CSCI 5622 - Homework 2

Aritra Chakrabarty

March 13, 2024

1 Library and data import

The first step is to import all the necessary libraries for usage throughout this assignment.

```
[ ]: # Importing all necessary libraries into notebook
     import pandas as pd
     import numpy as np
     import statistics as stat
     import seaborn as sns
     import sklearn
     import matplotlib.pyplot as plt
```

Since I am having some issues directly reading the *csv* using pandas, related to UTF-8, I decided to check the encoding of the .csv first.

NOTE: *The .csv file was updated and the chardet portion as show below is no longer necessary, but I am keeping it in here for use in future homework assignments.*

File importing works perfectly fine now.

```
[ ]: import chardet
     with open('hw-2.csv','rb') as file:
         print(chardet.detect(file.read(512)))
         file.close()
         #limit the number of bytes being read
         #do not want to go through all 22 MB of data
```

```
{'encoding': 'ascii', 'confidence': 1.0, 'language': ''}
```

We import the .csv file below, and display the columns to ensure they match the *provided instructions.*

```
[ ]: # Import the .csv file for usage
     df = pd.read_csv("hw-2.csv")
     #Due to the data not being in UTF-8 format, we have to use ISO-8859-1 - No␣
      ↪longer the case
     df.columns
```

```
[ ]: Index(['Depthm', 'Salnty', 'O2ml_L', 'STheta', 'O2Sat', 'Oxymol', 'ChlorA',
            'Phaeop', 'PO4uM', 'SiO3uM', 'NO2uM', 'NH3uM', 'C14As1', 'C14As2',
            'DarkAs', 'LightP', 'Year'],
           dtype='object')
```

2 Data Exploration 1

Plot 2-D scatter plots and compute the Pearson's correlation coefficient between features and the outcome. Which features are the most predictive of salinity level?

2.1 Plotting all Features and Correlations

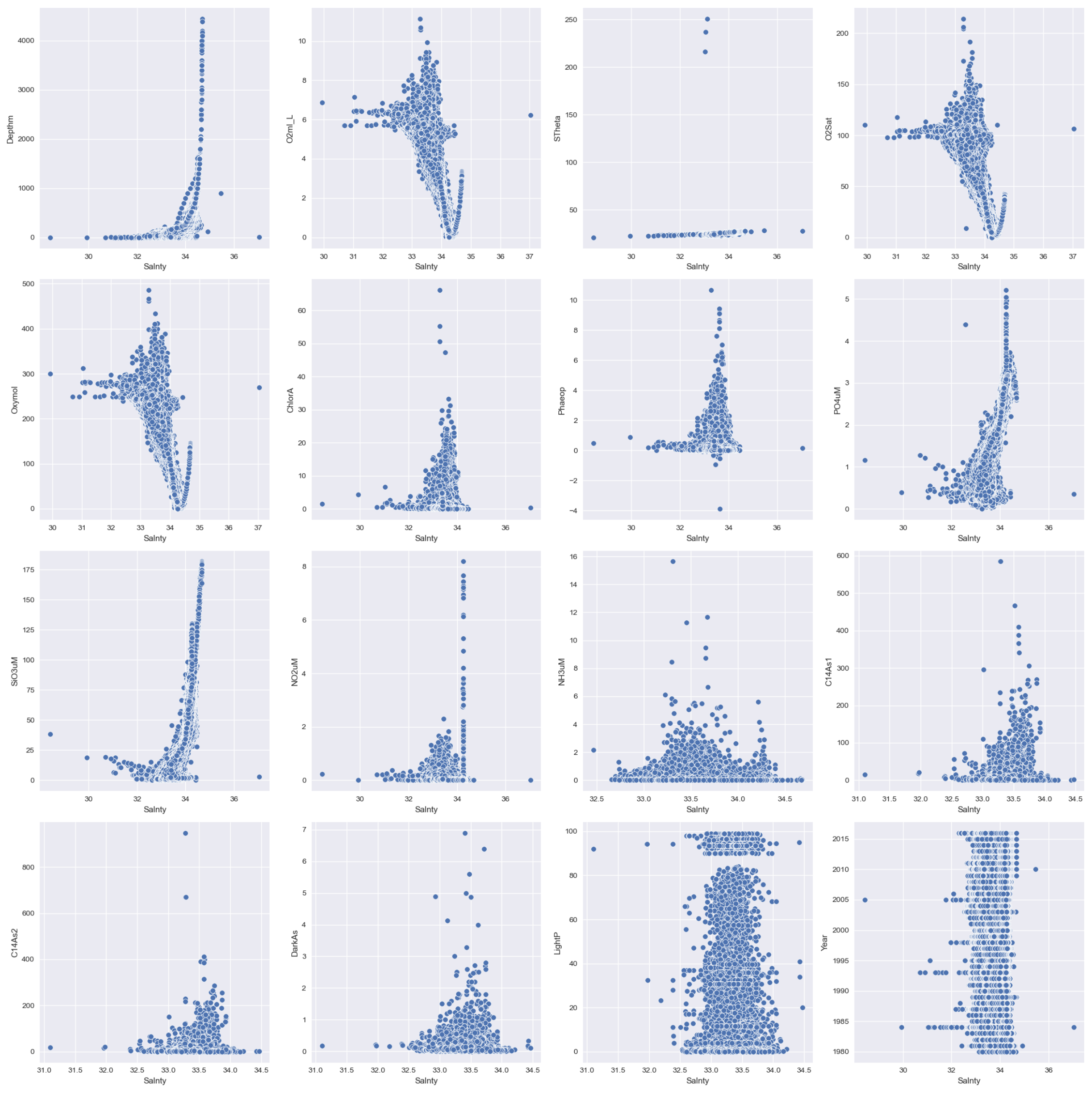
Since there are a total of 16 features exculding *Salinity*, we can plot a basic grid of scatterplots using the **seaborn** library. The cell below uses a seaborn theme for plot representation, and uses *subplot* form **matplotlib** to get an organized output with the plots in a grid.

[ ]:
```
plt.style.use('seaborn-v0_8')

numsubplots = 16
figcols = 4
figrows = int(np.ceil(numsubplots/figcols))
fig, axes = plt.subplots(nrows=figrows, ncols=figcols, figsize=(20,20))

plotnumber = 0
for column in df.columns:
    if column!='Salnty':
        row, col = divmod(plotnumber, figcols)
        ax = axes[row, col]
        sns.scatterplot(x=df['Salnty'], y=df[column], ax=ax)
        ax.tick_params(axis='x')
        plotnumber += 1

plt.tight_layout()
```



To get a better idea of the correlations, we use pearson's correlation to obtain how each feature relates to *Salinity*. **Pandas** has a function to make a correlation matrix, which we use to obtain each correlation, linked below.

pandas.DataFrame.corr

```
[ ]: #Pearson correlation
     for column in df.columns:
         if column != 'Salnty':
             pearson_matrix = df[['Salnty', column]].corr(method='pearson')␣
      ↪#relating salnty to other cols
             correlation_value = pearson_matrix.loc['Salnty', column]
```

```
        print(f"Correlation between 'Salnty' and '{column}':␣
  ↪{correlation_value}")
```

```
Correlation between 'Salnty' and 'Depthm': 0.7150357558238011
Correlation between 'Salnty' and 'O2ml_L': -0.9137646831628342
Correlation between 'Salnty' and 'STheta': 0.6136725558411762
Correlation between 'Salnty' and 'O2Sat': -0.9034429826802403
Correlation between 'Salnty' and 'Oxymol': -0.9140322469454201
Correlation between 'Salnty' and 'ChlorA': -0.06713344927597674
Correlation between 'Salnty' and 'Phaeop': -0.04622498395677666
Correlation between 'Salnty' and 'PO4uM': 0.8975988894062825
Correlation between 'Salnty' and 'SiO3uM': 0.8654148732234828
Correlation between 'Salnty' and 'NO2uM': -0.19566336662153697
Correlation between 'Salnty' and 'NH3uM': -0.1187469917425513
Correlation between 'Salnty' and 'C14As1': 0.19751420081486085
Correlation between 'Salnty' and 'C14As2': 0.18596024631363084
Correlation between 'Salnty' and 'DarkAs': 0.1413377429200838
Correlation between 'Salnty' and 'LightP': -0.028925799637096145
Correlation between 'Salnty' and 'Year': -0.028696586774554448
```

2.2 High Correlation Features against Salinity

Pearson correlation coefficient is said to be *strong* if the value lies between ± 0.5 and ± 1. It is of a *moderate* correlation when the values are between ± 0.3 and ± 0.5. So, we should only make use of the values that indicate a strong correlation.

Different sources mention different cutoff ranges for what is considered to be *strong* correleation.

Therefore, the cell below uses a cutoff range to determine the features which are most correleated to *Salinity*.

```
[ ]: #cutoff range
pm_range = 0.5

#make a new correlation matrix
corr_mat = df.corr(method='pearson')
salnty_correlations = corr_mat['Salnty'].drop('Salnty') #there is no need to␣
  ↪compare salnty with itself

#keep only the correlations that are within the absolute range
high_salnty_correlations = salnty_correlations[abs(salnty_correlations) >␣
  ↪pm_range]

#unwrap the pandas dataframe and reset to only keep correlation to salinity
high_salnty_correlations = high_salnty_correlations.reset_index().
  ↪rename(columns={'index': 'Feature', 'Salnty': 'Correlation'})

print(high_salnty_correlations)
```

```
  Feature  Correlation
0  Depthm     0.715036
1  O2ml_L    -0.913765
2  STheta     0.613673
3   O2Sat    -0.903443
4  Oxymol    -0.914032
5   PO4uM     0.897599
6  SiO3uM     0.865415
```

We keep the *high_salnty_correlations* array in memory for usage in future sections to improve performance.

3 Data Exploration 2

Compute matrix $C \in \mathbb{R}^{16\times16}$ *that contains the Pearson's correlation coefficients between all pairs of features. Visualise the matrix* C *using a heatmap. Which features are the most correlated to each other?*

3.1 Heatmap for all features

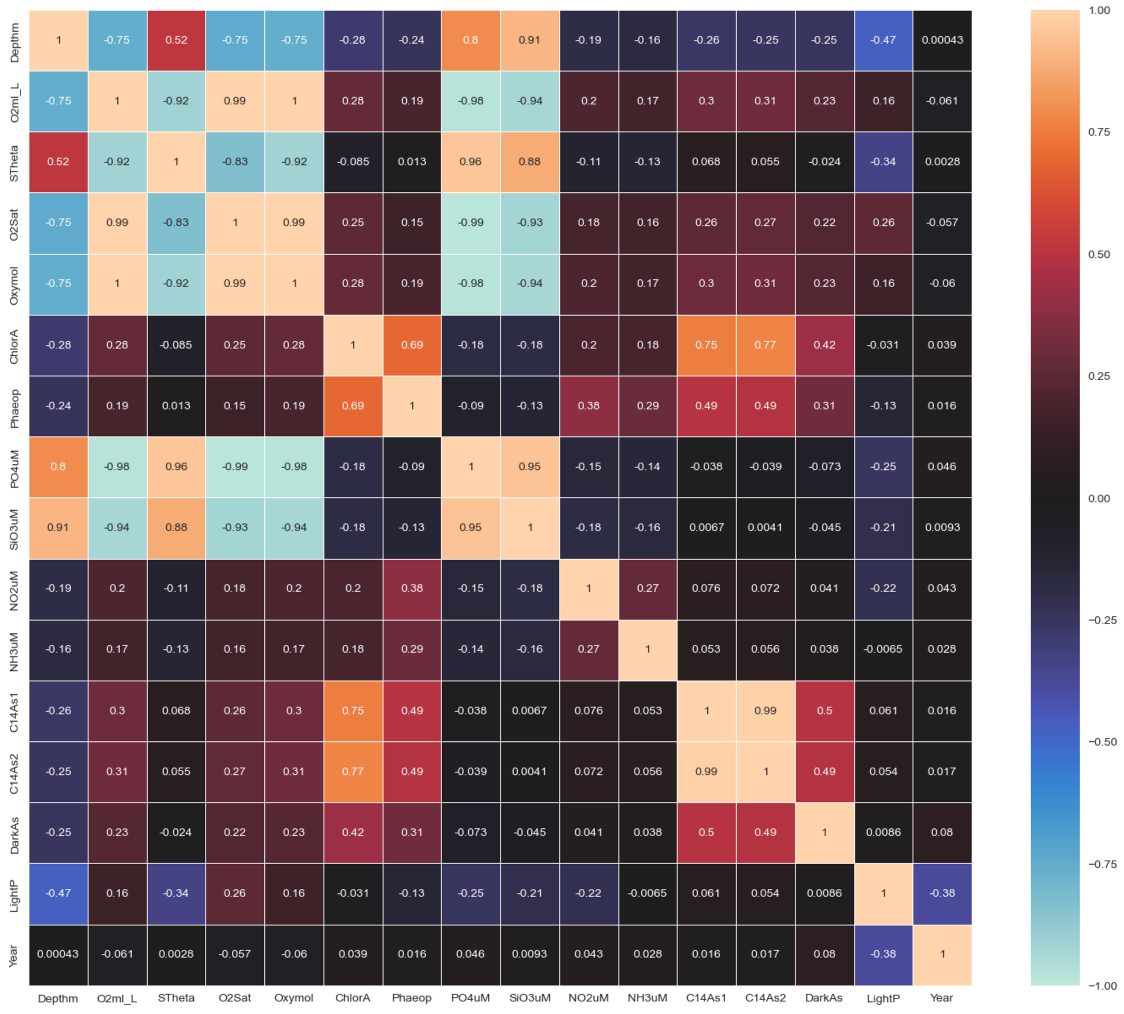
Seaborn has in-built heatmaps which can take an input of the *correlation matrix*, and provides a well-formatted and easily digestable output. The doumentation for it is linked below. Mentioning the *vmin*, *vmax*, and *center* helps us get an organized color scheme to help understand the relations well.

seaborn.heatmap

```
[ ]: #Used resources:
     #https://www.geeksforgeeks.org/how-to-create-a-correlation-matrix-using-pandas/

     df2 = df.drop(columns='Salnty', axis=1) #remove salinity as we are considering␣
      ↪multicolinearity
     correlation_matrix = df2.corr(method='pearson')

     plt.style.use('seaborn-v0_8')
     plt.figure(figsize=(18,15))
     sns.heatmap(correlation_matrix, annot=True, linewidth=.5, vmin=-1, vmax=1,␣
      ↪center=0)
     plt.show()
```



	Depthm	O2ml_L	STheta	O2Sat	Oxymol	ChlorA	Phaeop	PO4uM	SiO3uM	NO2uM	NH3uM	C14As1	C14As2	DarkAs	LightP	Year
Depthm	1	-0.75	0.52	-0.75	-0.75	-0.28	-0.24	0.8	0.91	-0.19	-0.16	-0.26	-0.25	-0.25	-0.47	0.00043
O2ml_L	-0.75	1	-0.92	0.99	1	0.28	0.19	-0.98	-0.94	0.2	0.17	0.3	0.31	0.23	0.16	-0.061
STheta	0.52	-0.92	1	-0.83	-0.92	-0.085	0.013	0.96	0.88	-0.11	-0.13	0.068	0.055	-0.024	-0.34	0.0028
O2Sat	-0.75	0.99	-0.83	1	0.99	0.25	0.15	-0.99	-0.93	0.18	0.16	0.26	0.27	0.22	0.26	-0.057
Oxymol	-0.75	1	-0.92	0.99	1	0.28	0.19	-0.98	-0.94	0.2	0.17	0.3	0.31	0.23	0.16	-0.06
ChlorA	-0.28	0.28	-0.085	0.25	0.28	1	0.69	-0.18	-0.18	0.2	0.18	0.75	0.77	0.42	-0.031	0.039
Phaeop	-0.24	0.19	0.013	0.15	0.19	0.69	1	-0.09	-0.13	0.38	0.29	0.49	0.49	0.31	-0.13	0.016
PO4uM	0.8	-0.98	0.96	-0.99	-0.98	-0.18	-0.09	1	0.95	-0.15	-0.14	-0.038	-0.039	-0.073	-0.25	0.046
SiO3uM	0.91	-0.94	0.88	-0.93	-0.94	-0.18	-0.13	0.95	1	-0.18	-0.16	0.0067	0.0041	-0.045	-0.21	0.0093
NO2uM	-0.19	0.2	-0.11	0.18	0.2	0.2	0.38	-0.15	-0.18	1	0.27	0.076	0.072	0.041	-0.22	0.043
NH3uM	-0.16	0.17	-0.13	0.16	0.17	0.18	0.29	-0.14	-0.16	0.27	1	0.053	0.056	0.038	-0.0065	0.028
C14As1	-0.26	0.3	0.068	0.26	0.3	0.75	0.49	-0.038	0.0067	0.076	0.053	1	0.99	0.5	0.061	0.016
C14As2	-0.25	0.31	0.055	0.27	0.31	0.77	0.49	-0.039	0.0041	0.072	0.056	0.99	1	0.49	0.054	0.017
DarkAs	-0.25	0.23	-0.024	0.22	0.23	0.42	0.31	-0.073	-0.045	0.041	0.038	0.5	0.49	1	0.0086	0.08
LightP	-0.47	0.16	-0.34	0.26	0.16	-0.031	-0.13	-0.25	-0.21	-0.22	-0.0065	0.061	0.054	0.0086	1	-0.38
Year	0.00043	-0.061	0.0028	-0.057	-0.06	0.039	0.016	0.046	0.0093	0.043	0.028	0.016	0.017	0.08	-0.38	1

3.2 High Multicollinearity between features

Very strong correlation between features can cause multiple issues in a linear regression model that we are trying to build in the upcoming sections due to the following reasons:

- Redundancy: Strong correlation could mean that by having two similar features, we are providing redundant data, that can not only slow down the performance, it can also interfere with the weights as we could be "double counting" them.
- Instability: The $\mathbf{w_n}$ values in the model that we form could be biased towards certain features.
- Overfitting: Some overfitting can occur if we are essentially double counting certain features.

So, we should have a list of features with high multicolinearity in order to remove them for future usage.

```
[ ]: #list of all high correlation features
     def show_unique_high_correlations(correlation_matrix, range):
```

```
    #unstack corr matrix and reset index to make a new dataframe giving feature␣
↪pairs w correlation
    correlation_pairs = correlation_matrix.unstack().reset_index()
    correlation_pairs.columns = ['FeatureA', 'FeatureB', 'Correlation']

    #keep only pairs with absolute values higher than range, then exclude all␣
↪self correlations
    filtered_pairs = correlation_pairs[((correlation_pairs['Correlation'].abs()␣
↪> range)
                                        & (correlation_pairs['FeatureA'] !=␣
↪correlation_pairs['FeatureB']))]

    #keep only one direction, to remove repeated duplicates
    filtered_pairs_no_duplicates = filtered_pairs[filtered_pairs['FeatureA'] <␣
↪filtered_pairs['FeatureB']]

    #sort by the first column just to make the output look a bit more␣
↪organised, and print
    unique_pairs = filtered_pairs_no_duplicates.sort_values(by='FeatureA')
    print(unique_pairs)
```

```
[ ]: show_unique_high_correlations(correlation_matrix, pm_range)
```

```
    FeatureA FeatureB  Correlation
181   C14As1   ChlorA     0.753438
188   C14As1   C14As2     0.988701
197   C14As2   ChlorA     0.772853
86    ChlorA   Phaeop     0.693789
8     Depthm   SiO3uM     0.913041
1     Depthm   O2ml_L    -0.750968
2     Depthm   STheta     0.521110
3     Depthm    O2Sat    -0.746203
7     Depthm    PO4uM     0.796075
4     Depthm   Oxymol    -0.751041
56     O2Sat   SiO3uM    -0.927659
55     O2Sat    PO4uM    -0.991510
52     O2Sat   Oxymol     0.994698
50     O2Sat   STheta    -0.830720
49     O2Sat   O2ml_L     0.994579
24    O2ml_L   SiO3uM    -0.936703
23    O2ml_L    PO4uM    -0.983838
20    O2ml_L   Oxymol     0.999999
18    O2ml_L   STheta    -0.917149
66    Oxymol   STheta    -0.917604
71    Oxymol    PO4uM    -0.983984
72    Oxymol   SiO3uM    -0.936599
114    PO4uM   STheta     0.957313
```

```
120    PO4uM   SiO3uM     0.945701
40    STheta   SiO3uM     0.881600
```

Looking at the output of high multicolinearity, it makes sense. For example O_2 *Saturation* has high positive correlation with $O_2 \frac{ml}{L}$ which is just a different measure of the oxygen. In order to not double count things, the whole list can be reduced to include features that make sense.

Note: For the first linear regression we perform using OLS, we will be using *ALL* features. The *Experimentation* section will utilise a feature downsizing to determine the best feature subset to improve performance.

4 Data Cleaning

Please clean the dataset. You can identify and substitute any missing data, normalize the features, etc.

Pandas has enough tools to clean up the dataset as we see fit. I have also taken the liberty to make functions that *clean* and *split* the dataset in this section. In order to experiment with feature combinations in the future, I have followed a structure to:

- Substitute missing values with the mean
- Normalize the scale for each feature
- Divide the dataset according to year.

Since we are dealing with *ALL* the features for the first calculation of salinity prediction, we can take the data subset as all features.

```
[ ]: alldata = df.copy() #making a copy to avoid errors in the future
```

We already know there are no categorical features to deal with, so no conversions will be necessary.

```
[ ]: alldata.isna().sum()
```

```
[ ]: Depthm          0
     Salnty       3270
     O2ml_L      26357
     STheta       5510
     O2Sat       26964
     Oxymol      26970
     ChlorA     116829
     Phaeop     116832
     PO4uM       35625
     SiO3uM      34752
     NO2uM       37090
     NH3uM      260319
     C14As1     310849
     C14As2     310867
     DarkAs     302632
     LightP     306630
```

```
Year           0
dtype: int64
```

As we can see, there are a lot of missing datapoints.

While we *could* remove datapoints that have missing data in any feature, that would lead us to way to small of a dataset in this situation. Therefore, the strategy for now is to take the *mean* of the available data for that feature and *substitute* that into the empty spots.

Fortunately, *Year* does not have any missing datapoints, so data *splitting* will not be affected by any *cleaning* and *normalising* actions.

The function below *substitutes* the missing data with the mean and also *normalizes* the data using the *StandardScaler* fucntion from the **scikit-learn** library. The documentation is linked below. The function uses the formula $z = \frac{x-\mu}{\sigma}$ to normalise. It is essentially providing the *z-score normalisation*.

Since I re-use the function in future sections, I have also added an option to work with a *binarized* version of *Salinity*, that we use with logistic regression.

sklearn.preprocessing.StandardScaler

```
[ ]: from sklearn.preprocessing import StandardScaler
     def clean_data_and_normalize(data_subset, logistic=False):
         for column in data_subset.columns:
             data_subset[column].fillna(data_subset[column].mean(), inplace=True)

             if logistic == False:
                 if column!='Year' and column!='Salnty':
                     data_subset[[column]] = StandardScaler().
      ↪fit_transform(data_subset[[column]])
             elif logistic == True:
                 if column!='Year' and column!='Salnty_binarized':
                     data_subset[[column]] = StandardScaler().
      ↪fit_transform(data_subset[[column]])
             else:
                 print("SOMETHING HORRIBLE HAS GONE WRONG IN DATA CLEANING")
```

```
[ ]: clean_data_and_normalize(alldata)
     alldata.isna().sum()
```

```
[ ]: Depthm    0
     Salnty    0
     O2ml_L    0
     STheta    0
     O2Sat     0
     Oxymol    0
     ChlorA    0
     Phaeop    0
     PO4uM     0
     SiO3uM    0
```

```
NO2uM      0
NH3uM      0
C14As1     0
C14As2     0
DarkAs     0
LightP     0
Year       0
dtype: int64
```

After running the *clean_data_and_normalize()* function, we no longer have any missing data points, as displayed in the above cell.

I have also provided a function below that splits the entered dataset into $\mathbf{X}_{\text{train}}$, $\vec{y}_{\text{train}}$, $\mathbf{X}_{\text{dev}}$, $\vec{y}_{\text{dev}}$, $\mathbf{X}_{\text{test}}$, and $\vec{y}_{\text{test}}$ subsets for usage in future sections. The function also takes a *binarized* version of *Salinity*, if being used in a logistic regression problem.

```
[ ]: def train_dev_test_split(dataset, logistic=False):
         if logistic == False:
             name = 'Salnty'
         else: name = 'Salnty_binarized'

         traindataset = dataset[(dataset['Year'] >= 1980) & (dataset['Year'] <=␣
      ↪2010)]
         X_train = traindataset.drop([name, 'Year'], axis=1)
         y_train = traindataset[name]

         devdataset = dataset[(dataset['Year'] >= 2011) & (dataset['Year'] <= 2013)]
         X_dev = devdataset.drop([name, 'Year'], axis=1)
         y_dev = devdataset[name]

         testdataset = dataset[(dataset['Year'] >= 2014) & (dataset['Year'] <= 2016)]
         X_test = testdataset.drop([name, 'Year'], axis=1)
         y_test = testdataset[name]

         return X_train, y_train, X_dev, y_dev, X_test, y_test
```

Note*: The order in which we normalise and split the data is important as that can affect the means. So, some variance can occur due to the order of operations.*

5 Predicting Salinity

The goal of this question is to predict water salinity using the considered biochemical features. Implement a linear regression model using the ordinary least squares (OLS) solution. Report the coefficient of determination R^2, Pearson's correlation r, and mean absolute error MAE between the actual and predicted salinity values on the development and testing sets.

5.1 Obtain the OLS solution

Since we already processed the *alldata* matrix in the previous step, we can go ahead and *split* the data as shown below.

```
[ ]: X_train, y_train, X_dev, y_dev, X_test, y_test = train_dev_test_split(alldata)
```

The closed form *OLS* solution is calculated per the formula $\vec{w}^* = (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top \vec{y}$.

We use the **numpy** library for calculating the weights as per the function below. Since the $\mathbf{X}_{\text{train}}$ matrix is a pandas dataframe object, we frist convert it to a numpy matrix.

Then we add an array of ones in front of the current data set in order to convert $\mathbf{X} = \begin{Bmatrix} \mathbf{x_1}^\top \\ \vdots \\ \mathbf{x_N}^\top \end{Bmatrix} \in \mathbb{R}^{N_{\text{train}} \times D}$ to $\mathbf{X} = \begin{Bmatrix} 1, \mathbf{x_1}^\top \\ \vdots \\ 1, \mathbf{x_N}^\top \end{Bmatrix} \in \mathbb{R}^{N_{\text{train}} \times D+1}$

```
[ ]: def ols_weights(X_train, y_train):
         #Convert X_train to numpy and make the data vertical
         #initialise the vector of ones
         ones = np.ones((X_train.shape[0], 1))
         #use hstack to add the ones in front of the xdata
         X_train_np = np.hstack((ones, X_train))

         #make y a vertical vector
         #-1 reshape to let numpy determine the number of rows based on length
         y_train_np = y_train.to_numpy().reshape(-1,1)

         #calculating weights using provided OLS formula
         weights = np.linalg.pinv(X_train_np.T @ X_train_np) @ X_train_np.T @␣
      ↪y_train_np

         return weights
```

After obtaining the vector of weights $\vec{w}^*$, we can then calculate the predictions using the formula $\vec{y}_{\text{pred}} = \vec{w}^\top \mathbf{X}$. This can be for either the development set or the testing set.

```
[ ]: #function to predict y values given the above weights using y = wT * X

     def predict_y_values(X, weights):
         #reshape input X
         ones = np.ones((X.shape[0], 1))
         X_np = np.hstack((ones, X))

         #adjusted formula for the matrix shape I have after the manipulations
         y_values = X_np @ weights

         return y_values
```

```
y_pred_dev = predict_y_values(X_dev, ols_weights(X_train, y_train))
y_pred_test = predict_y_values(X_test, ols_weights(X_train, y_train))
```

5.2 Evaluate the OLS Solution

We are using three different metrics as per the *instructions*. The function below sets up model evaluations to compare $\vec{y}_{\text{pred}}$ to $\vec{y}_{\text{true}}$. By comparing to the ground truth we can obtain an understanding of how the model performs.

- R^2 error: The R^2 coefficient of determination is a statistical measure of how *well* the regression predictions approximate the real data points. An R^2 value of 1 means perfect correlation. However, the value can range from $-\infty$ to 1. To get a value less than 0, the performance is horrible, and the *residual sum of squares* exceeds the *total sum of squares* in the formula $R^2 = 1 - \frac{SS_{\text{residual}}}{SS_{\text{total}}}$. This page offers more details.
- Pearson correlation coefficient r: This can be calculated using the formula $\frac{\sum_{i=1}^{n}(x_i-\bar{x})(y_i-\bar{y})}{\sqrt{\sum_{i=1}^{n}(x_i-\bar{x})^2}\sqrt{\sum_{i=1}^{n}(y_i-\bar{y})^2}}$. The values can range from $+1$, meaning a perfect positive relationship, -1, meaning a perfect negative relationship. 0 means no relationship.
- Mean Absolute Error: This uses the formula $\sum_{i=1}^{D}|x_i - y_i|$. The values can range from 0 to ∞. The closer to 0, the better the evaluation.

The following functions are used for evaluation:

- sklearn.metrics.r2_score
- sklearn.metrics.mean_absolute_error
- scipy.stats.pearsonr

[ ]:
```
#setting up for model evaluations
from sklearn.metrics import r2_score, mean_absolute_error
from scipy.stats import pearsonr

def model_evaluations(y_values, y_correct):
    #uses the function directly
    r_squared = r2_score(y_correct, y_values)

    #need to flatten the array using ravel, can't be a vector
    pearson_r, pval= pearsonr(y_correct.ravel(), y_values.ravel()) #returns a␣
 ↪tuple

    #uses the function directly again
    mae = mean_absolute_error(y_correct, y_values)

    return r_squared, pearson_r, mae
```

[ ]:
```
r_squared_dev, pearson_r_dev, mae_dev = model_evaluations(y_pred_dev, y_dev)
```

```
print(f"Development Set\nR^2: {r_squared_dev},\nPearson's correlation r:␣
↪{pearson_r_dev},\nMean absolute Error: {mae_dev}")
```

```
Development Set
R^2: 0.9212452827273497,
Pearson's correlation r: 0.9625305308463652,
Mean absolute Error: 0.07112108497675049
```

```
[ ]: r_squared_test, pearson_r_test, mae_test = model_evaluations(y_pred_test,␣
     ↪y_test)
     print(f"Test Set\nR^2: {r_squared_test},\nPearson's correlation r:␣
     ↪{pearson_r_test},\nMean absolute Error: {mae_test}")
```

```
Test Set
R^2: -16.923256511690877,
Pearson's correlation r: 0.20074852765442017,
Mean absolute Error: 0.11203710228175817
```

As we can see, the model fits greatly to the training data, and performs very well when compared to the *Development* Set. However, it performs much worse in comparison to the *Test* Set. The only thing we can estimate from this evaluation is that the model is too overfit to the *training* data, and is not generalized enough for predicting from the *test* data.

We can make some changes in the *Experimentation* section to obtain a better model.

6 Discussion of predictions

Please discuss the estimated coefficients of the linear regression model. How might these findings support stakeholders in making informed decisions? Furthermore, in what ways could this model and its outcomes be utilized to educate the public?

We can observe the weights for each feature as shown in the output of the code cell below.

```
[ ]: weights = ols_weights(X_train, y_train)

     features = np.vstack(np.insert(X_train.columns,0,['Bias']))

     print(np.concatenate([weights,features], axis=1))
```

```
[['33.707570045014485' 'Bias']
 ['0.02054513470010297' 'Depthm']
 ['-0.2192437286854702' 'O2ml_L']
 ['0.6513113499660901' 'STheta']
 ['1.8879513474922087' 'O2Sat']
 ['-1.635575391441779' 'Oxymol']
 ['0.011837114801396875' 'ChlorA']
 ['0.0017997392465610319' 'Phaeop']
 ['0.08966291312584707' 'PO4uM']
 ['-0.14763261064069128' 'SiO3uM']
 ['-0.0020047924834751887' 'NO2uM']
 ['0.0026680631482935424' 'NH3uM']
 ['0.005705331767043302' 'C14As1']
 ['-0.006170660355563908' 'C14As2']
 ['0.004278788917914712' 'DarkAs']
 ['0.0018486154329883727' 'LightP']]
```

The largest weight is given to w_0 which is the *bias* term as seen that it has a value of 33.7075.

O_2Sat, $O_2\mu mol/kg$, S_θ, O_2ml/L, and $SiO_3\mu M$ have the next highest **absolute** weights at 1.8880, 1.63557, 0.6513, 0.2192, and, 0.1476 respectively. These values have absolute weights that are much higher in comparison to the rest of the factors and thus affect the Salinity more, according to the current model. Everything else has weights that are factors lower in comparison to these main features. There is certainly some issue with giving *more* importance to the Oxygen related features, as the model is probably becoming too dependent on it.
These higher absolute weight features relate to the features that have a high correlation to *Salinity* from the section *Data Exploration 1.*

These findings can support stakeholders like environmental agencies and marine biologists because the model's insight provides info on the leading causes behind salinity. Stakeholders can effectively change rules and regulations. They can also make informed decisions regarding climate change assessments, conservation strategies, and marine management.

This model and the outcomes provided can help educate the public on what the leading causes are behind conservation efforts. If the public can be convinced of the factors leading to harmful effects on marine life (via salinity), people may be more willing to change their practices. As public knowledge increases, so does the effort for conservation.

7 Experimenting

Based on your findings from questions (i) and (ii), experiment with different feature combinations using linear and non-linear regression models. Please use the development data for hyper-parameter tuning (i.e., to assess the feature selection and the linear/non-linear regression models) based on the mean absolute error MAE between the actual and predicted salinity values. Please report and discuss the MAE results from the experiments on the development data. Using the model that gave the best MAE in the development data, please report the MAE on the test set.

7.1 Best feature combination

7.1.1 Choosing a smaller subset of features

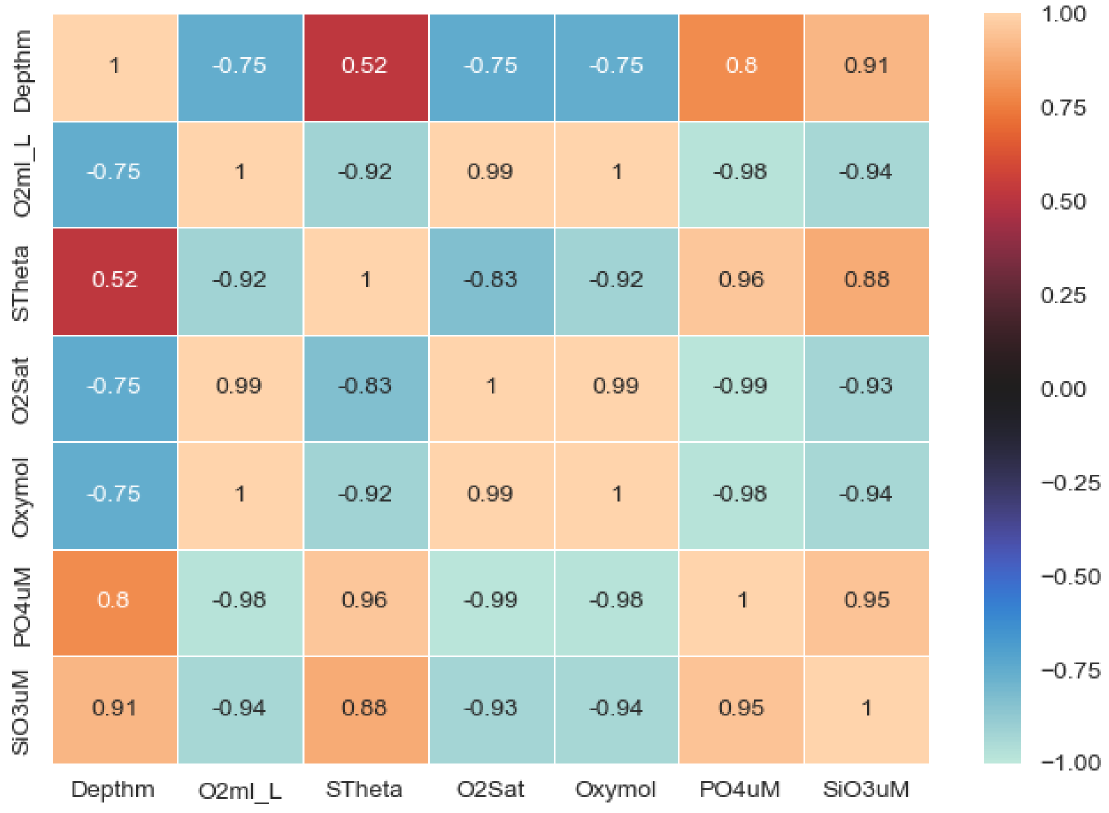
We see from the above sections, that not every feature affects the *weights* equally. We also do not want a situation where the model is overfit to the training data. One method, is to decrease the number of features being used, so that even if the model performance on the *development* dataset drops, the performance on the *testing* dataset is better. This would mean the model is more *generalized.*

Seeing as we have a list of features that greatly affect salinity, let's form a correlation matrix heatmap with only those features. This will help us minimise redundant features that we use for our calculations.

```
[ ]: feature_list = high_salnty_correlations['Feature'].tolist()
     data_subset_high_corr = df[feature_list]

     sns.heatmap(data_subset_high_corr.corr(method='pearson'), annot=True,␣
      ↪linewidth=.5, vmin=-1, vmax=1, center=0)
```

```
[ ]: <Axes: >
```



	Depthm	O2ml_L	STheta	O2Sat	Oxymol	PO4uM	SiO3uM
Depthm	1	-0.75	0.52	-0.75	-0.75	0.8	0.91
O2ml_L	-0.75	1	-0.92	0.99	1	-0.98	-0.94
STheta	0.52	-0.92	1	-0.83	-0.92	0.96	0.88
O2Sat	-0.75	0.99	-0.83	1	0.99	-0.99	-0.93
Oxymol	-0.75	1	-0.92	0.99	1	-0.98	-0.94
PO4uM	0.8	-0.98	0.96	-0.99	-0.98	1	0.95
SiO3uM	0.91	-0.94	0.88	-0.93	-0.94	0.95	1

```
[ ]: mlcol_range = 0.8
     show_unique_high_correlations(data_subset_high_corr.corr(method='pearson'),␣
      ↪mlcol_range)
```

```
   FeatureA FeatureB  Correlation
6    Depthm   SiO3uM     0.913041
22    O2Sat   O2ml_L     0.994579
23    O2Sat   STheta    -0.830720
25    O2Sat   Oxymol     0.994698
26    O2Sat    PO4uM    -0.991510
27    O2Sat   SiO3uM    -0.927659
9    O2ml_L   STheta    -0.917149
11   O2ml_L   Oxymol     0.999999
```

```
12   O2ml_L    PO4uM   -0.983838
13   O2ml_L   SiO3uM   -0.936703
30   Oxymol   STheta   -0.917604
33   Oxymol    PO4uM   -0.983984
34   Oxymol   SiO3uM   -0.936599
37    PO4uM   STheta    0.957313
41    PO4uM   SiO3uM    0.945701
20   STheta   SiO3uM    0.881600
```

We are only going to keep certain features. O2Sat, O2ml_L, and O2micromol/Kg all measure similar things. Only one is enough. Oxygen saturation is also highly correlated with PO4 and SiO3, both molecules have oxygen in them. Perhaps Depth, Stheta, and Oxygen saturation are the most unique ones that we should keep.

The function below will run a linear regression on the chosen features from the original dataset in order to allow us experiment with a smaller subset of features.

```
[ ]: def run_linear_regression(df, features):
         data = df[features].copy()

         clean_data_and_normalize(data)

         X_train, y_train, X_dev, y_dev, X_test, y_test = train_dev_test_split(data)

         y_pred_dev = predict_y_values(X_dev, ols_weights(X_train, y_train))
         y_pred_test = predict_y_values(X_test, ols_weights(X_train, y_train))

         r_squared_dev, pearson_r_dev, mae_dev = model_evaluations(y_pred_dev, y_dev)
         r_squared_test, pearson_r_test, mae_test = model_evaluations(y_pred_test,␣
      ↪y_test)

         return r_squared_dev, pearson_r_dev, mae_dev, r_squared_test,␣
      ↪pearson_r_test, mae_test
```

Keeping only *Depth*, S_θ, and $O_2 Saturation$, we can see the results below.

```
[ ]: feat = ['O2Sat','Depthm','STheta','Salnty','Year']

     r_squared_dev, pearson_r_dev, mae_dev, r_squared_test, pearson_r_test, mae_test␣
      ↪= run_linear_regression(df,feat)
     print(f"Development Set\nR^2: {r_squared_dev},\nPearson's correlation r:␣
      ↪{pearson_r_dev},\nMean absolute Error: {mae_dev}"
           f"\n\nTest Set\nR^2: {r_squared_test},\nPearson's correlation r:␣
      ↪{pearson_r_test},\nMean absolute Error: {mae_test}")
```

```
Development Set
R^2: 0.837367152397408,
Pearson's correlation r: 0.9161641057411477,
Mean absolute Error: 0.11572734889066594
```

```
Test Set
R^2: -2.0465891460108363,
Pearson's correlation r: 0.4220623600585193,
Mean absolute Error: 0.14643089606216936
```

While the performance has improved, it can be even better. We can try removing S_θ from the feature list as it only had a 0.63 correlation in section *Data Exploration 1*.

```
[ ]: feat = ['O2Sat','Depthm','Salnty','Year']

     r_squared_dev, pearson_r_dev, mae_dev, r_squared_test, pearson_r_test, mae_test␣
      ↪= run_linear_regression(df,feat)
     print(f"Development Set\nR^2: {r_squared_dev},\nPearson's correlation r:␣
      ↪{pearson_r_dev},\nMean absolute Error: {mae_dev}"
           f"\n\nTest Set\nR^2: {r_squared_test},\nPearson's correlation r:␣
      ↪{pearson_r_test},\nMean absolute Error: {mae_test}")
```

```
Development Set
R^2: 0.8373961450235929,
Pearson's correlation r: 0.9178585155072176,
Mean absolute Error: 0.11747345125981559

Test Set
R^2: 0.8001572625059774,
Pearson's correlation r: 0.9063913118543812,
Mean absolute Error: 0.13610134143424152
```

Lowering the features down to just two, $O_2 saturation$, and Depth makes a huge difference in the performance of the model. While it no longer performs as well on the *development* set, we can say the model is a lot more generalized as it has similar performance on **both** the *development* and *test* datasets.

7.1.2 Confirming Hypothesis by iterating through options

While I would be satisfied with the performance of the model with the features being $O_2 saturation$, and Depth, I still want to check if there is a better combination of features available. The function below iterates through all different combinations I feed in to find the one that has the highest performance to see if my hypothesis is correct or not.

Since we are obtaining the best performance for both the *development* and *testing* results, this is not a regular procedure. This subsection and the code only exists to see how well the feature selection above does is in comparison.

```
[ ]: from itertools import combinations
     import numpy as np

     def find_best_feature_combination_linear(df, all_features, max_features=None):
         #need to drop year and salinity for iterations
```

```
    features_to_consider = [feat for feat in all_features if feat not in␣
↪['Year', 'Salnty']]

    best_r2 = -np.inf
    best_combination = []
    best_combo_result = None

    if max_features is None:
        max_features = len(features_to_consider)

    for r in range(1, max_features + 1):
        for feature_combination in combinations(features_to_consider, r):
            totalList = list(feature_combination)
            totalList.append('Year')
            totalList.append('Salnty')

            r_squared_dev, pearson_r_dev, mae_dev, r_squared_test,␣
↪pearson_r_test, mae_test = run_linear_regression(df, totalList)

            if r_squared_test + r_squared_dev > best_r2:
                best_r2 = r_squared_test + r_squared_dev
                #simply just choosign the features that offer the best r^2
                best_combination = feature_combination
                best_combo_result = (r_squared_dev, pearson_r_dev, mae_dev,␣
↪r_squared_test, pearson_r_test, mae_test)

    print(f"Best feature combination: {best_combination}\n"
          f"Development set - R^2: {best_combo_result[0]}, Pearson's r:␣
↪{best_combo_result[1]}, MAE: {best_combo_result[2]}\n"
          f"Test set - R^2: {best_combo_result[3]}, Pearson's r:␣
↪{best_combo_result[4]}, MAE: {best_combo_result[5]}")
```

```
[ ]: feat = ['O2Sat','Depthm','STheta','O2ml_L','Oxymol','PO4uM','SiO3uM']
     find_best_feature_combination_linear(df, feat)
```

```
Best feature combination: ('O2Sat', 'Depthm', 'Oxymol', 'PO4uM')
Development set - R^2: 0.8600526444172428, Pearson's r: 0.9309933584366783, MAE:
0.11274012143574183
Test set - R^2: 0.8221373172608087, Pearson's r: 0.9238749577343719, MAE:
0.13115680915756742
```

Shockingly enough, from the limited subset of features, the best results were obtained by the combination of $O_2Saturation$, $Depth$, $Oxy\mu Mol$, and $PO_4\mu M$. While this list *does* add two features that I previously hypothesized to be redundant, it only performs better due to being a good fit with the development set. I would still prefer the list that is more generalized with $O_2Saturation$ and $Depth$. We also notice that the performance is not significantly better in comparison to what we obtained on the limited dataset.

For the *development* dataset, the improvement in R^2 from 0.84 to 0.86 exists, but the more generalized model can be argued to be better.
Similarly, for the *test* dataset, the improvement in R^2 was only from 0.80 to 0.82.

7.2 Regularization and Non-Linear Regression

We can try another method to obtain better performance, which is regularization. Instead of using manual feature choices according to correlation and multicollinearity, we can change the strength of the weights to obtain the best performance.

The code cell below sets up the data normalisation and splits to be used in further steps.

[ ]:
```
featlist = df.columns.to_list()
df_exp = df[featlist].copy()

clean_data_and_normalize(df_exp)

X_train, y_train, X_dev, y_dev, X_test, y_test = train_dev_test_split(df_exp)
```

7.2.1 L2 Regularization

I have used multiple external libraries for this portion.

sklearn.linear_model.Ridge is a linear regression model that takes in a *hyperparameter* α and performs a regression model where the loss function is the linear least squares function. Regularization is provided by the l2-norm.

sklearn.model_selection.GridSearchCV provides a simple way to do repetitive hyperparameter tuning without manual for loops on my part. It simply takes in a grid of parameters, a method, and the way to score performance, to return the best output. Finally, we can fit our data to that output in order to get a performance evaluation.

[ ]:
```
from sklearn.model_selection import GridSearchCV
from sklearn.linear_model import Ridge #uses l2 reg

param_grid = {'alpha': [0.001, 0.01, 0.1, 1.0, 10.0, 100.0]}

#the manual states the string 'neg_mean_absolute_error' is the way to get MAE
grid_search = GridSearchCV(Ridge(), param_grid,␣
  ↪scoring='neg_mean_absolute_error')

grid_search.fit(X_train, y_train)

print("Best alpha for l2 regularization:", grid_search.best_params_)

best_model_l2 = grid_search.best_estimator_
mae_dev_l2 = -grid_search.score(X_dev, y_dev)
print("MAE on development set with l2 regularization:", mae_dev_l2)
```

```
Best alpha for l2 regularization: {'alpha': 10.0}
MAE on development set with l2 regularization: 0.07143459924388597
```

7.2.2 L1 Regularization

Similar to l2 regularization, I have used external libraries for this portion. sklearn.linear_model.Lasso is also a linear regression model, but the loss function is different from l-2. The hyperparameter to be tuned is still α. I have also increased the maximum number of iterations from a *default* of 1,000 to 10,000 in order for it to converge, as I was getting warnings otherwise.

```
[ ]: from sklearn.linear_model import Lasso #l1 regularization

     param_grid = {'alpha': [0.001, 0.01, 0.1, 1.0, 10.0, 100.0]}

     grid_search = GridSearchCV(Lasso(max_iter=10000), param_grid,␣
      ↪scoring='neg_mean_absolute_error')
     grid_search.fit(X_train, y_train)

     print("Best alpha for l1 regularization:", grid_search.best_params_)

     best_model_l1 = grid_search.best_estimator_
     mae_dev_l1 = -grid_search.score(X_dev, y_dev)
     print("MAE on development set with l1 regularization:", mae_dev_l1)
```

```
Best alpha for l1 regularization: {'alpha': 0.001}
MAE on development set with l1 regularization: 0.07883076521041436
```

7.2.3 Non-Linear Regression with L2 Regularization

Seeing the results in the previous subsections, L2 regularization performs a bit better and also converges faster. Therefore, we will be using the function *Ridge()* after the polynomial conversion.

sklearn.preprocessing.PolynomialFeatures allows us to transform the input vector and get polynomials as the output in terms of degrees. For example, if the input had $[x_1, x_2]$, then the output provided would have $[1, x_1, x_2, x_1^2, x_2^2, x_1x_2]$. It has the hyper-parameter input as *degree* which determines how much the polynomials will change.

sklearn.pipeline.Pipeline then combines the transformed output from the *PolynomialFeatures()* function and allows us to evaluate it using regular L2 regression. This still has the same hyper-parameter α as before.

If all available features are to be used for the *PolynomialFeatures()* function, we end up having too many computations to perform. So, the total list has to be cut down first. For the sake of simplicity, we can just use $O_2 saturation$ and *Depth* for our features. Polynomial is also prone to easily overfitting from the experimentation that I have done, so the more generalized, the better.

```
[ ]: #Same as above, just cutting down on total features to save time in the␣
      ↪polynomial portion
     feat = ['O2Sat','Depthm','Salnty','Year']
```

```
df_polynomial = df[feat].copy()
clean_data_and_normalize(df_polynomial)
X_train_p, y_train_p, X_dev_p, y_dev_p, X_test_p, y_test_p =␣
 ↪train_dev_test_split(df_polynomial)
```

```
[ ]: from sklearn.preprocessing import PolynomialFeatures
     from sklearn.pipeline import make_pipeline

     pipeline = make_pipeline(PolynomialFeatures(), Ridge()) #transformer and␣
      ↪estimator

     #the grid allows us to try many different combinations
     param_grid = {
         'polynomialfeatures__degree': [2, 3, 4],
         'ridge__alpha': [0.001, 0.01, 0.1, 1.0, 10.0, 100.0]
     }

     grid_search = GridSearchCV(pipeline, param_grid,␣
      ↪scoring='neg_mean_absolute_error')

     grid_search.fit(X_train_p, y_train_p)

     print("Best parameters:", grid_search.best_params_)

     best_model_poly = grid_search.best_estimator_
     mae_dev_poly = -grid_search.score(X_dev_p, y_dev_p)
     print("Development set MAE with Polynomial Regression:", mae_dev_poly)
```

```
Best parameters: {'polynomialfeatures__degree': 4, 'ridge__alpha': 0.001}
Development set MAE with Polynomial Regression: 0.09708409447499868
```

7.3 Using Best Model on Test Set

We see in the above sections that the best performance on the development set was provided by the model obtained in l-2 regularization. In the interest of time and avoiding overfitting to the data, I will not be performing more experiments on the hyperparameters.

```
[ ]: y_pred = best_model_l2.predict(X_test)

     mae_test = mean_absolute_error(y_test, y_pred)

     print(f"Test set MAE with best model: {mae_test}")
```

```
Test set MAE with best model: 0.11208105130258542
```

8 Logistic Regression

Use the sample mean of the salinity outcome to binarize the data (i.e., assign samples with salinity larger than the mean to class 1 and samples with salinity lower than the mean to class -1). Run a logistic regression algorithm to classify between class 1 and -1. Use the development data for hyper-parameter tuning (i.e., to determine the regularization strength, regularization penalty term, etc.) based on the classification accuracy metric. After hyper-parameter tuning, report the accuracy of the classifier on the test data using the best combination that resulted from the development data.

We are going to use sklearn.linear_model.LogisticRegression in this section. It uses multiple penalties or regressions. The *regularization strength* from the equation for logistic regression is here as C, which is the inverse of the regularization strength. The *solver* being used here is *liblenear* due to us trying both *l1* and *l2* norm for penalties, and the manual page states that the choice of the algorithm depends on the penalty being chosen. It also helps that *liblenear* is recommended for small to medium datasets.

```
[ ]: from sklearn.linear_model import LogisticRegression
     from sklearn.model_selection import GridSearchCV
     from sklearn.metrics import accuracy_score
```

We can binarize the dataset as per the code cell below. We create a new column, and if the *Salinity* there is greater than the mean, we make it +1, otherwise we make it -1.

```
[ ]: mean_salnty = df['Salnty'].mean()
     df_logistic = df.copy()
     df_logistic['Salnty_binarized'] = np.where(df_logistic['Salnty'] > mean_salnty,␣
      ↪1, -1)
```

The cell below normalizes the dataset, and splits it accordingly.

```
[ ]: feature_list =␣
      ↪['Year','O2Sat','Depthm','STheta','O2ml_L','Oxymol','PO4uM','SiO3uM','Salnty_binarized']
     data_sub = df_logistic[feature_list].copy()

     clean_data_and_normalize(data_sub, logistic=True)
     X_train, y_train, X_dev, y_dev, X_test, y_test = train_dev_test_split(data_sub,␣
      ↪logistic=True)
```

The cell below performs hyper-parameter tuning, with C being evenly spaced on the scale from 10^{-4} to 10^{4}. It takes quite a bit of time using all the features that heavily affect salinity. We can cut down the run time by simply lowering the number of features.

```
[ ]: param_grid = {
         'C': np.logspace(-4, 4, 20),
         'penalty': ['l1', 'l2'],
         'solver': ['liblinear']
     }

     logreg = LogisticRegression()# no parameter changes necessary from default
     grid_search = GridSearchCV(logreg, param_grid, scoring='accuracy')
```

```
grid_search.fit(X_train, y_train)

print("Best hyper-parameters:", grid_search.best_params_)
```

```
Best hyper-parameters: {'C': 206.913808111479, 'penalty': 'l2', 'solver':
'liblinear'}
```

The cells below show the performance of the model on both the *validation* set and the final *test* set.

[ ]:
```
best_logistic_model = LogisticRegression(**grid_search.best_params_)

best_logistic_model.fit(X_train, y_train)

y_pred_dev = best_logistic_model.predict(X_dev)
dev_acc = accuracy_score(y_dev, y_pred_dev)

print("Development set accuracy with best hyperparameters:", dev_acc)
```

```
Development set accuracy with best hyperparameters: 0.9756027080581242
```

[ ]:
```
y_pred_test = best_logistic_model.predict(X_test)
test_acc = accuracy_score(y_test, y_pred_test)

print("Test set accuracy with best hyperparameters:", test_acc)
```

```
Test set accuracy with best hyperparameters: 0.9824329510083998
```

Given the accuracy of the model on both the *development* dataset and the *test* dataset, we can say that the performance of the model is great. More time for experimentation would be needed to get a higher accuracy score. Interestingly, the model performed better on the *test* set, but I would argue that the changes are negligible.