightly higher. This indicates that the generalisation performance of the RoBERTa model is slightly higher, and the possibility of overfitting is low.

f. Analyse the attention weights from the best-performing BERT model. Select one tweet from each class (relevant and not relevant) and explain how the model's attention differs between them using visualisation tools to highlight key words driving the classification.



Negative

In the case of the Bertweet model, among the forms of clear words, north, korea, bomb, and test were marked relatively strongly. Since words in an uncertain form and @@mark are a method of dividing words into sub-units in the Bertweet model, it is considered difficult for words other than the above clear words to actually evaluate the weight.



Poistive

It can be seen that the RoBERTa model has a little broader distribution of weights compared to the Bertweet model. That is, you can see the weights distributed relatively evenly in all news, north, announces, hydrogen, bomb, and test. The 'Ġ' means a space in front of the token, and RoBERTa also seems hard to interpret meaningful weights for words with unclear spaces and shapes.

In other words, the Bertweet model was concentrated on some specific words, resulting in

negative, while the RoBERTa model was relatively evenly distributed in weight, which can be interpreted as returning a positive result considering the context of the sentence.

3. Compare BERT's performance against a traditional machine learning approach.

a. Use the provided tfidf_features_small.csv to build a logistic regression baseline model.

```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression

tfidf_df = pd.read_csv('tfidf_features_small.csv')
tfidf_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1000 entries, 0 to 999
Columns: 4981 entries, aaa to 測文1
dtypes: float64(4981)
memory usage: 38.0 MB
```

```
X = tfidf_df

df= pd.read_csv("hydrogen_small.csv")
y= df['label'].values

random_state = 42
test_set_size = 0.3
X_train, X_test, y_train, y_test = train_test_split(X.values, y, test_size=test_set_size, stratify=y,
                                                    random_state=random_state)
model = LogisticRegression(random_state=random_state)

model.fit(X_train, y_train)
```

```
LogisticRegression(random_state=42)
```

To create baseline model for comparison, we created the logistic regression model (LR).
The hyperparameter of logistic regression was set:

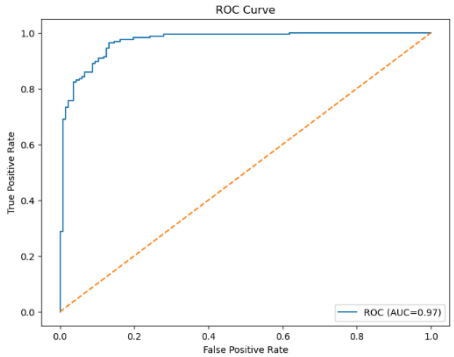
- random_state = 42 to train a variety of data.
- test_set_size = 0.3 for splitting data to 30% for testing and 70% for training.

b. Present results in an accuracy table, classification report, and ROC curve, comparing the best-performing BERT model against the logistic regression baseline.

Logistic Regression Model (LR)

```
from sklearn.metrics import classification_report
# training and test accuracy
print("Train accuracy:", model.score(X_train, y_train))
print("Test accuracy:", model.score(X_test, y_test))

# classification report on test data
y_pred = model.predict(X_test)
print(classification_report(y_test, y_pred))
```

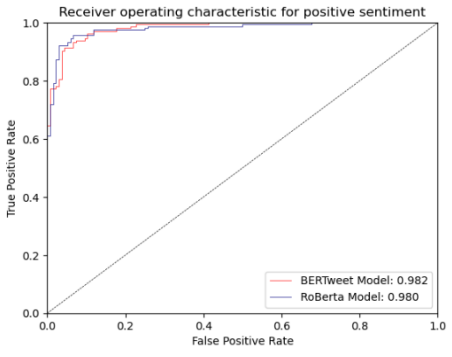
```
Train accuracy: 0.9885714285714285
Test accuracy: 0.8966666666666666
              precision    recall  f1-score   support

  Irrelevant       0.96      0.80      0.88       136
    Relevant       0.86      0.98      0.91       164

    accuracy                           0.90       300
   macro avg       0.91      0.89      0.89       300
weighted avg       0.91      0.90      0.90       300
```



BERTweet model

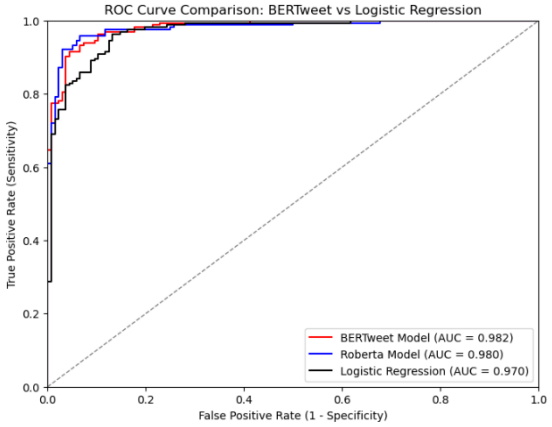
```
display_evaluation("Training", train_results_bertweet)
display_evaluation("Testing", test_results_bertweet)
```

```
Training Set Accuracy: 0.956
Training Set Precision: 0.931
Training Set Recall: 0.992
Training Set F1 score: 0.961
Testing Set Accuracy: 0.917
Testing Set Precision: 0.888
Testing Set Recall: 0.97
Testing Set F1 score: 0.927
```



BERTweet model was chosen as the best-performing BERT model compared ROC curve. Thus, we should compare this model with the baseline model. Both LR and BERTweet show high test accuracy, approximately 0.90. However, the difference in train/test accuracy of LR is higher than that of the BERTweet model, suggesting the possibility of overfitting in LR. ROC curve can be used to confirm the trade-off between specificity and sensitivity for a binary classification model. Based on the ROC curve, the BERTweet Model (0.982) has higher prediction performance than the baseline (0.97).

c. Discuss whether BERT outperformed the baseline and provide potential explanations for the observed performance difference.



According to the ROC curve, BERTweet model demonstrates the highest AUC value (0.982) since this model also performed high testing accuracy as well as precision and f1-score as discussed in 4.3.b. There are several reasons why the BERT model has higher performance than LR: BERT model is an encoder-only transformer, which can learn the contextual embedding. Furthermore, BERT model was fine-tuned better than LR to enhance the generalisation. In addition, the biggest difference between the Transformer model and LR is pretraining. Bertweet/RoBERTa models are pretrained to search the relation of words and context, which can find relevance more easily than LR, even with small amounts of data. However, LR relies on the frequency of words and can't understand the relevance of words and context.

Task 5: Building a Question Answering (Q&A) system with T5

5.1. Describe the preprocessing steps applied to the SQuAD_tiny dataset to prepare it for input into the T5 model-based Q&A system.

```
# Set seed for reproducibility
torch.manual_seed(42)

<torch._C.Generator at 0x34a1663d0>
```

The random number seed was set for all devices to return the same sequence of random numbers.

```
# Load and preprocess SQuAD dataset

dataset = load_dataset("squad")

print("Number of training examples:", len(dataset['train']))
print("Number of validation examples:", len(dataset['validation']))

dataset['train'][0]
```

```
Number of training examples: 87599
Number of validation examples: 10570
{'id': '5733be284776f41900661182',
 'title': 'University_of_Notre_Dame',
 'context': 'Architecturally, the school has a Catholic character. Atop the Main Building\'s gold dome is a golden statue of the Virgin Mary. Immediately in front of the Main Building and facing it, is a copper statue of Christ with arms upraised with the legend "Venite Ad Me Omnes". Next to the Main Building is the Basilica of the Sacred Heart. Immediately behind the basilica is the Grotto, a Marian place of prayer and reflection. It is a replica of the grotto at Lourdes, France where the Virgin Mary reputedly appeared to Saint Bernadette Soubirous in 1858. At the end of the main drive (and in a direct line that connects through 3 statues and the Gold Dome), is a simple, modern stone statue of Mary.',
 'question': 'To whom did the Virgin Mary allegedly appear in 1858 in Lourdes France?',
 'answers': {'text': ['Saint Bernadette Soubirous'], 'answer_start': [515]}}
```

There are 87599 samples in the training set in this dataset, which consists of context, questions, and answers.

```
# Data Filtering
filtered_dataset = dataset.filter(lambda x: x["answers"]['text'][0].upper() !="CANNOTANSWER")
print("Size of training set after removing unanswerable questions:", len(filtered_dataset['train']))
print("Size of validation set after removing unanswerable questions:", len(filtered_dataset['validation']))
```

```
Size of training set after removing unanswerable questions: 87599
Size of validation set after removing unanswerable questions: 10570
```

Several samples can contain a "CANNOTANSWER" value, which should be removed to avoid fine-tuning the model on unanswerable questions. However, our dataset does not have

“CANNOTANSWER”, according to the output of the lengths of dataset.

```
# Take subsets to avoid overload
train_dataset = dataset["train"].select(range(10000,11000))
val_dataset = dataset["validation"].select(range(3000,3100))
test_dataset = dataset["validation"].select(range(3100, 3200))

training_set = train_dataset
validation_set = val_dataset
testing_set = test_dataset

print("Size of training set:", len(train_dataset))
print("Size of validation set:", len(val_dataset))
print("Size of testing set:", len(test_dataset))
```

```
Size of training set: 1000
Size of validation set: 100
Size of testing set: 100
```

The small dataset in SQuAD was created to reduce computational requirements. this code sets the range of training, validation, and test sets through the whole SQuAD dataset. The reason to separate each dataset is to avoid repeated evaluation and tuning through the validation set, which can lead to overfitting to the validation set.

```
# Loading the Tokenizer
MODEL_NAME = "t5-small"

MAX_INPUT_LENGTH = 512
MAX_OUTPUT_LENGTH = 128

# Load tokenizer and model
tokenizer = T5Tokenizer.from_pretrained(MODEL_NAME)
model = T5ForConditionalGeneration.from_pretrained(MODEL_NAME)
```

This part is to convert the text into **tokens** using **T5Tokenizer** class by the HuggingFace Transformers.
In our project, t5-small model was utilised to reduce training time. Also, the maximum size of input and output are defined.

To convert text strings to a series of numeric token IDs, we created three functions: **encode_question_and_context(), extract_sample_parts(), and preprocess().** This preprocessing step is essential for the direct usage of datasets in the T5 model.

```
def encode_question_and_context(question, context):
    return f"question: {question}  context: {context}"

# Obtains the context, question and answer from a given sample.
def extract_sample_parts(sample):
    context = sample["context"]
    question = sample["question"]
    answer = sample["answers"]['text'][0]
    question_with_context = encode_question_and_context(question, context)
    return (question_with_context, question, answer)
```

encode_question_and_context() combines the question and context, and **extract_sample_parts()** obtains the context, question, and answer from the sample. This return values in **extract_sample_parts()** was utilised to encode the sample in **preprocess(),** generating the tokenizers. After that, this function pads all sequences in a batch to the maximum length. Finally, training_set_enc, validation_set_enc, and testing_set_enc were datasets changed to input_ids, attenstion_mask, labels to enter the T5 model.

```
# Encodes the sample, returning token IDs.
def preprocess(sample):
    # Extract data from sample.
    question_with_context, question, answer = extract_sample_parts(sample)

    # Using truncation causes the tokenizer to emit a warning for every sample.
    # This will generate a significant amount of messages, and likely crash
    # your browser tab. We temporarily disable log messages to work around this.
    # See https://github.com/huggingface/transformers/issues/14285
    old_level = transformers_logging.get_verbosity()
    transformers_logging.set_verbosity_error()

    # Generate tokens for the input.
    # We include both the context and the question (first two parameters).
    input_tokens = tokenizer(question_with_context, question, padding="max_length",
                             truncation=True, max_length=MAX_INPUT_LENGTH)

    # Generate tokens for the expected answer. There is no need to include the
    output_tokens = tokenizer(answer, padding="max_length", truncation=True,
                              max_length=MAX_OUTPUT_LENGTH)

    # Restore old logging level, see above.
    transformers_logging.set_verbosity(old_level)

    # The output of the tokenizer is a map containing {input_ids, attention_mask}.
    # For trianing, we need to add the labels (answer/output tokens) to the map.
    input_tokens["labels"] = np.array(output_tokens["input_ids"])

    return input_tokens
```

```
# Preprocess the datasets
training_set_enc = train_dataset.map(preprocess, batched=False)
validation_set_enc = val_dataset.map(preprocess, batched=False)
testing_set_enc = test_dataset.map(preprocess, batched=False)
```

5.2. Fine-tune the T5 model using the training portion of the SQuAD_tiny dataset.

a. Describe the fine-tuning procedure and detail the hyperparameters used.

```
try:
    del model
except NameError:
    pass

model = T5ForConditionalGeneration.from_pretrained(MODEL_NAME)
```

```
columns = ["input_ids", "attention_mask", "labels"]
training_set_enc.set_format(type="torch", columns=columns)
validation_set_enc.set_format(type="torch", columns=columns)
testing_set_enc.set_format(type="torch", columns=columns)
```

We should delete the existing model before training of new model.
After that, we need to set the format to use PyTorch tensors to improve the performance of training.

Setting 1	Setting2

```
# hyperparameter: setting1

training_args = TrainingArguments(
    output_dir="./results",
    num_train_epochs=3, #5-10
    per_device_train_batch_size=8,
    per_device_eval_batch_size=8,
    eval_strategy="epoch",
    save_strategy="epoch",
    learning_rate=3e-4,
    weight_decay=0.01,
    save_total_limit=2,
    logging_dir="./logs",
    logging_steps=10,
    metric_for_best_model = "loss",
    load_best_model_at_end = True #Early Stopping
)
# Train T5 model: setting1

model.train()
trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=training_set_enc,
    eval_dataset=validation_set_enc,
    processing_class=tokenizer,
    data_collator=DataCollatorForSeq2Seq(tokenizer),
    callbacks = [EarlyStoppingCallback(early_stopping_patience=3)],
)

trainer.train()
```

num_train_epochs = 3
per_device_train_batch_size = 8
per_device_eval_batch_size = 8
eval_strategy = “epoch”
weight_decay = 0.01
learning_rate = 3e-4
save_tatal_limit = 2
And put the early stopping strategy, which stops training if there is no improvement for 3 times.

```
# hyperparameter: setting2

training_args_change = TrainingArguments(
    output_dir="./results",
    num_train_epochs=8,
    per_device_train_batch_size=8,
    per_device_eval_batch_size=16,
    eval_strategy="epoch",
    save_strategy="epoch",
    learning_rate=2e-4,
    weight_decay=0.01,
    save_total_limit=2,
    logging_dir="./logs",
    logging_steps=20,
    metric_for_best_model = "loss",
    load_best_model_at_end = True #Early Stopping
)
# Train T5 model: setting2

model.train()
trainer_change = Trainer(
    model=model,
    args=training_args_change,
    train_dataset=training_set_enc,
    eval_dataset=validation_set_enc,
    processing_class=tokenizer,
    data_collator=DataCollatorForSeq2Seq(tokenizer),
    callbacks = [EarlyStoppingCallback(early_stopping_patience=3)]
)

trainer_change.train()
```

num_train_epochs = 8
per_device_train_batch_size = 8
per_device_eval_batch_size = 16
- For faster evaluation

eval_strategy = “epoch”
weight_decay = 0.01
learning_rate = 2e-4
- To converge slower than 3e-4

save_tatal_limit = 2
And put the early stopping strategy, which stops training if there is no improvement for 3 times.

b. Report both training and validation losses during fine-tuning.

Setting 1

[375/375 08:06, Epoch 3/3]

Epoch	Training Loss	Validation Loss
1	0.030700	0.022716
2	0.020500	0.020635
3	0.014900	0.020975

Setting 2

[625/1000 13:04 < 07:52, 0.79 it/s, Epoch 5/8]

Epoch	Training Loss	Validation Loss
1	0.037100	0.021770
2	0.021600	0.019465
3	0.014900	0.020631
4	0.013000	0.021221
5	0.011700	0.021623

	global_step	train_loss	runtime	samples_ per_second	train_steps_p er _second	total_flos	epoch
1	375	0.2287	487.6024	6.153	0.769	406025404416000	3.0
2	625	0.1823	785.1482	10.189	1.274	676709007360000	5.0

The training loss steadily decreased in setting 1 (from 0.0307 to 0.0149) and setting 2 (from 0.0371 to 0.0117). Furthermore, validation loss remains around 0.020 in both settings without reduction. This situation means both models have stable performance with low training and testing loss. In terms of running time, setting 2 has more running time than setting 1 because of the number of epochs and the low learning rate. Moreover, training of setting 2 was stopped in epoch 5, which means there are no meaningful results after epoch 5. As a result, the model of setting 1 was saved since setting 2 is the degree to analyse setting 1 closely:

```
trainer.save_model("t5_pretrained")
```

c. Explain the strategies applied to prevent overfitting.

To reduce the risk of overfitting, we can apply several strategies:

- Early stopping is automatic learning stopping if there is no improvement of validation loss.
- Weight decay was used Adam optimizer.
- Learning rate was adjusted for stable convergence
- Limited Epochs was utilised to prevent unnecessary long leaning.

5.3. Evaluate the fine-tuned model on the test set (input: question + context).

a. Report performance using ROUGE metrics.

```
def display_evaluation(setname, results):
    print(f"{setname} Set Loss:", round(results["eval_loss"], 3))
# Switch the model to evaluation mode, disabling dropout etc layer
model.eval()

# Evaluate the datasets.
display_evaluation("Training", trainer.evaluate(training_set_enc))
display_evaluation("Testing", trainer.evaluate(testing_set_enc))
```

```
/opt/anaconda3/lib/python3.12/site-packages/torch/utils/data/datal
rgument is set as true but not supported on MPS now, then device p
  warnings.warn(warn_msg)
```

[125/125 00:49]

```
Training Set Loss: 0.009
Testing Set Loss: 0.018
```

We create function, named **display_evaluation()** to show the training set loss and testing set loss easily. Firstly, we change the model to the evaluation mode. After that, the average loss data was calculated.

```
# Computes the average score of a given metric from a list of ROUGE scores.
def compute_average_score(scores, metric, key):
    total = 0
    for i in range(len(scores)):
        # Since it's not a map, we have to manually read the attribute.
        total += getattr(scores[i][metric], key)
    return total / len(scores)

# Computes ROGUE-1, ROGUE-2 and ROGUE-L scores for the given generated
# answers and reference answers.
def compute_rouge(predictions, references):
    # Compute ROUGE-1, ROGUE-2 and ROUGE-L.
    metrics = ["rouge1", "rouge2", "rougeL"]

    # Use Porter stemmer to strip word suffixes to improve matching.
    scorer = rouge_scorer.RougeScorer(metrics, use_stemmer=True)

    # For each answer/reference pair, compute the ROUGE metrics.
    scores = []
    for prediction, reference in zip(predictions, references):
        scores.append(scorer.score(reference, prediction))

    # Compute the average precision, recall and F1 score for each metric.
    results = {}
    for metric in metrics:
        for k in ["precision", "recall", "fmeasure"]:
            results[f"{metric}_{k}"] = compute_average_score(
                scores, metric, k)
    return results
```

```
answers_ctx, refs_ctx, questions_ctx = generate_answers(tokenizer, model, testing_set, True, 100)
answers_noctx, refs_noctx, questions_noctx = generate_answers(tokenizer, model, testing_set, False, 100)
```

```
print("ROUGE with context:", compute_rouge(answers_ctx, refs_ctx))
print()
print("ROUGE without context:", compute_rouge(answers_noctx, refs_noctx))
```

```
ROUGE with context: {'rouge1_precision': 0.60075, 'rouge1_recall': 0.5761666666666667, 'rouge1_fmeasure': 0
.5695733808674984, 'rouge2_precision': 0.37666666666666665, 'rouge2_recall': 0.3610833333333334, 'rouge2_fm
easure': 0.3624920634920635, 'rougeL_precision': 0.60075, 'rougeL_recall': 0.5761666666666667, 'rougeL_fmea
sure': 0.5695733808674984}

ROUGE without context: {'rouge1_precision': 0.005543040293040293, 'rouge1_recall': 0.01477777777777779, 'r
ouge1_fmeasure': 0.007686868686868686, 'rouge2_precision': 0.0, 'rouge2_recall': 0.0, 'rouge2_fmeasure': 0.
0, 'rougeL_precision': 0.005543040293040293, 'rougeL_recall': 0.01477777777777779, 'rougeL_fmeasure': 0.00
7686868686868686}
```

ROUGE-1: the overlap of individual words between the generated and reference texts.
ROUGE-2: the overlap of consecutive words, which captures grammatical structure and fluency.
ROUGE-L: the longest common subsequence algorithm to identify the longest sequence of matching words.
High scores in all these metrics means better alignment between the generated words and reference, which can lead to word-level accuracy.
We created **compute_rouge()** and **compute_average_score()** functions.

b. Interpret what the results suggest about model accuracy and fluency.

In this part, we should mainly address the case of input with context.

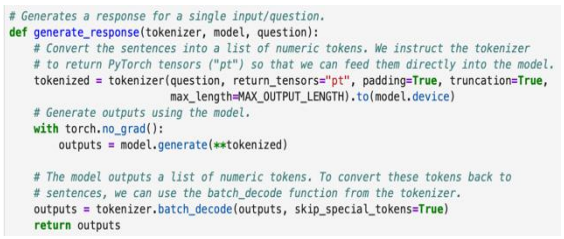
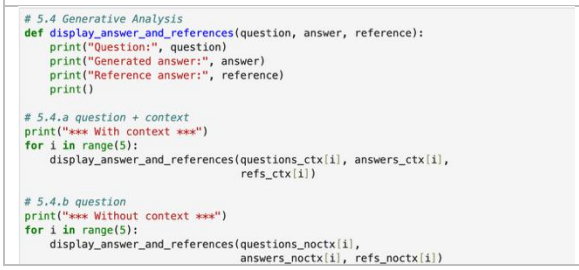
With context	precision	recall	F1	without context	Precision	Recall	F1
rouge 1	0.60075	0.576167	0.569873	rouge 1	0.0055	0.0148	0.00769
rouge 2	0.37697	0.36108	0.362492	rouge 2	0.0	0.0	0.0
rouge L	0.60075	0.57617	0.569573	rouge L	0.0054	0.01478	0.00769

When the model was inputted with both the question and context, the ROUGE scores is high score, leading to high accuracy and fluency. This result indicates high scores regarding individual words and sentence structure of the longest sequences according to F1 of ‘rouge1’, ‘rouge2’ as 0.569873 and 0.569574. The result of rouge 2 is low score as 0.362492, which means that consecutive word pairs

cannot be predicted accurately. As a result, the model can predict high accuracy about the meaning, but this model has limitation about the fluency because of low rouge 2 score. However, when context does not input the model, the result shows low F1 score of 'rouge 1', 'rouge2', 'rouge L' as 0.00769, 0.0, 0.00769. This results mean this model failed to generate relevant and meaningful answers without context.

5.4 Generative Analysis on Test Samples using the first 5 data points (indices 0 to 4) from the test set of *SQuAD_tiny*.

To generate the model responses, two functions such as **generate_response()** and **generate_answers()** were defined to process both questions and contexts.

	In the **generate_response()** function, the t5 model generates an answer for a question through three steps. 1. Tokenization: The input question is converted to numeric tokens as the T5 model processes encoded numeric token. 2. Model generation: The **model.generate()** method produces the output using the trained T5 model. 3. Decoding: The **batch_decoded()** function converts the generated numeric tokens back into text.
	In the **generate_answers()** function, all answers are generated at once through five steps. 1. Sampling: The **range(limit)** method selects some data points. 2. Data separation: The function extracts the question, context, and reference answer from each sample. 3. Input composition: Two types of input are created – input both question and context (**append(question_with_context)**) and only input question (**append(question)**). 4. Model processing: The model generates responses for each input by calling **generate_response()** multiple times. 5. Output return: The function returns three lists containing the original questions, generated answers, and reference answers.
	The **display_answer_and_question()** is to print inputted question, generated answer, and reference answer depending on the absence or presence of context using the first 5 data points.

a. Generate answers when the input includes both question + context.
b. Generate answers when the input contains only the question.
Compare the quality of answers in both scenarios.
• Highlight strengths, limitations, or errors in the generated responses.
• Discuss how the presence or absence of context impacts performance.

a. question + context

```
*** With context ***
Question: What country initially received the largest number of Huguenot refugees?
Generated answer: Dutch Republic
Reference answer: the Dutch Republic

Question: How many refugees emigrated to the Dutch Republic?
Generated answer: 2 million
Reference answer: an estimated total of 75,000 to 100,000 people

Question: What was the population of the Dutch Republic before this emigration?
Generated answer: ca. 2 million
Reference answer: ca. 2 million

Question: What two areas in the Republic were first to grant rights to the Huguenots?
Generated answer: the Cévennes
Reference answer: Amsterdam and the area of West Frisia

Question: What declaration predicated the emigration of Huguenot refugees?
Generated answer: Edict of Nantes
Reference answer: the revocation of the Edict of Nantes
```

b. only the question

```
*** Without context ***
Question: What country initially received the largest number of Huguenot refugees?
Generated answer:
Reference answer: the Dutch Republic

Question: How many refugees emigrated to the Dutch Republic?
Generated answer:
Reference answer: an estimated total of 75,000 to 100,000 people

Question: What was the population of the Dutch Republic before this emigration?
Generated answer:
Reference answer: ca. 2 million

Question: What two areas in the Republic were first to grant rights to the Huguenots?
Generated answer:
Reference answer: Amsterdam and the area of West Frisia

Question: What declaration predicated the emigration of Huguenot refugees?
Generated answer:
Reference answer: the revocation of the Edict of Nantes
```

According to the two printed results, the generated answers were significantly affected by whether the context existed or not. When the context was included, the T5 model mostly generated accurate and well-relevant answers, which were highly similar to the reference answers. In contrast, when the input did not include any context, the model failed to produce meaningful responses. Although most generated answers were similar to the reference answers, slight variations of words were observed. As discussed in 5.3, the ROUGE performances also proved that including the context can lead to a higher precision score. Therefore, the presence or absence of context directly impacts the generative capability of the T5 model and highlights the importance of contextual information to ensure answer accuracy and reduce errors.

5.5 Comparison with a Pre-trained Model.

a. Load the pre-trained model "*mrm8488/t5-base-finetuned-squadv2*" from Hugging Face, which was trained trained on the full *SQuAD* dataset.

```python
MODEL_NAME = "mrm8488/t5-base-finetuned-squadv2"

MAX_INPUT_LENGTH = 512
MAX_OUTPUT_LENGTH = 128
```

```python
try:
    del model
except NameError:
    pass

tokenizer = T5Tokenizer.from_pretrained(MODEL_NAME)
model = T5ForConditionalGeneration.from_pretrained(MODEL_NAME)
```

To compare the performance with a fine-tuned model, the model **"mrm8488/t5-base-finetuned-squadv2"** was loaded from Hugging Face with the T5 tokeniser, which was originally trained on the SQuAD dataset. Also, the **try: del model** logic prevents crashes by deleting previously loaded model from memory.

b. Replicate the evaluation steps from Tasks 3 and 4.

```python
answers_ctx, refs_ctx, questions_ctx = generate_answers(
    tokenizer, model, testing_set, True, 100)
answers_noctx, refs_noctx, questions_noctx = generate_answers(
    tokenizer, model, testing_set, False, 100)
```

```python
print("ROUGE with context:", compute_rouge(answers_ctx, refs_ctx))
print()
print("ROUGE without context:", compute_rouge(answers_noctx, refs_noctx))
```

```python
def display_answer_and_references(question, answer, reference):
    print("Question:", question)
    print("Generated answer:", answer)
    print("Reference answer:", reference)
    print()

print("*** With context ***")
for i in range(5):
    display_answer_and_references(questions_ctx[i], answers_ctx[i],
                                  refs_ctx[i])

# 5.4.b question
print("*** Without context ***")
for i in range(5):
    display_answer_and_references(questions_noctx[i],
                                  answers_noctx[i], refs_noctx[i])
```

These codes were duplicated from task 5.3 and 5.4 to evaluate both the ROUGE score and generated answers of the pre-trained model, depending on the presence or absence of contextual information.

c. Compare its performance against your fine-tuned *SQuAD_tiny* model.

d. Discuss differences in results and explain why they occur.

with context	Precision	Recall	F1
rouge 1	0.5610	0.5676	0.5423
rouge 2	0.3512	0.3542	0.3413
rouge L	0.5610	0.5676	0.5423

without context	Precision	Recall	F1
rouge 1	0.0058	0.0038	0.0045
rouge 2	0.0	0.0	0.0
rouge L	0.0058	0.0038	0.0045

Compared to the results between the fine-tuned model and the pre-trained model, all precision, recall, and F1 metrics of the pre-trained model achieved slightly lower ROUGE scores than the fine-tuned result. In particular, the result of Rouge L F1 shown nearly 0.54 in the pre-trained model including context, while the fine-tuned model scored almost 0.57, which indicates a moderate improvement in the word-level accuracy and overall coherence in the fine-tuning model. When the input excluded context, both models dropped to nearly zero across all metrics, and it can expect that both models cannot generate meaningful responses without contextual information.

```
*** With context ***
Question: What country initially received the largest number of Huguenot refugees?
Generated answer: Dutch Republic
Reference answer: the Dutch Republic

Question: How many refugees emigrated to the Dutch Republic?
Generated answer: multiply(n2,const_2)|
Reference answer: an estimated total of 75,000 to 100,000 people

Question: What was the population of the Dutch Republic before this emigration?
Generated answer: multiply(n0,n1)|subtract(n2,n0)|divide
Reference answer: ca. 2 million

Question: What two areas in the Republic were first to grant rights to the Huguenots?
Generated answer: add(n2,const_1)|
Reference answer: Amsterdam and the area of West Frisia

Question: What declaration predicated the emigration of Huguenot refugees?
Generated answer: Edict of Nantes
Reference answer: the revocation of the Edict of Nantes
```

```
*** Without context ***
Question: What country initially received the largest number of Huguenot refugees?
Generated answer: Lieu initial de réfugié(n0,n1)|
Reference answer: the Dutch Republic

Question: How many refugees emigrated to the Dutch Republic?
Generated answer: divide(n0,const_100)|divide(#0,const_
Reference answer: an estimated total of 75,000 to 100,000 people

Question: What was the population of the Dutch Republic before this emigration?
Generated answer: add(const_1,const_4)|population_year(n0)|
Reference answer: ca. 2 million

Question: What two areas in the Republic were first to grant rights to the Huguenots?
Generated answer:
Reference answer: Amsterdam and the area of West Frisia

Question: What declaration predicated the emigration of Huguenot refugees?
Generated answer:
Reference answer: the revocation of the Edict of Nantes
```

As discussed in 5.4, the fine-tuned model produced consistent and almost accurate natural-language answers when context was included, while it failed to respond without context. In contrast, the pre-trained model generated numeric or symbolic tokens such as **multiply(n2, const_2)|**, regardless of the context presences, which reflect unstable decoding and insufficient linguistic fluency. As a result, these findings suggest that fine-tuning help performance improve quantitatively as well as stability enhancement and fluency of generated answers. In summary, while both models significantly rely on context, the fine-tuned model demonstrated better adaption, producing more stable and realistic outputs.