Wild Blueberry Yield Prediction

@Author : - Priyangshu Sarkar

Import Libraries

```
!pip install -q shap
```

```
     |████████████████████████████████| 564 kB 4.4 MB/s
```

```
## EDA libraries
import numpy as np
import pandas as pd
import·matplotlib.pyplot·as·plt
import·seaborn·as·sns

from scipy import stats
import shap

## feature engineering libraries
from sklearn.preprocessing import StandardScaler
from sklearn.feature_selection import SelectKBest
from sklearn.feature_selection import f_regression, mutual_info_regression
from sklearn.model_selection import train_test_split

## model preparation libraries
from sklearn.linear_model import LinearRegression
from sklearn.ensemble import RandomForestRegressor
from sklearn.tree import DecisionTreeRegressor
from xgboost import XGBRegressor

## model evaluation libraries
from sklearn.metrics import mean_absolute_error, r2_score, mean_squared_error
from sklearn import metrics
from sklearn.model_selection import RepeatedKFold

## model hyperparameter tuning
from sklearn.pipeline import Pipeline
from sklearn.model_selection import GridSearchCV

import joblib
```

Import Data

According to data dictionary, we have 3 unknown **fields, fruitset, fruitmass, seeds** having high correlation values with the target varialble, the **yield** value.

1.we have a data dictionary of mutual correlated values of each of the fields with yield values

2.as this clearly is a regression problem, we can perform feature selection on the data

3.there are the following approaches:

f_regression() parameter for selecting by correlation

mutual_info_regression() for selecting by information gain

raw data features kept intact

```
df=pd.read_csv("WildBlueberryPollinationSimulationData.csv")
```

```
df
```

	Row#	clonesize	honeybee	bumbles	andrena	osmia	MaxOfUpperTRange	MinOfUppe
0	0	37.5	0.750	0.250	0.250	0.250	86.0	
1	1	37.5	0.750	0.250	0.250	0.250	86.0	
2	2	37.5	0.750	0.250	0.250	0.250	94.6	
3	3	37.5	0.750	0.250	0.250	0.250	94.6	
4	4	37.5	0.750	0.250	0.250	0.250	86.0	
...	...	...	...	...	...	...	...	
772	772	10.0	0.537	0.117	0.409	0.058	86.0	
773	773	40.0	0.537	0.117	0.409	0.058	86.0	
774	774	20.0	0.537	0.117	0.409	0.058	86.0	
775	775	20.0	0.537	0.117	0.409	0.058	89.0	
776	776	20.0	0.537	0.117	0.409	0.058	89.0	

777 rows × 18 columns

```
df.head(20)
```

	Row#	clonesize	honeybee	bumbles	andrena	osmia	MaxOfUpperTRange	MinOfUpper
0	0	37.5	0.75	0.25	0.25	0.25	86.0	
1	1	37.5	0.75	0.25	0.25	0.25	86.0	
2	2	37.5	0.75	0.25	0.25	0.25	94.6	
3	3	37.5	0.75	0.25	0.25	0.25	94.6	
4	4	37.5	0.75	0.25	0.25	0.25	86.0	
5	5	37.5	0.75	0.25	0.25	0.25	86.0	
6	6	37.5	0.75	0.25	0.25	0.25	94.6	
7	7	37.5	0.75	0.25	0.25	0.25	94.6	
8	8	37.5	0.75	0.25	0.25	0.25	77.4	
9	9	37.5	0.75	0.25	0.25	0.25	77.4	
10	10	37.5	0.75	0.25	0.25	0.25	69.7	
11	11	37.5	0.25	0.25	0.25	0.25	86.0	
12	12	37.5	0.25	0.25	0.25	0.25	86.0	
13	13	37.5	0.25	0.25	0.25	0.25	94.6	
14	14	37.5	0.25	0.25	0.25	0.25	94.6	
15	15	37.5	0.25	0.25	0.25	0.25	86.0	

```
df.shape
```

```
(777, 18)
```

18	18	37.5	0.25	0.25	0.25	0.25	94.6

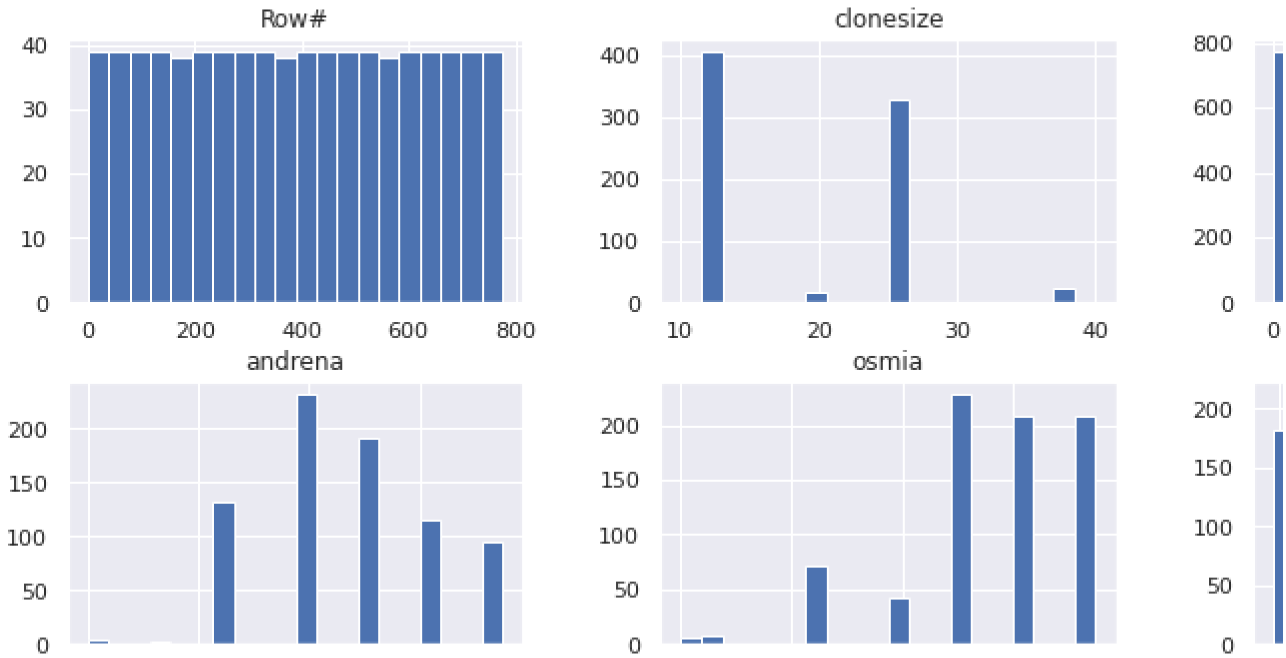
```
df.isna().sum()
```

```
Row#                    0
clonesize               0
honeybee                0
bumbles                 0
andrena                 0
osmia                   0
MaxOfUpperTRange        0
MinOfUpperTRange        0
AverageOfUpperTRange    0
MaxOfLowerTRange        0
MinOfLowerTRange        0
AverageOfLowerTRange    0
RainingDays             0
AverageRainingDays      0
fruitset                0
fruitmass               0
seeds                   0
yield                   0
dtype: int64
```

```
df.describe()
```

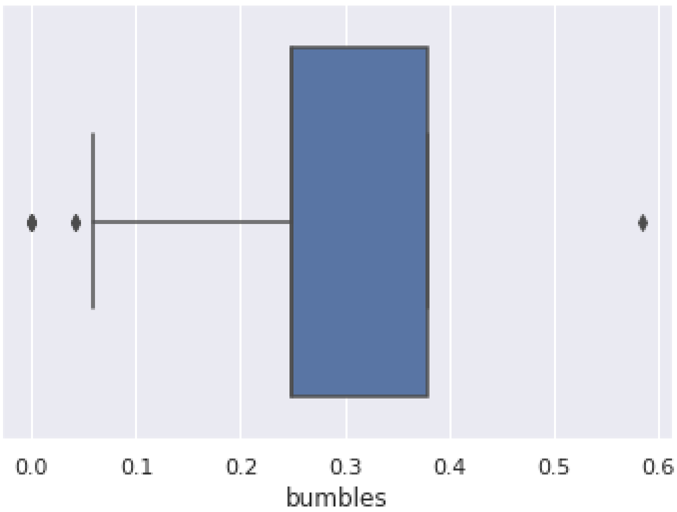
	Row#	clonesize	honeybee	bumbles	andrena	osmia	MaxOfUp
count	777.000000	777.000000	777.000000	777.000000	777.000000	777.000000	7
mean	388.000000	18.767696	0.417133	0.282389	0.468817	0.562062	
std	224.444871	6.999063	0.978904	0.066343	0.161052	0.169119	
min	0.000000	10.000000	0.000000	0.000000	0.000000	0.000000	
25%	194.000000	12.500000	0.250000	0.250000	0.380000	0.500000	
50%	388.000000	12.500000	0.250000	0.250000	0.500000	0.630000	
75%	582.000000	25.000000	0.500000	0.380000	0.630000	0.750000	
max	776.000000	40.000000	18.430000	0.585000	0.750000	0.750000	

```
df.hist(layout=(5,4), figsize=(20,15), bins=20)
plt.show()
```




```
sns.boxplot(df["bumbles"])
```

```
Pass the following variable as a keyword arg: x. From version 0.12, the only valid p
<matplotlib.axes._subplots.AxesSubplot at 0x7fb70519e690>
```



```
sns.boxplot(df["honeybee"])
```

```
Pass the following variable as a keyword arg: x. From version 0.12, the only valid p
<matplotlib.axes._subplots.AxesSubplot at 0x7fb7046fe690>
```



```
plt.figure(figsize=(20,20))
c = df.corr()
plt.figure(figsize=(15,12))
sns.heatmap(c, annot=True, cmap="YlGnBu")
plt.title('Understanding the Correlation between Input Data by a Heatmap', fontsize=15)
plt.show()
```

```
<Figure size 1440x1440 with 0 Axes>
```

Understanding the Correlation between Input Data by a Heatmap

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 777 entries, 0 to 776
Data columns (total 18 columns):
 #   Column                Non-Null Count  Dtype  
---  ------                --------------  -----  
 0   Row#                  777 non-null    int64  
 1   clonesize             777 non-null    float64
 2   honeybee              777 non-null    float64
 3   bumbles               777 non-null    float64
 4   andrena               777 non-null    float64
 5   osmia                 777 non-null    float64
 6   MaxOfUpperTRange      777 non-null    float64
 7   MinOfUpperTRange      777 non-null    float64
 8   AverageOfUpperTRange  777 non-null    float64
 9   MaxOfLowerTRange      777 non-null    float64
 10  MinOfLowerTRange      777 non-null    float64
 11  AverageOfLowerTRange  777 non-null    float64
 12  RainingDays           777 non-null    float64
 13  AverageRainingDays    777 non-null    float64
 14  fruitset              777 non-null    float64
 15  fruitmass             777 non-null    float64
 16  seeds                 777 non-null    float64
 17  yield                 777 non-null    float64
dtypes: float64(17), int64(1)
memory usage: 109.4 KB
```

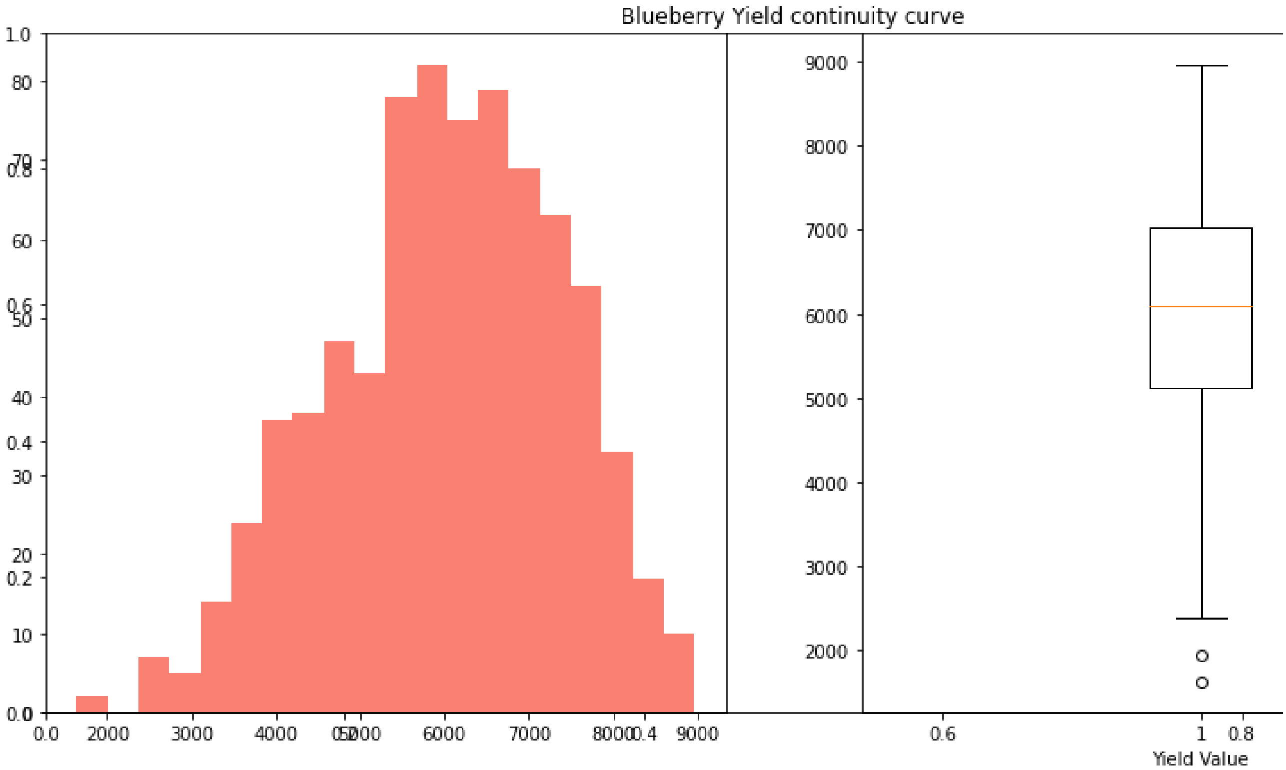
```
df.nunique()
```

```
Row#                    777
clonesize                 6
honeybee                  7
bumbles                  10
andrena                  12
osmia                    12
MaxOfUpperTRange          5
MinOfUpperTRange          5
AverageOfUpperTRange      5
MaxOfLowerTRange          5
MinOfLowerTRange          5
AverageOfLowerTRange      5
RainingDays               5
AverageRainingDays        5
fruitset                777
fruitmass               777
seeds                   777
yield                   777
dtype: int64
```

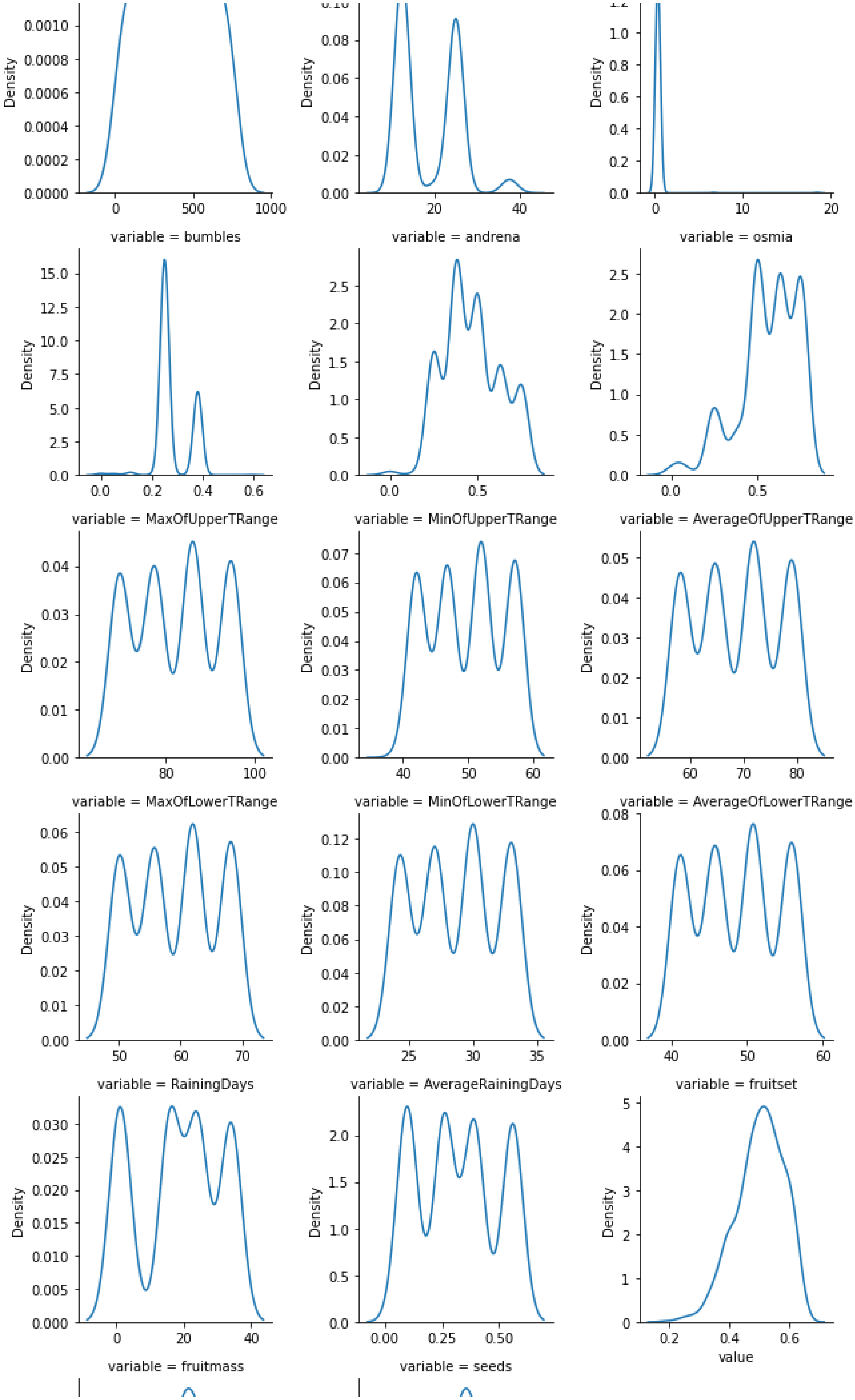
Analysis

Univariate Analysis

```
figs = plt.figure(figsize=(15,7))
plt.title("Blueberry Yield continuity curve")
ax1 = figs.add_subplot(121)
ax2 = figs.add_subplot(122)
x = df["yield"]
plt.xlabel("Yield Value")
ax1.hist(x, bins=20, color="salmon")
ax2.boxplot(x);
```

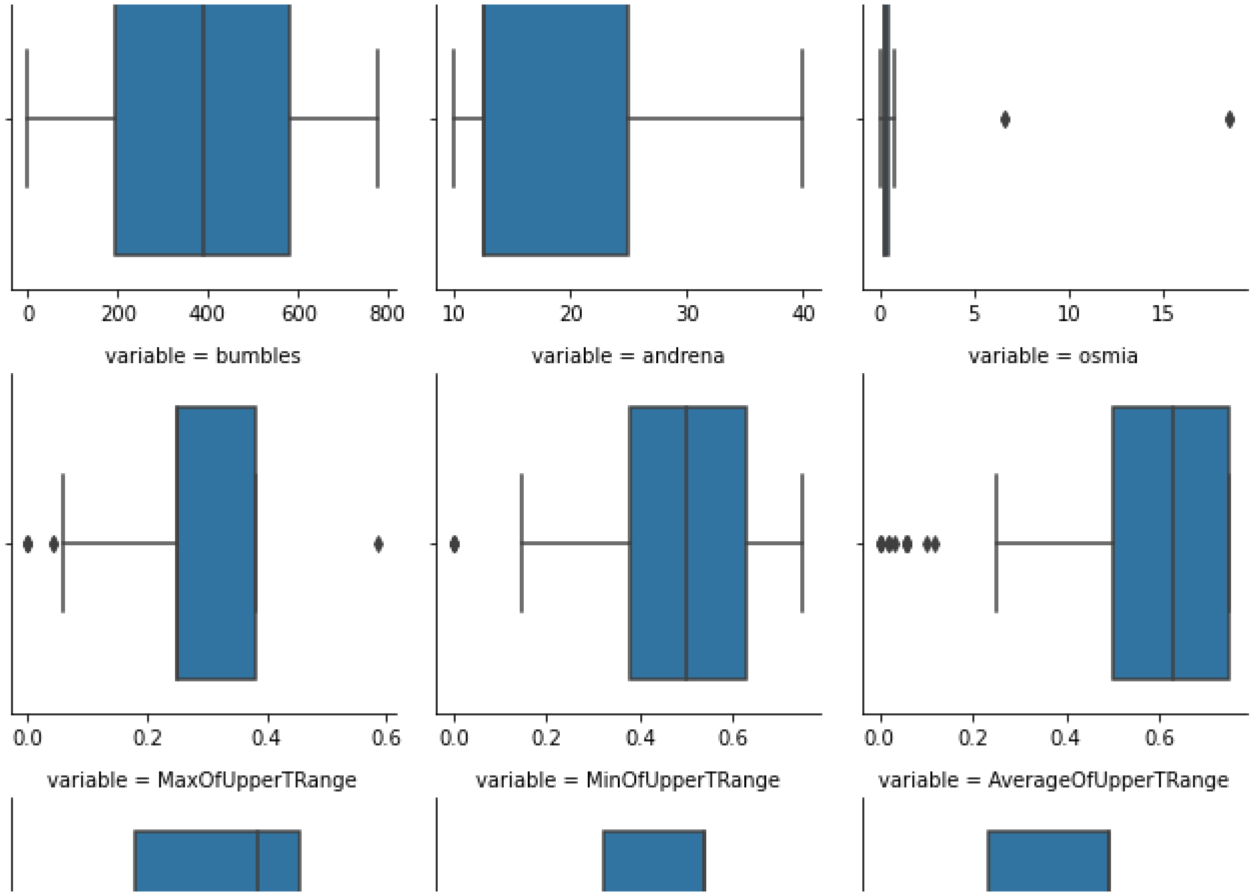



```
unpivot = pd.melt(df, df.describe().columns[-1], df.describe().columns[:-1])

g = sns.FacetGrid(unpivot, col="variable", col_wrap=3, sharex=False, sharey=False)
g.map(sns.kdeplot, "value")

plt.show()
```



variable = bumbles
variable = andrena
variable = osmia
variable = MaxOfUpperTRange
variable = MinOfUpperTRange
variable = AverageOfUpperTRange
variable = MaxOfLowerTRange
variable = MinOfLowerTRange
variable = AverageOfLowerTRange
variable = RainingDays
variable = AverageRainingDays
variable = fruitset
variable = fruitmass
variable = seeds
Density
value

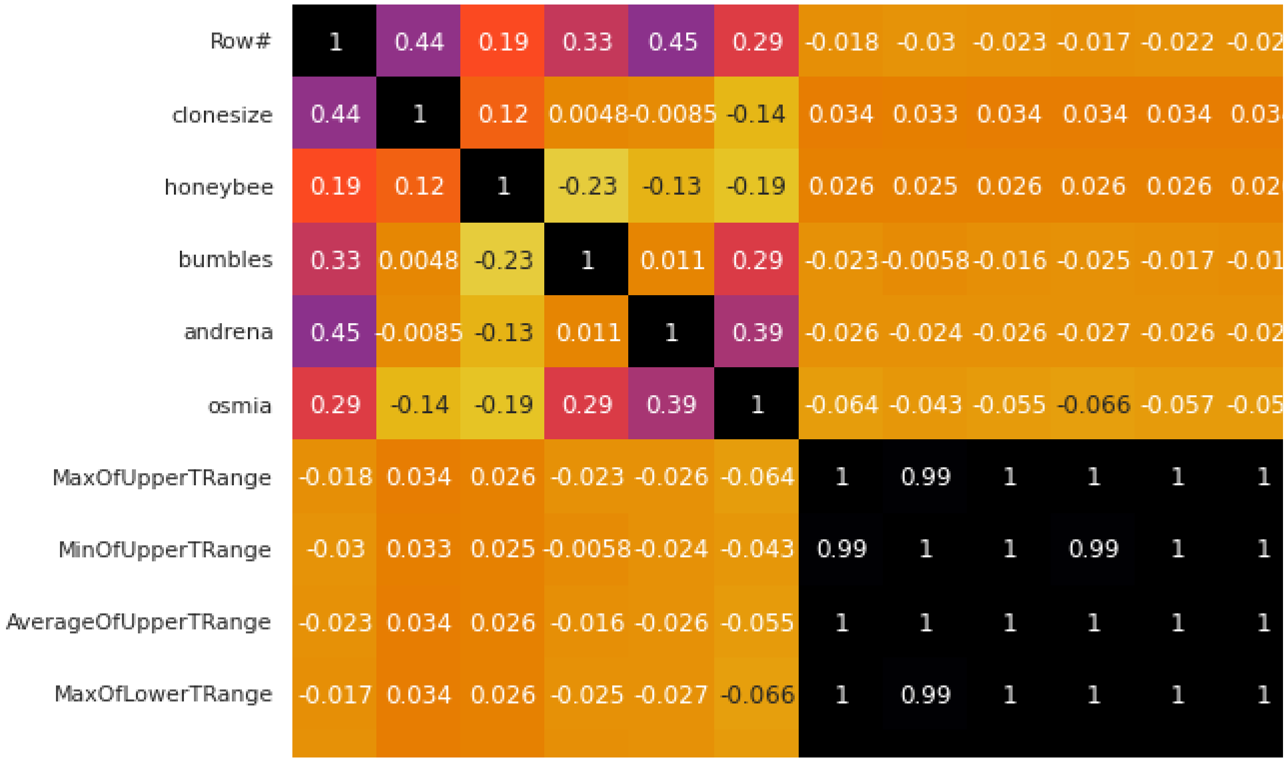

```
unpivot = pd.melt(df, df.describe().columns[-1], df.describe().columns[:-1])

g = sns.FacetGrid(unpivot, col="variable", col_wrap=3, sharex=False, sharey=False)
g.map(sns.boxplot, "value")

plt.show()
```



Multivariate Analysis

```
plt.figure(figsize=(17,12))
sns.set()
sns.heatmap(df.corr(), annot=True, cmap=plt.cm.CMRmap_r)
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7fb704af6890>
```



Preprocessing

columns to drop: 'Row#', 'MaxOfUpperTRange', 'MinOfUpperTRange', 'MaxOfLowerTRange', 'MinOfLowerTRange', 'RainingDays', 'honeybee'

```
df.columns
```

```
Index(['Row#', 'clonesize', 'honeybee', 'bumbles', 'andrena', 'osmia',
       'MaxOfUpperTRange', 'MinOfUpperTRange', 'AverageOfUpperTRange',
       'MaxOfLowerTRange', 'MinOfLowerTRange', 'AverageOfLowerTRange',
       'RainingDays', 'AverageRainingDays', 'fruitset', 'fruitmass', 'seeds',
       'yield'],
      dtype='object')
```

```
bbry_data_process = df.drop(columns=['Row#', 'MaxOfUpperTRange', 'MinOfUpperTRange', 'Max
bbry_data_process
```

	clonesize	bumbles	andrena	osmia	AverageOfUpperTRange	AverageOfLowerTRange
0	37.5	0.250	0.250	0.250	71.9	50.8
1	37.5	0.250	0.250	0.250	71.9	50.8
2	37.5	0.250	0.250	0.250	79.0	55.9
3	37.5	0.250	0.250	0.250	79.0	55.9
4	37.5	0.250	0.250	0.250	71.9	50.8
...	...	...	...	...	...	...

```
part1 = bbry_data_process.drop(columns=['yield'])
part2 = bbry_data_process[['yield']]
part1
```

	clonesize	bumbles	andrena	osmia	AverageOfUpperTRange	AverageOfLowerTRange
0	37.5	0.250	0.250	0.250	71.9	50.8
1	37.5	0.250	0.250	0.250	71.9	50.8
2	37.5	0.250	0.250	0.250	79.0	55.9
3	37.5	0.250	0.250	0.250	79.0	55.9
4	37.5	0.250	0.250	0.250	71.9	50.8
...	...	...	...	...	...	...
772	10.0	0.117	0.409	0.058	71.9	50.8
773	40.0	0.117	0.409	0.058	71.9	50.8
774	20.0	0.117	0.409	0.058	71.9	50.8
775	20.0	0.117	0.409	0.058	65.6	45.3
776	20.0	0.117	0.409	0.058	65.6	45.3

777 rows × 10 columns

```
Q1 = part1.quantile(0.25)
Q3 = part1.quantile(0.75)
IQR = Q3 - Q1
print(IQR)
```

```
clonesize               12.500000
bumbles                  0.130000
andrena                  0.250000
osmia                    0.250000
AverageOfUpperTRange     7.200000
AverageOfLowerTRange     5.000000
AverageRainingDays       0.290000
fruitset                 0.106571
fruitmass                0.059869
seeds                    6.123577
dtype: float64
```

```
bbry_data_iqr = bbry_data_process[~((bbry_data_process < (Q1 - 1.5 * IQR)) | (bbry_data_pr
bbry_data_iqr.shape
```

```
Automatic reindexing on DataFrame vs Series comparisons is deprecated and will raise
(752, 11)
```

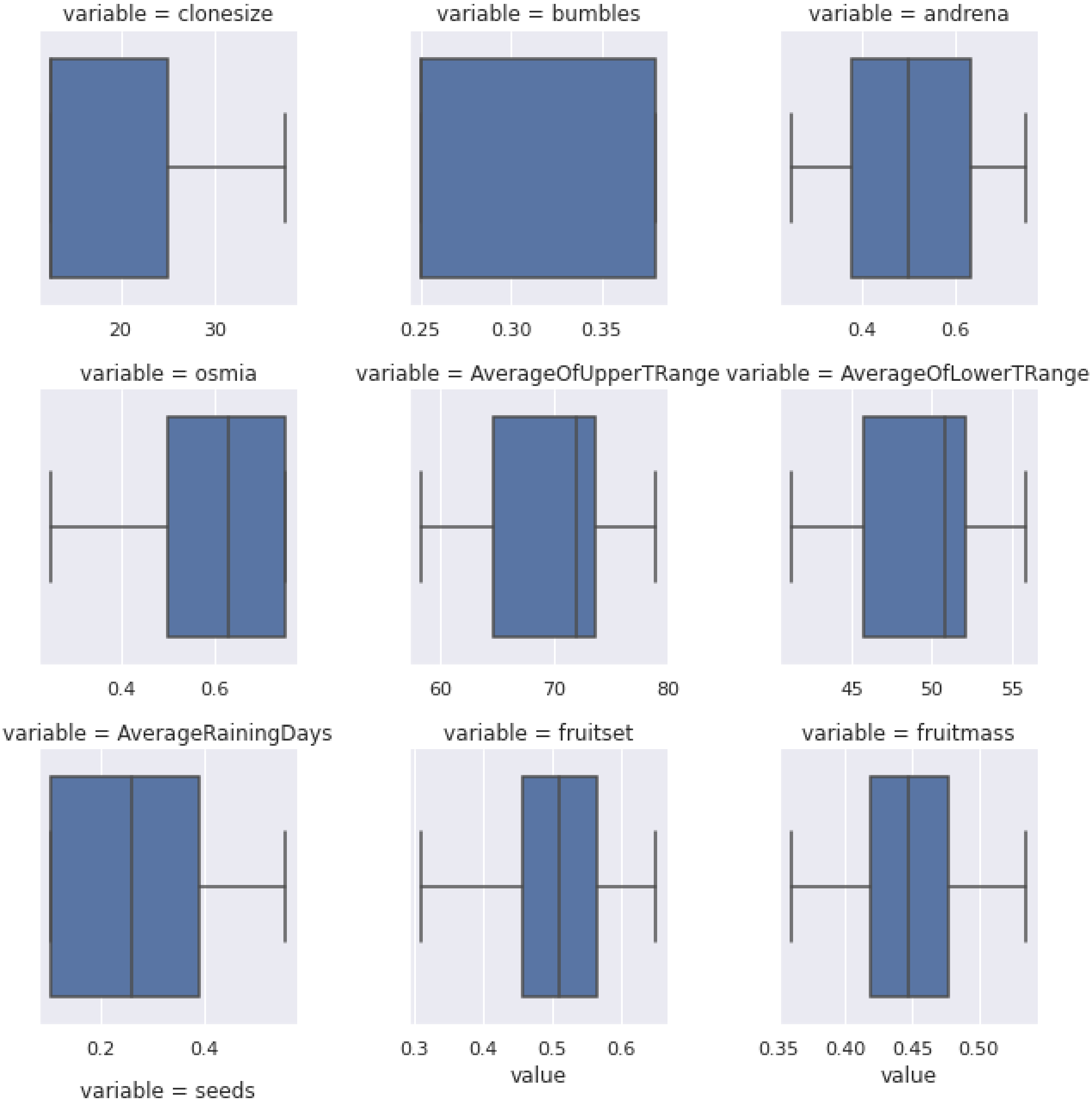
```
bbry_data_iqr
```

	clonesize	bumbles	andrena	osmia	AverageOfUpperTRange	AverageOfLowerTRange
0	37.5	0.25	0.25	0.25	71.9	50.8
1	37.5	0.25	0.25	0.25	71.9	50.8
2	37.5	0.25	0.25	0.25	79.0	55.9
3	37.5	0.25	0.25	0.25	79.0	55.9
4	37.5	0.25	0.25	0.25	71.9	50.8
...	...	...	...	...	...	...
754	25.0	0.38	0.63	0.50	64.7	45.8
755	25.0	0.38	0.63	0.50	58.2	41.2
756	25.0	0.38	0.63	0.50	58.2	41.2
757	25.0	0.38	0.63	0.50	64.7	45.8
758	25.0	0.38	0.63	0.50	64.7	45.8

752 rows × 11 columns

```
unpivot = pd.melt(bbry_data_iqr, bbry_data_iqr.describe().columns[-1], bbry_data_iqr.descr

g = sns.FacetGrid(unpivot, col="variable", col_wrap=3, sharex=False, sharey=False)
g.map(sns.boxplot, "value")

plt.show()
```

```
Using the boxplot function without specifying `order` is likely to produce an incorr
```




```
z = np.abs(stats.zscore(bbry_data_process))
print(z)
```

```
[[2.67812564 0.4885117  1.35954903 ... 0.93841323 1.01564827 1.62208748]
 [2.67812564 0.4885117  1.35954903 ... 0.50923815 0.61097218 0.7855304 ]
 [2.67812564 0.4885117  1.35954903 ... 1.16136881 1.27452236 1.58253738]
 ...
 [0.17618037 2.4945233  0.37165479 ... 1.0994045  1.22836665 1.33459611]
 [0.17618037 2.4945233  0.37165479 ... 1.14243699 1.26632705 1.36167994]
 [0.17618037 2.4945233  0.37165479 ... 1.33180188 1.45822756 1.52624822]]
```

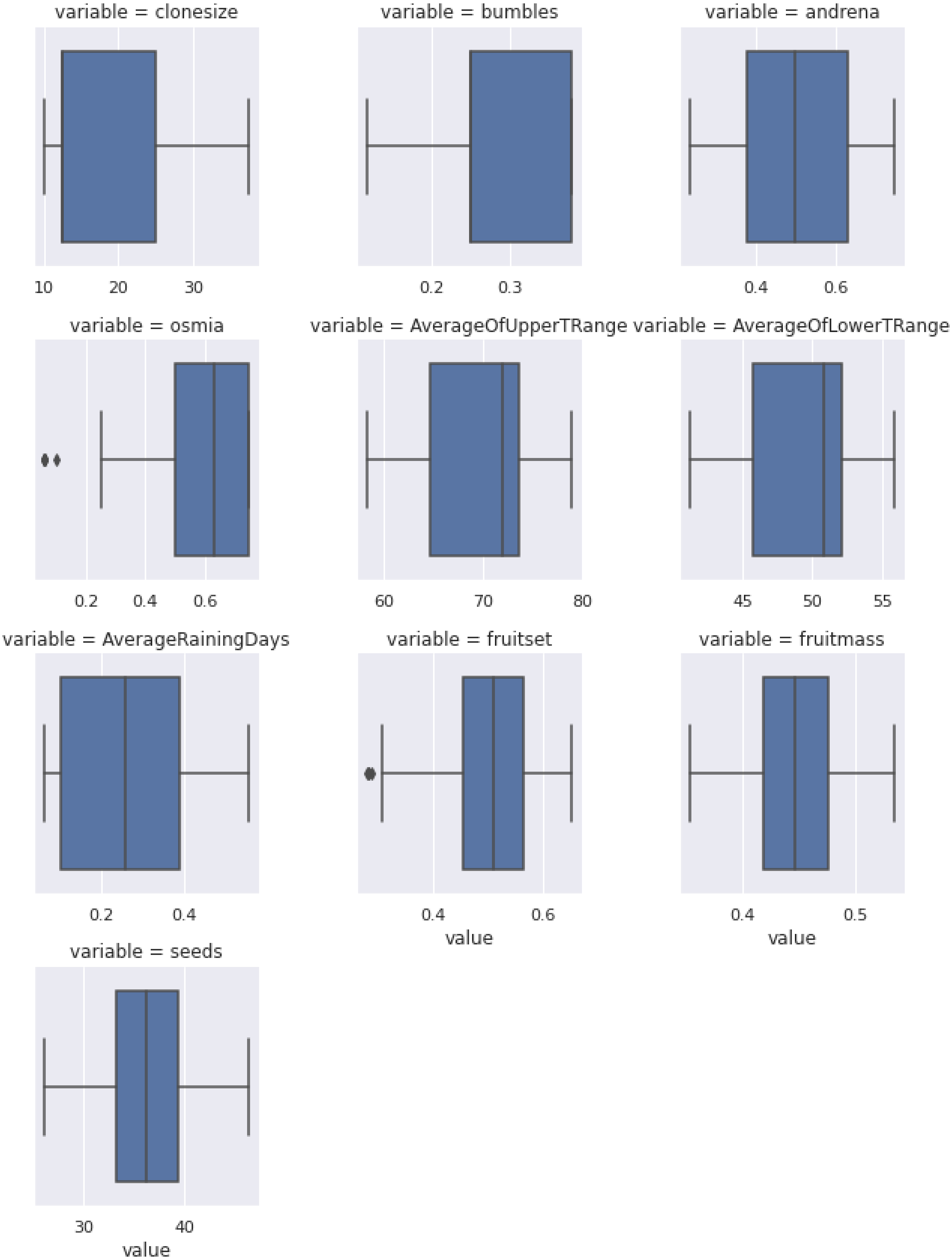
```
bbry_data_zscore = bbry_data_process[(z < 3).all(axis=1)]
bbry_data_zscore.shape
```

```
(764, 11)
```

```
unpivot = pd.melt(bbry_data_zscore, bbry_data_zscore.describe().columns[-1], bbry_data_zsc
```

```
g = sns.FacetGrid(unpivot, col="variable", col_wrap=3, sharex=False, sharey=False)
g.map(sns.boxplot, "value")

plt.show()
```

```
Using the boxplot function without specifying `order` is likely to produce an incorr
```




```
bbry_data_process = bbry_data_zscore
bbry_data_process
```

	clonesize	bumbles	andrena	osmia	AverageOfUpperTRange	AverageOfLowerTRange
0	37.5	0.250	0.250	0.250	71.9	50.8
1	37.5	0.250	0.250	0.250	71.9	50.8
2	37.5	0.250	0.250	0.250	79.0	55.9
3	37.5	0.250	0.250	0.250	79.0	55.9
4	37.5	0.250	0.250	0.250	71.9	50.8
...	...	...	...	...	...	...
770	20.0	0.293	0.234	0.058	71.9	50.8
772	10.0	0.117	0.409	0.058	71.9	50.8
774	20.0	0.117	0.409	0.058	71.9	50.8
775	20.0	0.117	0.409	0.058	65.6	45.3
776	20.0	0.117	0.409	0.058	65.6	45.3

764 rows × 11 columns

Feature Selection

Creating 2 splits on Dataset, and each will be analysed on the importance of either **Mutual Information gain** or **Correlation Regression** values

```
def select_features_corr_based(X_train, y_train, X_test, x="all"):
    if type(x) == str:
        fs_corr = SelectKBest(score_func=f_regression, k='all')
    else:
        fs_corr = SelectKBest(score_func=f_regression, k = x)
    fs_corr.fit(X_train, y_train)
    X_train_fs = fs_corr.transform(X_train)
    X_test_fs = fs_corr.transform(X_test)

    return X_train_fs, X_test_fs, fs_corr

def select_features_infogain_based(X_train, y_train, X_test, x="all"):
    if type(x) == str:
        fs_info = SelectKBest(score_func=mutual_info_regression, k='all')
    else:
        fs_info = SelectKBest(score_func=mutual_info_regression, k=x)
    fs_info.fit(X_train, y_train)
    X_train_fs = fs_info.transform(X_train)
    X_test_fs = fs_info.transform(X_test)

    return X_train_fs, X_test_fs, fs_info
```

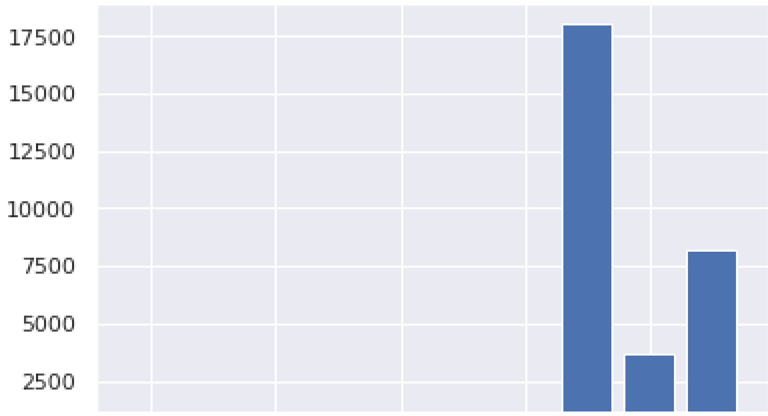
To perform many folds of tuning on the decided dataset. hence I would have to choose either one option of the below

info-gain vs correlation

The **KSelection score plot** describes a good behavior of the **Information Gain Values**. Hence, all the dataset will be based on the same

```
X = bbry_data_process.drop(["yield"], axis=1)
y = bbry_data_process['yield']

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33, random_state=1)

X_train_fs_corr, X_test_fs_corr, fs_corr = select_features_corr_based(X_train, y_train, X_
X_train_fs_info, X_test_fs_info, fs_info = select_features_infogain_based(X_train, y_trair


def fs_score_plot(fs_func):

    for i in range(len(fs_func.scores_)):
        print('Feature %d: %f' % (i, fs_func.scores_[i]))
    # plot the scores
    plt.bar([i for i in range(len(fs_func.scores_))], fs_func.scores_)
    plt.show()

fs_score_plot(fs_corr)
fs_score_plot(fs_info)
```

```
Feature 0: 175.463722
Feature 1: 35.414625
Feature 2: 8.092982
Feature 3: 95.866044
Feature 4: 23.685723
Feature 5: 23.512387
Feature 6: 265.725911
Feature 7: 18027.130769
Feature 8: 3685.065541
Feature 9: 8199.278441
```



Modelling

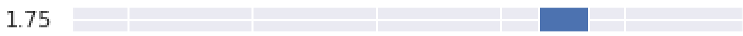
Training the model on top 9 features, using both the splits of dataset,and check the metrics of the same on 4 Models

Linear Regression

Random Forest

Decision Tree

XGBoost




```
X = bbry_data_process.drop(["yield"], axis=1)
y = bbry_data_process['yield']

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33, random_state=1)

X_train_fs, X_test_fs, fs_info = select_features_infogain_based(X_train, y_train, X_test,
```

LinearRegression

```
lreg = LinearRegression()

lreg.fit(X_train_fs, y_train)

yhat = lreg.predict(X_test_fs)


mae_linear = mean_absolute_error(y_test, yhat)
```

```
mse_linear = mean_squared_error(y_test, yhat)
rmse_linear = np.sqrt(mse_linear)
rsq_linear = r2_score(y_test, yhat)

print('MAE: %.3f' % mae_linear)
print('MSE: %.3f' % mse_linear)
print('RMSE: %.3f' % rmse_linear)
print('R-Square: %.3f' % rsq_linear)
```

```
MAE: 103.080
MSE: 19166.573
RMSE: 138.443
R-Square: 0.988
```

RandomForest Regression

```
rf = RandomForestRegressor()

rf.fit(X_train_fs, y_train)

yhat = rf.predict(X_test_fs)


mae_rf = mean_absolute_error(y_test, yhat)
mse_rf = mean_squared_error(y_test, yhat)
rmse_rf = np.sqrt(mse_rf)
rsq_rf = r2_score(y_test, yhat)

print('MAE: %.3f' % mae_rf)
print('MSE: %.3f' % mse_rf)
print('RMSE: %.3f' % rmse_rf)
print('R-Square: %.3f' % rsq_rf)
```

```
MAE: 118.298
MSE: 24546.444
RMSE: 156.673
R-Square: 0.984
```

Decision Tree Regression

```
dtree = DecisionTreeRegressor()

dtree.fit(X_train_fs, y_train)

yhat = dtree.predict(X_test_fs)


mae_dt = mean_absolute_error(y_test, yhat)
mse_dt = mean_squared_error(y_test, yhat)
rmse_dt = np.sqrt(mse_dt)
rsq_dt = r2_score(y_test, yhat)
```

```
print('MAE: %.3f' % mae_dt)
print('MSE: %.3f' % mse_dt)
print('RMSE: %.3f' % rmse_dt)
print('R-Square: %.3f' % rsq_dt)
```

```
MAE: 173.064
MSE: 55319.084
RMSE: 235.200
R-Square: 0.965
```

XGBRegression

```
xgb = XGBRegressor()

xgb.fit(X_train_fs, y_train)

yhat = xgb.predict(X_test_fs)
```

```
[08:43:48] WARNING: /workspace/src/objective/regression_obj.cu:152: reg:linear is no
```

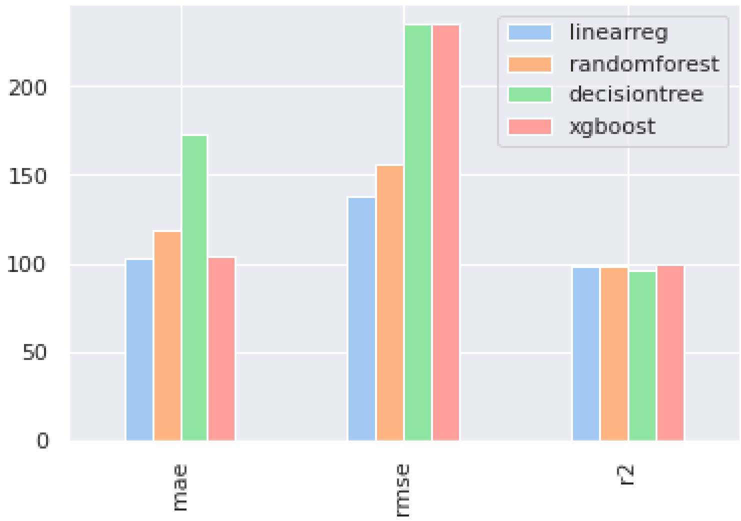
```
mae_xgb = mean_absolute_error(y_test, yhat)
mse_xgb = mean_squared_error(y_test, yhat)
rmse_xgb = np.sqrt(mse_dt)
rsq_xgb = r2_score(y_test, yhat)

print('MAE: %.3f' % mae_xgb)
print('MSE: %.3f' % mse_xgb)
print('RMSE: %.3f' % rmse_xgb)
print('R-Square: %.3f' % rsq_xgb)
```

```
MAE: 103.536
MSE: 17426.014
RMSE: 235.200
R-Square: 0.989
```

Model Evaluation

```
error_rec = {
    "linearreg": {
        "mae": mae_linear,
        "rmse": rmse_linear,
        'r2': rsq_linear*100
    },
    "randomforest": {
        "mae": mae_rf,
        "rmse": rmse_rf,
        'r2': rsq_rf*100
    },
```

```
    "decisiontree": {
        "mae": mae_dt,
        "rmse": rmse_dt,
        'r2': rsq_dt*100
    },
    "xgboost": {
        "mae": mae_xgb,
        "rmse": rmse_xgb,
        'r2': rsq_xgb*100
    },
}
pd.DataFrame(error_rec).plot(kind="bar",
             color=[
                 sns.color_palette("pastel")[0],
                 sns.color_palette("pastel")[1],
                 sns.color_palette("pastel")[2],
                 sns.color_palette("pastel")[3]]);
```



The comparative BarPlot shows the values of each. We have **Linear Regression** (138.443) and **XGBoost** (155.946) at the lowest Error Rate

Hyperparameter Tuning

crossvalidation, RepeatedKFold and GridSearchCV are the popular methods for Parameter Tuning

We have only tuned the model for K values, plus an addition model parameter. With greater processor, the tuning can afford to produce folds 3 times over 4 parameter values

Another alternative is: **TuneSearchCV**

```
cv = RepeatedKFold(n_splits= 50, n_repeats = 3, random_state = 1)
fs_info_v0 = SelectKBest(score_func = mutual_info_regression)


# define pipeline for each algorithm
# define GSCV for each
```

```
# loop through it


pipe_lr = Pipeline([
    ('sel', fs_info_v0),
    ('model', LinearRegression())
])

pipe_rf = Pipeline([
    ('sel', fs_info_v0),
    ('model', RandomForestRegressor(random_state=1))
])

pipe_dtree = Pipeline([
    ('sel', fs_info_v0),
    ('model', DecisionTreeRegressor(random_state=1))
])

pipe_xgb = Pipeline([
    ('sel', fs_info_v0),
    ('model', XGBRegressor(random_state=1))
])

# pipe_lr.get_params().keys()


param_range = [15, 18, 10]
param_range_fl = [5.0, 10.0]

grid_params_lr = [{'sel__k': [i for i in range(X_train_fs.shape[1]-6, X_train_fs.shape[1]-
        }]



grid_params_rf = [{'sel__k': [i for i in range(X_train_fs.shape[1]-6, X_train_fs.shape[1]-
        'model__criterion': ['mse', 'mae'],
#         'model__max_depth': param_range,
#         'model__min_samples_split': param_range[1:]
                  }]



grid_params_dtree = [{'sel__k': [i for i in range(X_train_fs.shape[1]-6, X_train_fs.shape[
                    'model__criterion': ['mse', 'mae'],
#                      'model__max_depth': param_range,
#                      'model__max_features': ['auto', 'sqrt']
                     }]

grid_params_xgb = [{'sel__k': [i for i in range(X_train_fs.shape[1]-6, X_train_fs.shape[1]
#                     'model__max_depth': [9,12],
#                    'model__min_child_weight': [7,8],
                   'model__subsample': [i/10. for i in range(9,11)]
                   }]



LR = GridSearchCV(estimator=pipe_lr,
```

```
            param_grid=grid_params_lr,
            scoring='neg_mean_absolute_error',
            cv=cv)

RF = GridSearchCV(estimator=pipe_rf,
            param_grid=grid_params_rf,
            scoring='neg_mean_absolute_error',
            cv=cv,
            n_jobs= -1)

DT = GridSearchCV(estimator=pipe_dtree,
            param_grid=grid_params_dtree,
            scoring='neg_mean_absolute_error',
            cv=cv,
            n_jobs= -1)

XGB = GridSearchCV(estimator=pipe_xgb,
            param_grid=grid_params_xgb,
            scoring='neg_mean_absolute_error',
            cv=cv,
            n_jobs= -1)

grids = [LR,RF,XGB,DT]

# Creating a dict for our reference
grid_dict = {0: 'Linear Regression',
        1: 'Random Forest',
        2: 'XGBoost',
        3: 'Decision Tree'}



# Start form initial scaled model: X_train17 and X_test17, y_train17 and y_test17
def extract_best_model(grids: list, grid_dict: dict):
    print('Performing model optimizations...')
    least_mae = 270817
    best_regr = 0
    best_gs = ''
    for idx, gs in enumerate(grids):
        print('\nEstimator: %s' % grid_dict[idx])
        gs.fit(X_train_fs, y_train)
        print('Best Config: %s' % gs.best_params_)
        # Best training data accuracy
        print('Best MAE: %.3f' % gs.best_score_)
        # Predict on test data with best params
        y_pred_v0 = gs.predict(X_test_fs)
        # Test data accuracy of model with best params
        print('Test set mean absolute error for best params: %.3f ' % mean_absolute_error(
        print('Test set root mean squared error for best params: %.3f ' % np.sqrt(mean_abs

        # Track best (least test error) model
        if mean_absolute_error(y_test, y_pred_v0) < least_mae:
            least_mae = mean_absolute_error(y_test, y_pred_v0)
            best_gs = gs
            best_regr = idx
```

```
    print('\nClassifier with least test set MAE: %s' % grid_dict[best_regr])


    ########### summarize all values of parameters (uncomment only if nescessary)
    ######### means = results.cv_results_['mean_test_score']
    ######## params = results.cv_results_['params']
    ####### for mean, param in zip(means, params):
    ###### print(">%.3f with: %r" % (mean, param))

    return (grid_dict[best_regr], best_gs, least_mae)
```

Prediction and Evaluation

Running the GridSearchCV and saving the best model.

```
best_model_name_v0, best_model_v0, least_mae_v0 = extract_best_model(grids= grids, grid_di

print(f"Best Model: {best_model_name_v0}")
print(f"Error Rate: {least_mae_v0}")
print(best_model_v0)
```

```
Performing model optimizations...

Estimator: Linear Regression
Best Config: {'sel__k': 4}
Best MAE: -116.867
Test set mean absolute error for best params: 127.375
Test set root mean squared error for best params: 11.286

Estimator: Random Forest
Criterion 'mse' was deprecated in v1.0 and will be removed in version 1.2. Use `crit
Best Config: {'model__criterion': 'mse', 'sel__k': 4}
Best MAE: -130.967
Test set mean absolute error for best params: 130.067
Test set root mean squared error for best params: 11.405

Estimator: XGBoost
[08:52:01] WARNING: /workspace/src/objective/regression_obj.cu:152: reg:linear is no
Best Config: {'model__subsample': 1.0, 'sel__k': 4}
Best MAE: -135.114
Test set mean absolute error for best params: 140.048
Test set root mean squared error for best params: 11.834

Estimator: Decision Tree
Best Config: {'model__criterion': 'mse', 'sel__k': 4}
Best MAE: -175.420
Test set mean absolute error for best params: 177.971
Test set root mean squared error for best params: 13.341

Classifier with least test set MAE: Linear Regression
Best Model: Linear Regression
Error Rate: 127.3749731578002
GridSearchCV(cv=RepeatedKFold(n_repeats=3, n_splits=50, random_state=1),
             estimator=Pipeline(steps=[('sel',
```

```
                                        SelectKBest(score_func=<function mutual_info
                                       ('model', LinearRegression())]),
                 param_grid=[{'sel__k': [3, 4]}],
                 scoring='neg_mean_absolute_error')
Criterion 'mse' was deprecated in v1.0 and will be removed in version 1.2. Use `crit
```

Although Linear Regression is the best of the listed models, **RandomForest** would potentially produce closer precise results, due to good learning rate, hence I would retune it in addition to the best params extracted in the first search

```
grid_params_rf1 = [{
        'model__max_depth': param_range,
        'model__min_samples_split': [2,5]
                  }]

RF1·=·GridSearchCV(
····estimator·=·Pipeline([
····················('sel',·SelectKBest(score_func=mutual_info_regression,·k=8)),·
····················('model',·RandomForestRegressor(random_state=1,·criterion='mse'))
················]),
            param_grid=grid_params_rf1,
            scoring='neg_mean_absolute_error',
            cv=cv,
            n_jobs= -1)


print("Random Forest V-1 optimising...")
RF1.fit(X_train_fs, y_train)
print('Best Config: %s' % RF1.best_params_)
print('Best MAE: %.3f' % RF1.best_score_)
y_pred_v1_rf1 = RF1.predict(X_test_fs)
print('Test set mean absolute error for best params: %.3f ' % mean_absolute_error(y_test,
print('Test set root mean squared error for best params: %.3f ' % np.sqrt(mean_absolute_er
```

```
Random Forest V-1 optimising...
Criterion 'mse' was deprecated in v1.0 and will be removed in version 1.2. Use `crit
Best Config: {'model__max_depth': 10, 'model__min_samples_split': 2}
Best MAE: -120.603
Test set mean absolute error for best params: 116.359
Test set root mean squared error for best params: 10.787
```