Student Declaration of Authorship

HERIOT WATT UNIVERSITY
UK | DUBAI | MALAYSIA

Course code and name:	F20DL: Data Mining and Machine Learning
Type of assessment:	**Group**
Coursework Title:	Coursework 4 : Neural Networks and Convolutional Neural Networks.
Student Name:	Arshati Ajay Marchande, Asmitha Krishnakumar, Gauri Revankar, Pooja Sheni Meledath, Prasitha Naidu
Student ID Number:	H00382093, H00376043, H00373987, H00386700, H00379641

Declaration of authorship. By signing this form:

- **I declare** that the work I have submitted for individual assessment OR the work I have contributed to a group assessment, is entirely my own. I have NOT taken the ideas, writings or inventions of another person and used these as if they were my own. My submission or my contribution to a group submission is expressed in my own words. Any uses made within this work of the ideas, writings or inventions of others, or of any existing sources of information (books, journals, websites, etc.) are properly acknowledged and listed in the references and/or acknowledgements section.

- I confirm that I have read, understood and followed the University's Regulations on plagiarism as published on the University's website, and that I am aware of the penalties that I will face should I not adhere to the University Regulations.

- I confirm that I have read, understood and avoided the different types of plagiarism explained in the University guidance on Academic Integrity and Plagiarism

Student Signature *(type your name):* *Arshati, Asmitha, Gauri, Pooja, Prasitha*

Date: *26/11/2023*

Copy this page and insert it into your coursework file in front of your title page.
For group assessment each group member must sign a separate form and all forms must be included with the group submission.

Your work will not be marked if a signed copy of this form is not included with your submission.

PART 4: NEURAL NETWORK & CNN

TABLE OF CONTENTS

CONTRIBUTIONS

- Arshati
  - Implementation of CNN model and experimented with different layers.
  - Report
- Asmitha
  - Linear Classifier Analysis
  - Report
- Gauri
  - Implemented MLP and tested different hyper parameters.
  - Report
- Pooja
  - Performing hyper parameter tuning systematically.
  - Report
- Prasitha
  - Implementation of CNN model and experimented with different layers.
  - Report

LINEAR CLASSIFIER ANALYSIS

We've used Support Vector Machines (SVM) as a linear classifier to explore the classification performance on two distinct datasets: one without oversampling and another with oversampling applied to it. The table below shows the comparison with both the datasets.

	Non-oversampled Datasets			Oversampled Datasets		
Model	**Cross validation**	**Without Cross Validation**	**Accuracy**	**Cross validation**	**Without Cross Validation**	**Accuracy**
SVM	0.957	0.962	Training set score: 1.0000 Test set score: 0.9623 Model Accuracy Score : 0.9623 Precision Score : 0.9623 Recall Score : 0.9623 Mean Absolute Error : 0.0588 F1 Score : 0.9623	0.988	0.989	Model Accuracy Score : 0.9898 Precision Score : 0.9898 Recall Score : 0.9898 Mean Absolute Error : 0.0171 F1 Score : 0.9898

CONCLUSION

- **Generalization:** The SVM model, demonstrates high accuracy on both training and test sets thus, it suggests strong generalization to new data.
- **Hypothesis:** The high accuracy and consistency in performance between training and test sets suggest that the data is linearly separable. Evaluation metrics such as precision, recall and F1 scores also indicate a robust ability to distinguish between classes.
- **Overall Comparison between oversampled and non-oversampled data:** The SVM model when trained on oversampled data, demonstrate better performance, surpassing the accuracy achieved without oversampling. From the results above, we can infer, that oversampling appears to enhance the model's ability to handle imbalanced classes and thus improve generalization.

MULTILAYER PERCEPTRON

Relation between hyperparameters of MLP and model accuracy

The model is trained on the Training Dataset and is evaluated after splitting the training set in an 80:20 ratio. The hyperparameters dealt with in this task are:

1. **Activation functions**

```
Accuracies of the model based on the Activation functions:
96.34% - logistic
96.76% - tanh
94.26% - relu
```

The best accuracy obtained is **96.76% with tanh activation function**, followed by logistic and relu. The lowest accuracy obtained is 94.26% with relu activation function.

2. **Number of hidden layers and number of nodes in each layer**

Number of nodes

```
25.619999999999997% - (1,)
25.619999999999997% - (2,)
58.57% - (3,)
88.6% - (4,)
55.66% - (5,)
79.86999999999999% - (6,)
91.18% - (7,)
80.2% - (8,)
87.35000000000001% - (9,)
93.58999999999999% - (10,)
92.17999999999999% - (11,)
92.01% - (12,)
89.42999999999999% - (13,)
83.53% - (14,)
93.34% - (15,)
```

The highest accuracy obtained is with **93.58% with 10 hidden layers** out of the 15 layers tested. The lowest accuracy obtained is 25.619% with 1 and 2 hidden layers. The accuracy increases with the increasing number of nodes in a layer and reaches its peak at 10 layers and then decreases. However, this increase is not steady.

Number of layers

(1, 2 and 3 layers with 10 nodes in each layer)

```
Accuracies of the model based on the Number of Layers (10 nodes):
85.77% - (10,)
93.43% - (10, 10)
90.60000000000001% - (10, 10, 10)
```

(1, 2 and 3 layers with 4 nodes in each layer)

```
Accuracies of the model based on the Number of Layers (4 nodes):
69.97% - (4,)
54.910000000000004% - (4, 4)
63.06% - (4, 4, 4)
48.089999999999996% - (4, 4, 4, 4)
```

From the above two experiments, it is understood that for the given dataset, the relation between accuracy of the model and the number of layers is not linear.

3. **Learning rate**

```
Accuracies of the model based on the Learning Rate:
96.00999999999999% - constant
95.42% - invscaling
95.76% - adaptive
```

The best accuracy obtained is **96.009% with a constant learning rate**, followed by adaptive and invscaling learning rates. The lowest accuracy obtained is 95.42% with invscaling learning rate.

4. **Momentum**

```
Accuracies of the model based on the Momentum:
81.11% - 0.2
86.44% - 0.5
95.34% - 0.9
```

The best accuracy obtained is **95.34% with 0.9 momentum value**, followed by 0.5 and 0.2 momentum values. The lowest accuracy obtained is 81.11% with 0.2 momentum value. This shows that there is a steady increase in accuracies with increasing values of momentum.

5. **Validation threshold**

```
Accuracies of the model based on the Validation Threshold:
95.17% - 0.1
93.34% - 0.3
90.77% - 0.5
```

The best accuracy obtained is **95.17% with 0.1 validation threshold**, followed by 0.3 and 0.5. The lowest accuracy obtained is 90.77% with 0.5 validation threshold. This shows that there is a steady decrease in the accuracies with increasing values of validation threshold.

6. **Epochs**

```
Accuracies of the model based on the Epochs:
92.67999999999999% - 50
95.50999999999999% - 100
95.84% - 200
96.00999999999999% - 300
```

The best accuracy obtained is **96.009% with 300 epochs**, followed by 200, 100 and 50 epochs. The lowest accuracy obtained is 92.679% with 50 epochs. This shows that there is a steady increase in the accuracies with increasing values of epochs.

Hyperparameter combinations with train test split (80:20 ratio)

The following are the hyperparameter combinations and the respective accuracies of the model.

```
mlp = MLPClassifier(
    activation = 'tanh',
    hidden_layer_sizes = (10,),
    learning_rate = 'constant',
    solver= 'sgd',
    momentum = 0.9,
    early_stopping= True,
    validation_fraction = 0.1,
    max_iter=100)

mlp.fit(X1_train, y1_train)
y_pred = mlp.predict(X1_test)
accuracy = accuracy_score(y1_test,
print("Model Accuracy - {}%".format
```

```
Model Accuracy - 85.94000000000001%
```

```
mlp = MLPClassifier(
    activation = 'tanh',
    hidden_layer_sizes = (10,),
    learning_rate = 'constant',
    solver= 'sgd',
    momentum = 0.9,
    early_stopping= True,
    validation_fraction = 0.1,
    max_iter=300)

mlp.fit(X1_train, y1_train)
y_pred = mlp.predict(X1_test)
accuracy = accuracy_score(y1_test
print("Model Accuracy - {}%".form
```

```
Model Accuracy - 90.93%
```

```
mlp = MLPClassifier(
    activation = 'logistic',
    hidden_layer_sizes = (10,),
    learning_rate = 'constant',
    solver= 'sgd',
    momentum = 0.9,
    early_stopping= True,
    validation_fraction = 0.1,
    max_iter=100)

mlp.fit(X1_train, y1_train)
y_pred = mlp.predict(X1_test)
accuracy = accuracy_score(y1_tes
print("Model Accuracy - {}%".for
```

```
Model Accuracy - 47.5%
```

```
mlp = MLPClassifier(
    activation = 'logistic',
    hidden_layer_sizes = (10,),
    learning_rate = 'constant',
    solver= 'sgd',
    momentum = 0.9,
    max_iter=100)

mlp.fit(X1_train, y1_train)
y_pred = mlp.predict(X1_test)
accuracy = accuracy_score(y1_test,
print("Model Accuracy - {}%".format
```

```
Model Accuracy - 49.33%
```

```
mlp = MLPClassifier(
    activation = 'tanh',
    hidden_layer_sizes = (10,),
    learning_rate = 'constant',
    solver= 'sgd',
    momentum = 0.9,
    max_iter=300)

mlp.fit(X1_train, y1_train)
y_pred = mlp.predict(X1_test)
accuracy = accuracy_score(y1_tes
print("Model Accuracy - {}%".for

Model Accuracy - 93.93%
```

```
mlp = MLPClassifier(
    activation = 'tanh',
    hidden_layer_sizes = (10,),
    learning_rate = 'constant',
    max_iter=300)

mlp.fit(X1_train, y1_train)
y_pred = mlp.predict(X1_test)
accuracy = accuracy_score(y1_test
print("Model Accuracy - {}%".form

Model Accuracy - 95.00999999999999%
```

```
mlp = MLPClassifier(
    activation = 'tanh',
    learning_rate = 'constant',
    max_iter=300)

mlp.fit(X1_train, y1_train)
y_pred = mlp.predict(X1_test)
accuracy = accuracy_score(y1_test
print("Model Accuracy - {}%".form

Model Accuracy - 97.0%
```

```
mlp = MLPClassifier(
    activation = 'tanh',
    learning_rate = 'constant')

mlp.fit(X1_train, y1_train)
y_pred = mlp.predict(X1_test)
accuracy = accuracy_score(y1_tes
print("Model Accuracy - {}%".for

Model Accuracy - 94.01%
```

```
mlp = MLPClassifier(
    activation = 'tanh',
    hidden_layer_sizes = (10,10),
    learning_rate = 'constant',
    max_iter=300)

mlp.fit(X1_train, y1_train)
y_pred = mlp.predict(X1_test)
accuracy = accuracy_score(y1_test,
print("Model Accuracy - {}%".forma

Model Accuracy - 91.18%
```

Hence it is observed that the best accuracy obtained with the training set is **97%** with **tanh activation function, constant learning rate and 300 epochs**.

Hyperparameter combinations with train and test data

```
mlp = MLPClassifier(
    activation = 'tanh',
    hidden_layer_sizes = (10,),
    learning_rate = 'constant',
    solver= 'sgd',
    momentum = 0.9,
    early_stopping= True,
    validation_fraction = 0.1,
    max_iter=100)

mlp.fit(normalized_df, train_y_df)
y_pred = mlp.predict(X_test_df)
accuracy = accuracy_score(y_test_df,
print("Model Accuracy - {}%".format(
```

```
Model Accuracy - 76.75999999999999%
```

```
mlp = MLPClassifier(
    activation = 'tanh',
    hidden_layer_sizes = (10,),
    learning_rate = 'constant',
    solver= 'sgd',
    momentum = 0.9,
    early_stopping= True,
    validation_fraction = 0.1,
    max_iter=300)

mlp.fit(normalized_df, train_y_df)
y_pred = mlp.predict(X_test_df)
accuracy = accuracy_score(y_test_df,
print("Model Accuracy - {}%".format(
```

```
Model Accuracy - 81.0%
```

```
mlp = MLPClassifier(
    activation = 'logistic',
    hidden_layer_sizes = (10,),
    learning_rate = 'constant',
    solver= 'sgd',
    momentum = 0.9,
    early_stopping= True,
    validation_fraction = 0.1,
    max_iter=100)

mlp.fit(normalized_df, train_y_df)
y_pred = mlp.predict(X_test_df)
accuracy = accuracy_score(y_test_df
print("Model Accuracy - {}%".format
```

```
Model Accuracy - 28.249999999999996%
```

```
mlp = MLPClassifier(
    activation = 'logistic',
    hidden_layer_sizes = (10,),
    learning_rate = 'constant',
    solver= 'sgd',
    momentum = 0.9,
    max_iter=100)

mlp.fit(normalized_df, train_y_df
y_pred = mlp.predict(X_test_df)
accuracy = accuracy_score(y_test_
print("Model Accuracy - {}%".form
```

```
Model Accuracy - 44.82%
```

```
mlp = MLPClassifier(
    activation = 'tanh',
    hidden_layer_sizes = (10,),
    learning_rate = 'constant',
    solver= 'sgd',
    momentum = 0.9,
    max_iter=300)

mlp.fit(normalized_df, train_y_df)
y_pred = mlp.predict(X_test_df)
accuracy = accuracy_score(y_test_c
print("Model Accuracy - {}%".forma
```

```
Model Accuracy - 82.85%
```

```
mlp = MLPClassifier(
    activation = 'tanh',
    hidden_layer_sizes = (10,),
    learning_rate = 'constant',
    max_iter=300)

mlp.fit(normalized_df, train_y_df)
y_pred = mlp.predict(X_test_df)
accuracy = accuracy_score(y_test_df,
print("Model Accuracy - {}%".format(r
```

```
Model Accuracy - 82.62%
```

```
mlp = MLPClassifier(
    activation = 'tanh',
    hidden_layer_sizes = (10,10),
    learning_rate = 'constant',
    max_iter=300)

mlp.fit(normalized_df, train_y_df)
y_pred = mlp.predict(X_test_df)
accuracy = accuracy_score(y_test_df, y_
print("Model Accuracy - {}%".format(rou
```

```
Model Accuracy - 77.99000000000001%
```

```
mlp = MLPClassifier(
    activation = 'tanh',
    learning_rate = 'constant',
    max_iter=300)

mlp.fit(normalized_df, train_y_df
y_pred = mlp.predict(X_test_df)
accuracy = accuracy_score(y_test_
print("Model Accuracy - {}%".form
```

```
Model Accuracy - 81.84%
```

```
mlp = MLPClassifier(
    activation = 'tanh',
    learning_rate = 'constant')

mlp.fit(normalized_df, train_y_df)
y_pred = mlp.predict(X_test_df)
accuracy = accuracy_score(y_test_df,
print("Model Accuracy - {}%".format(
```

```
Model Accuracy - 81.04%
```

It is observed that the best accuracy obtained with the training set is **82.85%** with **tanh activation function, constant learning rate, solver sgd, momentum 0.9, 1 hidden layer with 10 nodes and 300 epochs.**

Conclusions

- The accuracy of the model drops from 76.75% to 28.24% when switched from tanh to logistic activation function. This is so because logistic activation function is useful when dealing with binary classification problems whereas, tanh activation function works well for multiclass classification problems.
- This MLP classifier does not generalize will to new or unseen data. This is observed from the large gap in accuracies on evaluating model with train test split and train and test data.

Systematic Tuning of Hyperparameter

In our experiment we employed two automated hyperparameter tuning Grid Search and Random Search. Grid Search searches for all possible combination for the given parameters. While Random Search uses random combination to find the best solution for the model. Random Search works best with lower dimension as it can converge to a solution sooner however Grid Search ensures optimal set of parameters if the computation is completed. We found that Grid Search took comparatively longer time to compute when compared to Random Search. Given below are the results from our experiment.

Grid Search:

Parameter Given:

```
parameters = {
    'activation' : ['logistic', 'tanh', 'relu'],
    'hidden_layer_sizes' : [(10,), (10, 10), (10, 10, 10)],
    'solver' : ['adam', 'sgd'],
    'early_stopping' : [False, True],
    'learning_rate' : ['constant', 'adaptive'],
    'validation_fraction' : [0.1, 0.3],
    'max_iter': [50, 100],
}
```

From which we got the best result for Grid Search as {'activation': 'tanh', 'early_stopping': False, 'hidden_layer_sizes': (10,), 'learning_rate': 'adaptive', 'max_iter': 100, 'solver': 'adam', 'validation_fraction': 0.3} which gives an accuracy of 85.07%.

Random Search

Parameters given:

```
param_dist = {
    'hidden_layer_sizes': [(50,), (100,), (50, 50), (100, 50)],
    'activation': ['relu', 'tanh'],
    'alpha': [0.0001, 0.001, 0.01],
    'learning_rate': ['constant', 'adaptive'],
}
```

And for Random Search as {'learning_rate': 'constant', 'hidden_layer_sizes': (100,), 'alpha': 0.001, 'activation': 'tanh'} which gives an accuracy of 86.64%.

CONVOLUTIONAL NEURAL NETWORK

Conv1D

Number - number of filters or output channels increasing the value will increase the models capacity but will lead to overfitting when dataset is small

Kernel size - size of convolutional window -> increasing the size will allow to take in more data but will lose some of the details. Lesser values take in less data at a time but captures the finer details.

Activation - activation function applied to the output

Input shape - Input shape for the first layer in the model

Pooling

MaxPooling - performs MaxPooling on 1D data

Pool size -> size of window used for pooling

Flattening

Flatten - reshapes data to flatten the input

Dense - number of neurons in the layer (changing the value will change the complexity of the model)

Compile

Optimizer - used for training the model

Loss - function used during optimization

Metrics - used for training

Fitting

Epochs - number of iterations over the algorithm. Higher the epochs value higher the chances of better performance but too much will lead to overfitting

Batch size - defines no of samples that will go through the CNN before updating the models parameters. Larger batch size results in faster training but requires more memory.

Verbose - controls the amount of information displayed during training. It takes values of 0 (silent - no output), 1(progress bar), 2(one line per epoch).

Combination 1

Without layers :

Epochs = 10, batch size = 32, verbose=1

Without over sampling :

While running on the training dataset without cross validation:

```
CNN regression for dataset that has not been oversampled
Epoch 1/10
151/151 [==============================] - 6s 30ms/step - loss: 1.9535 - mse: 1.9535
Epoch 2/10
151/151 [==============================] - 5s 36ms/step - loss: 0.8540 - mse: 0.8540
Epoch 3/10
151/151 [==============================] - 5s 31ms/step - loss: 0.5776 - mse: 0.5776
Epoch 4/10
151/151 [==============================] - 5s 32ms/step - loss: 0.4895 - mse: 0.4895
Epoch 5/10
151/151 [==============================] - 5s 32ms/step - loss: 0.4279 - mse: 0.4279
Epoch 6/10
151/151 [==============================] - 5s 31ms/step - loss: 0.3688 - mse: 0.3688
Epoch 7/10
151/151 [==============================] - 5s 31ms/step - loss: 0.3229 - mse: 0.3229
Epoch 8/10
151/151 [==============================] - 5s 34ms/step - loss: 0.2902 - mse: 0.2902
Epoch 9/10
151/151 [==============================] - 6s 38ms/step - loss: 0.2714 - mse: 0.2714
Epoch 10/10
151/151 [==============================] - 5s 30ms/step - loss: 0.2459 - mse: 0.2459
38/38 [==============================] - 1s 15ms/step
Performance without cross-validation:
MSE: 0.4522586514867507
R-squared: 0.849809739029128
```

With 10-fold cross validation :

```
Performance with Time Series Cross-Validation:
Average MSE: 2.961875703775843
Average R-squared: -26.466459132206573
```

Over Sampling:

While running on the training dataset without cross validation:

```
CNN regression for dataset that has been oversampled
Epoch 1/10
410/410 [==============================] - 13s 30ms/step - loss: 1.5228 - mse: 1.5228
Epoch 2/10
410/410 [==============================] - 13s 33ms/step - loss: 0.3740 - mse: 0.3740
Epoch 3/10
410/410 [==============================] - 10s 24ms/step - loss: 0.3077 - mse: 0.3077
Epoch 4/10
410/410 [==============================] - 18s 43ms/step - loss: 0.2405 - mse: 0.2405
Epoch 5/10
410/410 [==============================] - 14s 34ms/step - loss: 0.1927 - mse: 0.1927
Epoch 6/10
410/410 [==============================] - 10s 24ms/step - loss: 0.1863 - mse: 0.1863
Epoch 7/10
410/410 [==============================] - 10s 24ms/step - loss: 0.1599 - mse: 0.1599
Epoch 8/10
410/410 [==============================] - 12s 30ms/step - loss: 0.1517 - mse: 0.1517
Epoch 9/10
410/410 [==============================] - 18s 43ms/step - loss: 0.1358 - mse: 0.1358
Epoch 10/10
410/410 [==============================] - 15s 37ms/step - loss: 0.1254 - mse: 0.1254
103/103 [==============================] - 2s 13ms/step
Performance without cross-validation:
MSE: 0.15569934304892644
R-squared: 0.9807425721989764
```

With 10 fold cross-validation :

```
Performance with Time Series Cross-Validation:
Average MSE: 1.2835283085250933
Average R-squared: -2.4909279677447875
```

Conclusion 1 :
Oversampling does not seem to significantly improve model performance in this case.
The model struggles with time series cross-validation, as indicated by the large negative R-squared values, suggesting that the model might not generalize well across different time series folds.

Combination 2

Without Oversampling :

With layer - **[(Conv1D(32, kernel_size=3, activation='relu', input_shape=input_shape)), (Flatten()), (Dense(1)) and (MaxPooling1D(pool_size=2))]**

Epochs = 10, batch size = 32, verbose=1

While running on the training dataset without cross validation:

```
CNN regression for dataset that has not been oversampled
Epoch 1/10
151/151 [==============================] - 8s 28ms/step - loss: 1.9049 - mse: 1.9049
Epoch 2/10
151/151 [==============================] - 4s 25ms/step - loss: 0.8560 - mse: 0.8560
Epoch 3/10
151/151 [==============================] - 3s 19ms/step - loss: 0.6326 - mse: 0.6326
Epoch 4/10
151/151 [==============================] - 3s 21ms/step - loss: 0.5260 - mse: 0.5260
Epoch 5/10
151/151 [==============================] - 3s 21ms/step - loss: 0.5001 - mse: 0.5001
Epoch 6/10
151/151 [==============================] - 3s 19ms/step - loss: 0.4596 - mse: 0.4596
Epoch 7/10
151/151 [==============================] - 3s 18ms/step - loss: 0.4412 - mse: 0.4412
Epoch 8/10
151/151 [==============================] - 3s 17ms/step - loss: 0.4025 - mse: 0.4025
Epoch 9/10
151/151 [==============================] - 3s 23ms/step - loss: 0.3853 - mse: 0.3853
Epoch 10/10
151/151 [==============================] - 3s 21ms/step - loss: 0.3656 - mse: 0.3656
38/38 [==============================] - 1s 13ms/step
Performance without cross-validation:
MSE: 0.5036649253763475
R-squared: 0.8327382652925004
```

With 10 fold cross-validation :

```
Performance with Time Series Cross-Validation:
Average MSE: 2.7617319311713873
Average R-squared: -23.182395127457795
```

With oversampling :

While running on the training dataset without cross validation:

```
CNN regression for dataset that has been oversampled
Epoch 1/10
410/410 [==============================] - 13s 27ms/step - loss: 1.7158 - mse: 1.7158
Epoch 2/10
410/410 [==============================] - 9s 21ms/step - loss: 0.4370 - mse: 0.4370
Epoch 3/10
410/410 [==============================] - 9s 22ms/step - loss: 0.3427 - mse: 0.3427
Epoch 4/10
410/410 [==============================] - 11s 26ms/step - loss: 0.2877 - mse: 0.2877
Epoch 5/10
410/410 [==============================] - 13s 31ms/step - loss: 0.2624 - mse: 0.2624
Epoch 6/10
410/410 [==============================] - 11s 26ms/step - loss: 0.2407 - mse: 0.2407
Epoch 7/10
410/410 [==============================] - 11s 26ms/step - loss: 0.2047 - mse: 0.2047
Epoch 8/10
410/410 [==============================] - 13s 31ms/step - loss: 0.1919 - mse: 0.1919
Epoch 9/10
410/410 [==============================] - 11s 26ms/step - loss: 0.1806 - mse: 0.1806
Epoch 10/10
410/410 [==============================] - 10s 25ms/step - loss: 0.1721 - mse: 0.1721
103/103 [==============================] - 5s 9ms/step
Performance without cross-validation:
MSE: 0.1993890421420984
R-squared: 0.9753388806389496
```

With 10 fold cross-validation :

```
Performance with Time Series Cross-Validation:
Average MSE: 1.3893941630393412
Average R-squared: -2.877819737081823
```

Conclusion 2 :
Oversampling has a positive impact on model performance, leading to better generalization, stability, and accuracy in predictions across different splits in time series cross-validation. However, the performance with time series cross-validation indicates potential variability in model predictions across different time series splits.

Combination 3

Without Oversampling :

With layer - **[(Conv1D(64, kernel_size=3, activation=sigmoid, input_shape=input_shape)), (Flatten()), (Dense(1)) and (MaxPooling1D(pool_size=2))]**

Epochs = 8, batch size = 32, verbose=2

While running on the training dataset without cross validation:

```
CNN regression for dataset that has not been oversampled
Epoch 1/8
151/151 - 13s - loss: 35.9661 - mse: 35.9661 - 13s/epoch - 85ms/step
Epoch 2/8
151/151 - 12s - loss: 3.4276 - mse: 3.4276 - 12s/epoch - 79ms/step
Epoch 3/8
151/151 - 12s - loss: 3.2709 - mse: 3.2709 - 12s/epoch - 81ms/step
Epoch 4/8
151/151 - 15s - loss: 3.5890 - mse: 3.5890 - 15s/epoch - 97ms/step
Epoch 5/8
151/151 - 11s - loss: 3.7098 - mse: 3.7098 - 11s/epoch - 74ms/step
Epoch 6/8
151/151 - 12s - loss: 4.5727 - mse: 4.5727 - 12s/epoch - 78ms/step
Epoch 7/8
151/151 - 13s - loss: 3.4832 - mse: 3.4832 - 13s/epoch - 83ms/step
Epoch 8/8
151/151 - 11s - loss: 3.4253 - mse: 3.4253 - 11s/epoch - 73ms/step
38/38 [==============================] - 1s 23ms/step
Performance without cross-validation:
MSE: 6.846356484361218
R-squared: -1.2736017624108547
```

With 10 fold cross-validation :

```
Performance with Time Series Cross-Validation:
Average MSE: 3.673682060927235
Average R-squared: -19.419989905793337
```

With oversampling :
While running on the training dataset without cross validation:

```
CNN regression for dataset that has been oversampled
Epoch 1/8
410/410 - 32s - loss: 18.3912 - mse: 18.3912 - 32s/epoch - 79ms/step
Epoch 2/8
410/410 - 30s - loss: 5.5512 - mse: 5.5512 - 30s/epoch - 74ms/step
Epoch 3/8
410/410 - 26s - loss: 3.2125 - mse: 3.2125 - 26s/epoch - 63ms/step
Epoch 4/8
410/410 - 24s - loss: 2.8035 - mse: 2.8035 - 24s/epoch - 58ms/step
Epoch 5/8
410/410 - 38s - loss: 2.5008 - mse: 2.5008 - 38s/epoch - 93ms/step
Epoch 6/8
410/410 - 27s - loss: 1.8468 - mse: 1.8468 - 27s/epoch - 66ms/step
Epoch 7/8
410/410 - 27s - loss: 1.9072 - mse: 1.9072 - 27s/epoch - 65ms/step
Epoch 8/8
410/410 - 26s - loss: 1.8922 - mse: 1.8922 - 26s/epoch - 63ms/step
103/103 [==============================] - 4s 39ms/step
Performance without cross-validation:
MSE: 1.261530438227429
R-squared: 0.8439695964206839
```

With 10 fold cross validation:

```
Performance with Time Series Cross-Validation:
Average MSE: 7.06999164655619
Average R-squared: -64.3422322571104
```

Conclusion 3 :

Oversampling improved the model's performance on the test set but led to reduced stability and generalization across different splits in time series cross-validation. The non-oversampled dataset's model performs better in terms of cross-validation, indicating more robustness in handling variations across time series splits.

Conclusion

In conclusion, our CNN ran with different distinct hyperparameters. We experimented with two activation functions, namely "relu" and "sigmoid," along with epochs set at both "8" and "10". Additionally, we explored the impact of different Conv1D configurations, for both "32" and "64".

CNN on oversampled data achieved the best MSE score of approximately **0.156**.
CNN on oversampled data attained the highest R-squared score of approximately **0.981**.

Models trained on the oversampled dataset demonstrated a better performance, showcasing higher R-squared values and lower MSE values which indicates a good fit to the data. Considering both MSE and R-squared as important metrics, the CNN model for the oversampled dataset appears to be the most effective and well-performing model among those evaluated.