Appendix 2

Source code for MLP algorithm and evaluation functionality

```python
# Local import
from data-processing.py import *

# Library imports
import pandas as pd
import numpy as np
from sklearn.metrics import *
from sklearn.model_selection import train_test_split

import seaborn as sns
import matplotlib.pyplot as plt
import plotly.express as px
import plotly.graph_objects as go

# Loading datas
labels = ['Lagged', 'MA', 'WMA', 'MA-Lagged', 'WMA-Lagged'] # names of each datasets

def load_datasets():
    """
    Excel files for each dataset are read into a
    dataframe an stored in a dictionary for easy
    access and use
    """
    datasets = dict()
    for lb in labels:
        new_df = pd.read_excel(f"River-Data-{lb}.xlsx")
        new_df.drop(["Unnamed: 0"], axis=1, inplace=True)
        datasets[lb] = new_df

    return datasets

data = load_datasets() # a dataframe for each dataset in a dictionary called data

# Plotting functions
def plot_correlation_matrix(corr_data, title, figsize=(16,6), mask=False):
    """
    Utility function for plotting a correlation heatmap of a given feature set
    """
    if mask:
        mask = np.triu(np.ones_like(corr_data, dtype=bool))
    plt.figure(figsize=figsize, dpi=500)
    heatmap = sns.heatmap(corr_data, vmin=-1, vmax=1, annot=True, mask=mask)
    heatmap.set_title(title)
    plt.show()

def plot_predictions(preds_df, standardised=False):
    """
    Utiltity function for plotting model predictions against actual value
    """
    preds_col = "Predicted Values"
    vals_col = "Actual Values"
    if standardised:
        preds_col += " (Standardised)"
        vals_col += " (Standardised)"

    line_plt = px.line(preds_df, y=vals_col)
    scatter_plt = px.scatter(preds_df, y=preds_col, color_discrete_sequence=["#ff0000"])

    go.Figure(line_plt.data + scatter_plt.data, layout={"title": "Actual vs Predicted Values"}).show()

    # Basic ANN class for MLP models
class BasicAnn:
    def __init__(self, layers, max_st_val, min_st_val, activ_func="sigmoid"):
        self.layers = layers
        self.num_layers = len(layers)
        self.max_val = max_st_val
```

```
        self.max_val = max_st_val
        self.min_val = min_st_val
        self.activ_func = activ_func

        weight_shapes = [(layers[i-1],layers[i]) for i in range(1, len(layers))]
        self.weights = {
            f"W{i+1}": np.random.standard_normal(s)/s[0]**0.5
            for i, s in enumerate(weight_shapes)
        } # weights are stored as matrices that are implemented as 2D numpy arrays
        self.biases = {
            f"B{i+1}": np.random.randn(l,1)/l**0.5
            for i, l in enumerate(layers[1:])
        } # biases are also stored as matrices that are implemented as 2D numpy arrays

    def activation(self, x):
        """
        Function to return value with the selected activation
        """
        if self.activ_func == "sigmoid":
            return 1/(1+np.exp(-x))
        elif self.activ_func == "tanh":
            return (np.exp(x)-np.exp(-x))/(np.exp(x)+np.exp(-x))
        elif self.activ_func == "relu":
            return x * (x > 0)
        elif self.activ_func == "linear":
            return x

    def activation_deriv(self, a):
        """
        Function to return value with the derivative of the selected activation
        """
        if self.activ_func == "sigmoid":
            return a * (1 - a)
        elif self.activ_func == "tanh":
            return 1 - a**2
        elif self.activ_func == "relu":
            return 1 * (a > 0)
        elif self.activ_func == "linear":
            return np.ones(a.shape)

    def train(self, features, targets, epochs=1000, learning_rate=0.1, val_set=None):
        """
        Function will train the model using the standard backpropogation algorithm
        and return a dataframe storing various error metrics for the model on the
        training set and, possibly, a validation set if that is given
        """
        results = pd.DataFrame()
        real_targets = unstandardise_value(targets, self.max_val, self.min_val)
        num_targets = len(targets)

        for _ in range(epochs):
            # Forward pass
            activations = self.forward_pass(features)

            # Error calculation
            output_layer = activations[f"A{self.num_layers - 1}"]
            real_preds = unstandardise_value(output_layer, self.max_val, self.min_val)
            error_data = { # storing error metrics for both standardised and unstandardised data
                "mse": mean_squared_error(real_targets, real_preds),
                "rmse": mean_squared_error(real_targets, real_preds, squared=False),
                "mae": mean_absolute_error(real_targets, real_preds),
                "r_sqr": r2_score(real_targets, real_preds),
                "st_mse": mean_squared_error(targets, output_layer),
                "st_rmse": mean_squared_error(targets, output_layer, squared=False),
                "st_mae": mean_absolute_error(targets, output_layer),
                "st_r_sqr": r2_score(targets, output_layer)
            }
```

```python
            if val_set:
                # if there is a validation set the prediction error of the model
                # on the validation set will be stored
                r, err = self.predict(val_set[0].to_numpy(), val_set[1].to_numpy())
                error_data.update({f"val_{col}": err[col][0] for col in err.columns})

            results = results.append(error_data, ignore_index=True)

            # Backward pass (backpropagation algorithm)
            deltas = self.compute_deltas(activations, targets, output_layer)
            self.update_weights(deltas, activations, features, num_targets, learning_rate)

        return results

    def predict(self, test_inputs, st_actual_outputs, actual_outputs=None):
        """
        Runs a forward pass of the network with the newly configured weights
        and biases and returns a dataframe comparing the predicted values
        to actual values as well as a dataframe with various error metrics
        """
        # Forward pass
        activations = self.forward_pass(test_inputs)
        st_preds = activations[f"A{self.num_layers - 1}"]

        # Comparing predicted values with actual values
        if actual_outputs is None:
            actual_outputs = unstandardise_value(st_actual_outputs, self.max_val, self.min_val)

        preds = unstandardise_value(st_preds, self.max_val, self.min_val)

        results = pd.DataFrame(
            data={
                "Actual Values": actual_outputs.flatten(),
                "Predicted Values": preds.flatten(),
                "Actual Values (Standardised)": st_actual_outputs.flatten(),
                "Predicted Values (Standardised)": st_preds.flatten(),
            }
        )

        # Error calculation
        results["Absolute Error"] = abs(results["Actual Values"] - results["Predicted Values"])
        st_absolute_err = abs(results["Actual Values (Standardised)"] - results["Predicted Values (Standardised)"])
        results["Absolute Error (Standardised Values)"] = st_absolute_err

        error_metrics = pd.DataFrame(data={
            "mse": [mean_squared_error(actual_outputs, preds)],
            "rmse": [mean_squared_error(actual_outputs, preds, squared=False)],
            "mae": [mean_absolute_error(actual_outputs, preds)],
            "r_sqr": [r2_score(actual_outputs, preds)],
            "st_mse": [mean_squared_error(st_actual_outputs, st_preds)],
            "st_rmse": [mean_squared_error(st_actual_outputs, st_preds, squared=False)],
            "st_mae": [mean_absolute_error(st_actual_outputs, st_preds)],
            "st_r_sqr": [r2_score(st_actual_outputs, st_preds)]
        })

        return results, error_metrics

    def forward_pass(self, features):
        """
        Runs a forward pass of neural network through repeated
        multiplication of weights and bias matrices. Returns
        list of each activation layer including the output layer.
        """
        activation = self.activation(np.dot(features, self.weights["W1"]) + self.biases["B1"].T)
        activations = {"A1": activation}
        for i in range(2, self.num_layers):
```

```python
            activation = self.activation(np.dot(activation, self.weights[f"W{i}"]) + self.biases[f"B{i}"].T)
            activations[f"A{i}"] = activation

        return activations

    def compute_deltas(self, activations, targets, output_layer):
        """
        Computes errors between layers for backprogation.
        Returns a dictionary of lists which contain the errors
        for each node in each layer.
        """
        output_err = targets - output_layer
        output_delta = output_err * self.activation_deriv(output_layer)
        deltas = {"dw1": output_delta}

        for i in range(self.num_layers - 1, 1, -1):
            dw = deltas[f"dw{self.num_layers - i}"]
            act = activations[f"A{i-1}"]
            w = self.weights[f"W{i}"]
            deltas[f"dw{self.num_layers - i + 1}"] = np.dot(dw, w.T) * self.activation_deriv(act)

        return deltas

    def update_weights(self, deltas, activations, features, num_targets, l_rate):
        """
        Updates weights and biases according to given errors, activations
        and the chosen learning rate
        """
        delta = deltas[f"dw{self.num_layers - 1}"]
        self.weights["W1"] += l_rate * (np.dot(features.T, delta)) / num_targets
        self.biases["B1"] += l_rate * (np.dot(delta.T, np.ones((num_targets, 1)))) / num_targets

        for i in range(2, self.num_layers):
            act = activations[f"A{i-1}"]
            dw = deltas[f"dw{self.num_layers - i}"]
            self.weights[f"W{i}"] += l_rate * (np.dot(act.T, dw)) / num_targets
            self.biases[f"B{i}"] += l_rate * np.dot(dw.T, np.ones((num_targets, 1))) / num_targets


# Function to build, train and test a model
def build_train_test(feature_set, feature_cols, target_cols, layers=("auto", 1), activ_func="linear", epochs=1000, l_rate=0.1):
    """
    Function to build, train and test MLP models
    """
    # Splitting and standardising datasets to create standardised and unstandardised
    # training, validation and testing sets.
    train_val_set, test_set = train_test_split(feature_set, test_size=0.2)
    st_train_val_set = standardise_columns(train_val_set, train_val_set.columns)
    st_test_set = standardise_columns(test_set, test_set.columns)

    # Preparing features and targets for training and testing
    features = st_train_val_set[feature_cols]
    targets = st_train_val_set[target_cols]

    X_train, X_val, y_train, y_val = train_test_split(features, targets, test_size=0.25)
    X_test, y_test = st_test_set[feature_cols], st_test_set[target_cols]

    # Getting standardisation values for targets
    min_val = train_val_set[target_cols].min()[0]
    max_val = train_val_set[target_cols].max()[0]

    # Building model
    if layers[0] == "auto":
        # if the size of the input layer is not specified
        # then it will be set to the number of predictors

        layers = (len(feature_cols),) + layers[1:]
```

```
    ann = BasicAnn(layers, max_val, min_val, activ_func)

    # Training model
    training_results = ann.train(
        X_train.to_numpy(),
        y_train.to_numpy(),
        val_set=(X_val, y_val), # training with a validation set
        epochs=epochs,
        learning_rate=l_rate
    )

    # Predicting model
    prediction_results = ann.predict(
        X_test.to_numpy(),
        y_test.to_numpy(),
        actual_outputs=test_set[target_cols].to_numpy()
    )

    predictions, error_metrics = prediction_results[0], prediction_results[1]

    return {
        "training_results": training_results,
        "final_test_results": predictions,
        "error_metrics": error_metrics,
        "model": ann
    }
```