1 Lab Notebook 5-6

In this notebook, we will apply the kNN and DT algorithm to a larger exoplanet dataset. We will then examine the effectiveness of the kNN model via a variety of metrics and diagnostics.

2 Part 1: Explore data

Start by importing the same modules as in labs 3 and 4, set the matplotlib.rc for graphing, and read in **phl_exoplanet_catalog.csv** using pandas. Additionally, with pandas, use set_option to make sure that the maximum number of columns and rows displayed for our dataframes is 100. Also using set_option, set max_colwidth=100, which modifies the default column width.

In [2]:
```
import pandas as pd
```

Find a way to print only the column names of the dataframe. These are the data's features.

In [3]:
```
df = pd.read_csv("./phl_exoplanet_catalog.csv")
```

We want to start familiarizing ourselves with our data. Use "describe()" on the dataframe to see some summary statistics. Note down what each of these statistics mean.

In [4]:
```
df.describe()
```

Out[4]:

	P_STATUS	P_MASS	P_MASS_ERROR_MIN	P_MASS_ERROR_MAX	P_RADIUS	P_RADIUS_ERROR_MIN	P_RADIUS_ERROR_MAX	
count	4048.0	1598.000000	1467.000000	1467.000000	3139.000000	3105.000000	3105.000000	404
mean	3.0	798.384920	-152.292232	190.289692	4.191426	-0.483990	0.621867	201
std	0.0	1406.808654	783.366353	1082.061976	4.776830	1.409048	2.007592	:
min	3.0	0.019070	-24965.390000	0.000000	0.336300	-54.592700	0.000000	198
25%	3.0	26.548968	-79.457001	4.449592	1.569400	-0.526870	0.145730	201
50%	3.0	273.332080	-24.154928	25.108412	2.331680	-0.235410	0.325090	201
75%	3.0	806.488560	-4.392383	85.813561	3.553570	-0.134520	0.661390	201
max	3.0	17668.059000	0.270000	26630.808000	77.349000	0.450000	68.919080	201

8 rows × 98 columns

We can group statistics by class. For each possible value of "P_HABITABLE" (0 = not habitable, 1 = possibly habitable, or 2 = probably habitable), display the count for each of the features.

In [5]:
```
grouped_data = df.groupby("P_HABITABLE")
grouped_data.describe()
```

Out[5]:

	P_STATUS								P_MASS		...	P_MASS_EST					
	count	**mean**	**std**	**min**	**25%**	**50%**	**75%**	**max**	**count**	**mean**	**...**	**75%**	**max**	**count**	**mean**	**std**	
P_HABITABLE																	
0	3993.0	3.0	0.0	3.0	3.0	3.0	3.0	3.0	1575.0	809.993111	...	156.689210	17668.059000	3923.0	4.063305	62.824346	0.0
1	21.0	3.0	0.0	3.0	3.0	3.0	3.0	3.0	16.0	1.941373	...	2.545899	3.931532	21.0	0.169579	0.197510	0.0
2	34.0	3.0	0.0	3.0	3.0	3.0	3.0	3.0	7.0	6.984497	...	6.584072	8.921432	34.0	0.393622	0.280729	0.0

3 rows × 776 columns

3 Part 2: Modify data

3.1 Step 2.1

We want a binary classification problem, so lump together "probably" (P_habitable=2) and "possibly" (P_habitable=1) habitable planets in a new dataframe. Check that the two classes are lumped together correctly.

In [6]:
```
from copy import copy

lumped_df = copy(df)
lumped_df['P_HABITABLE'] = lumped_df['P_HABITABLE'].replace(2, 1)

# Check that the two classes are lumped together correctly
lumped_df['P_HABITABLE'].value_counts()
```

Out[6]:
```
P_HABITABLE
0    3993
1      55
Name: count, dtype: int64
```

In [7]:
```
df['P_HABITABLE'].value_counts()
```

Out[7]:
```
P_HABITABLE
0    3993
2      34
1      21
Name: count, dtype: int64
```

3.2 Step 2.2

Let's simplify our data by only using the features 'S_MASS', 'P_PERIOD', and 'P_DISTANCE'. From the dataset created in step 2.1 create a new dataframe called "final_features" that is comprised of the columns 'S_MASS', 'P_PERIOD', and 'P_DISTANCE'. Display the first few rows.

In [8]:
```
final_features = lumped_df[['S_MASS','P_PERIOD','P_DISTANCE']]
```

Each column of a data frame is called a *series*. Create a new series named "targets" that is the column "P_HABITABLE". Display the first few rows.

In [9]:
```
targets = lumped_df['P_HABITABLE']
```

3.3 Step 2.3

We need to delete data points that contain missing (NaN) values. We can see that NaN values exist by comparing the shape of "final_features" with the count of non-NaN values (using "describe"). Complete these two steps below:

In [10]:
```
final_features.describe()
```

Out[10]:

	S_MASS	P_PERIOD	P_DISTANCE
count	3283.000000	3.938000e+03	3978.000000
mean	1.003838	2.309342e+03	4.047677
std	0.652903	1.167012e+05	62.435994
min	0.010000	9.070629e-02	0.004408
25%	0.810000	4.497336e+00	0.053110
50%	0.970000	1.187053e+01	0.103000
75%	1.130000	4.186661e+01	0.263415
max	23.560000	7.300000e+06	2500.000000

Count the number of NaN values in each column of "final_features" (hint: use isnull()). how many are there in each column?

In [11]:
```
final_features.isnull().sum()
```

Out[11]:
```
S_MASS        765
P_PERIOD      110
P_DISTANCE     70
dtype: int64
```

Remove rows that have one or more NaN values (hint: use "dropna"):

In [12]:
```
final_features = final_features.dropna(axis=0)
final_features.isnull().values.any()
final_features.info()
```

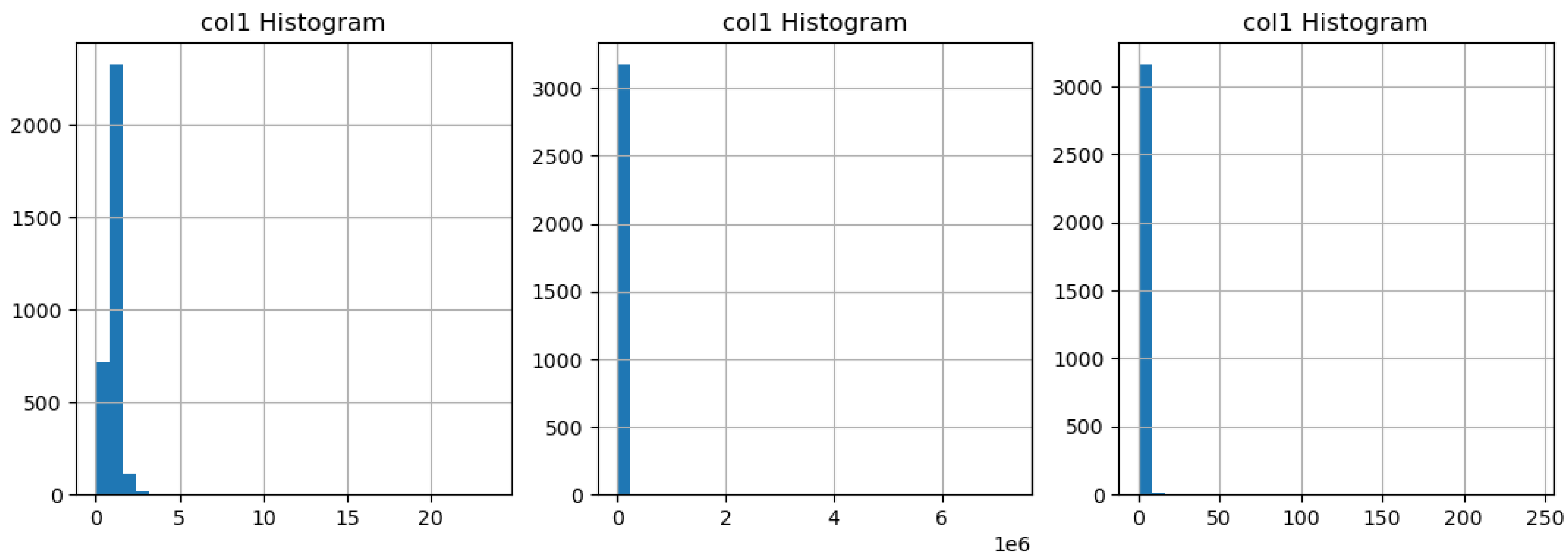
```
<class 'pandas.core.frame.DataFrame'>
Index: 3180 entries, 0 to 4047
Data columns (total 3 columns):
 #   Column      Non-Null Count  Dtype  
---  ------      --------------  -----  
 0   S_MASS      3180 non-null   float64
 1   P_PERIOD    3180 non-null   float64
 2   P_DISTANCE  3180 non-null   float64
dtypes: float64(3)
memory usage: 99.4 KB
```

3.4 Step 2.4

We will now search for outliers in the data and remove them. One quick way to check if there are any is to inspect the distribution of values in each column by creating a histogram. Do this for all three columns:

In [13]:
```
from matplotlib import pyplot as plt

plt.figure(figsize=(13,4))

plt.subplot(131)
final_features['S_MASS'].hist(bins=30)
plt.title('col1 Histogram')

plt.subplot(132)
final_features['P_PERIOD'].hist(bins=30)
plt.title('col1 Histogram')

plt.subplot(133)
final_features['P_DISTANCE'].hist(bins=30)
plt.title('col1 Histogram')
```

Out[13]: `Text(0.5, 1.0, 'col1 Histogram')`



Due to the wide range of the x-axis (without specifying the range), we can infer that there are outliers.

We can also tell that there are outliers when we look at the difference between the mean and median for each of the features. Do this below using "describe()".

In [14]:
```
final_features.describe()
```

Out[14]:

	S_MASS	P_PERIOD	P_DISTANCE
count	3180.000000	3.180000e+03	3180.000000
mean	1.018217	2.763531e+03	0.677663
std	0.649450	1.298246e+05	5.962161
min	0.020000	9.070629e-02	0.004408
25%	0.820000	4.175797e+00	0.050453
50%	0.970000	1.155546e+01	0.097369
75%	1.130000	5.474041e+01	0.274581
max	23.560000	7.300000e+06	243.000000

Time to remove the outliers.

With scipy.stats.zscore, you can compute the z-score of "final_features". The z-score is the distance between the observed data point and the population mean, scaled by the standard deviation. Keep all data with absolute value of the z-score less than 5 in the array final_features, and remove any data that does not fit this criteria.

Make sure that the data kept and removed in "targets" reflects this change.

In [15]:

```
import numpy as np
from scipy import stats
threshold = 5
z_scores = stats.zscore(final_features)
bool_table = np.abs(z_scores < threshold)
outlier_mask = ~final_features.apply(stats.zscore).abs().gt(threshold).any(axis=1)
final_features_below_threshold = final_features[outlier_mask]
final_features_below_threshold.head(n=10)
```

Out[15]:

	S_MASS	P_PERIOD	P_DISTANCE
0	2.70	326.03000	1.324418
1	2.78	516.21997	1.534896
2	2.20	185.84000	0.830000
3	0.90	1773.40000	3.130558
4	1.08	798.50000	2.043792
5	2.30	993.30000	2.608320
7	0.99	30.35060	0.190168
8	1.54	452.80000	1.338399
9	1.54	883.00000	2.167464
14	0.48	416.00000	0.920000

As you should see above in "final_features" (and "targets"), the label for each row is not the true row number. In other words, the row label doesn't increase as 0,1,2,3,...

Reset the index of the data frame using "reset_index(drop=True)". This resets the index of the DataFrame, and inserts index into the dataframe columns.

Do the same for "targets".

In [16]:

```
filtered_targets=targets[final_features_below_threshold.index]
final_features_below_threshold.reset_index(drop=True)
```

Out[16]:

	S_MASS	P_PERIOD	P_DISTANCE
0	2.70	326.030000	1.324418
1	2.78	516.219970	1.534896
2	2.20	185.840000	0.830000
3	0.90	1773.400000	3.130558
4	1.08	798.500000	2.043792
...	...	...	...
3166	0.41	28.165600	0.134560
3167	0.41	7.906961	0.057690
3168	0.12	3.204000	0.021000
3169	0.12	6.689000	0.035000
3170	0.12	13.031000	0.054000

3171 rows × 3 columns

4 Part 3: Explore data (again)

4.1 Step 3.1

We want to check the balance of the data set. Meaning, in "targets", how many ones versus zeros are there?

Print the sum of "targets" (sum of all the ones) divided by the length of "targets".

In [17]:

```
sum(filtered_targets)/len(filtered_targets)
```

Out[17]: 0.01639861242510249

Also, try using "bincount" on "targets" to show the distribution of ones and zeros:

In [18]:
```python
from numpy import bincount

bincount(filtered_targets)
```

Out[18]:
```
array([3119,   52], dtype=int64)
```

Now we know that the data set is very imbalanced (many more zeros than ones). This means that we need to be careful when constructing our machine learning model; briefly explain why this is the case.

4.2 Answer:

If the dataset is heavily imbalanced then we can expect that certain metrics such as accuracy will be wildly inaccurate. Because it is possible for the model to suggest the planet is always inhabitable and in this case that would result in an accuracy of > 0.9

4.3 Step 3.2

Concatenate "final_features" and "targets" without outliers:

In [19]:
```python
filtered_df = final_features_below_threshold.join(filtered_targets)
```

Group the data by "P_HABITABLE", display make one row for P_HABITABLE=0 and another row for P_HABITABLE=1 and use the .describe() method to display summary statistics.

In [20]:
```python
grouped_data = df.groupby('P_HABITABLE').describe()

print(grouped_data)
```

```
            P_STATUS                                    P_MASS              \
               count mean  std  min  25%  50%  75%  max  count        mean
P_HABITABLE
0             3993.0  3.0  0.0  3.0  3.0  3.0  3.0  3.0 1575.0  809.993111
1               21.0  3.0  0.0  3.0  3.0  3.0  3.0  3.0   16.0    1.941373
2               34.0  3.0  0.0  3.0  3.0  3.0  3.0  3.0    7.0    6.984497

             ...  P_MASS_EST               P_SEMI_MAJOR_AXIS_EST           \
             ...         75%           max                 count      mean
P_HABITABLE  ...
0            ...  156.689210  17668.059000                3923.0  4.063305
1            ...    2.545899      3.931532                  21.0  0.169579
2            ...    6.584072      8.921432                  34.0  0.393622


                   std      min       25%       50%       75%          max
P_HABITABLE
0            62.824346  0.00440  0.052847  0.101407  0.259030  2500.000000
1             0.197510  0.02144  0.037100  0.089000  0.213000     0.718000
2             0.280729  0.09100  0.175235  0.258808  0.589277     1.190229

[3 rows x 776 columns]
```

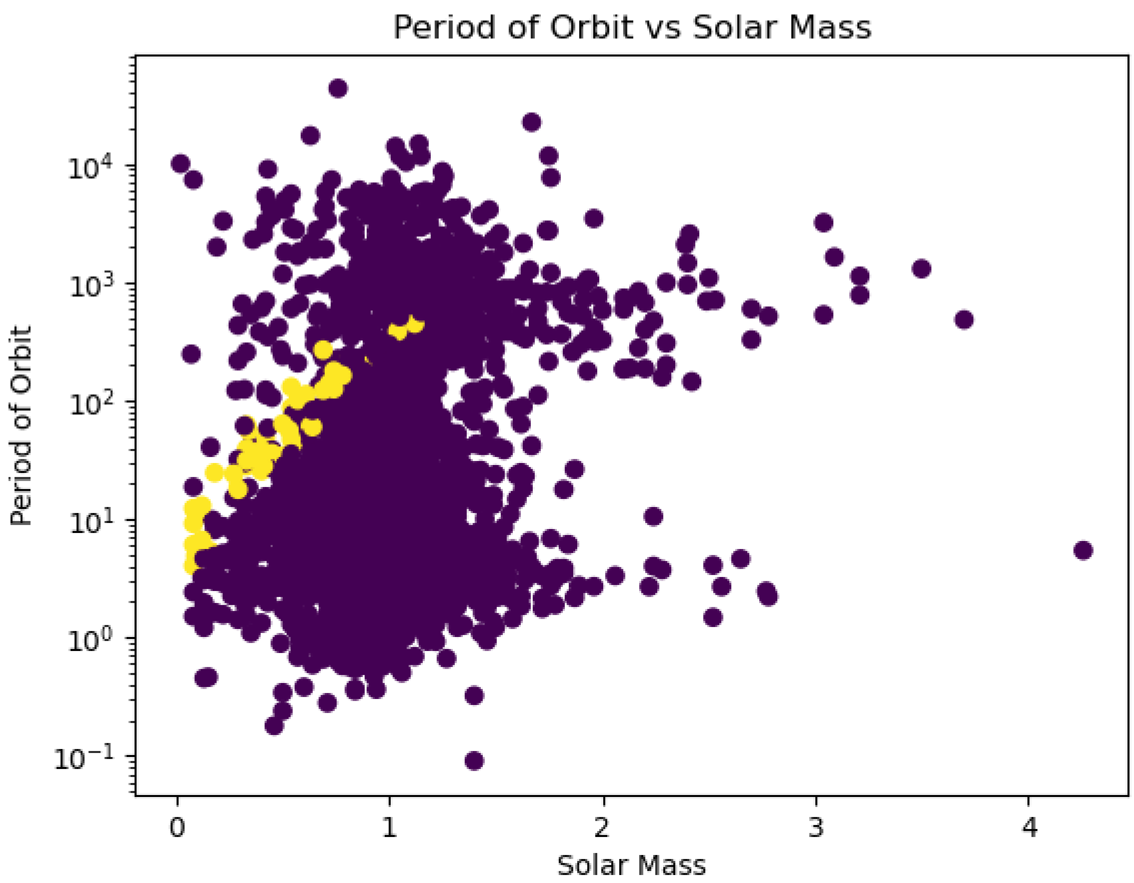
4.4 Step 3.3

Plot the period of orbit as a function of the mass of the parent star. It should look like fig 3.1 in our textbook.

Make sure your graph:

- has a legend (habitable versus not habitable)
- is a scatter plot
- has data (habitable versus not habitable) differentiated by colour
- has a log y-scale
- includes axis labels

In [21]:
```
fig = plt.figure()
legend_labels = filtered_df["P_HABITABLE"].unique()
plt.scatter(filtered_df["S_MASS"], filtered_df["P_PERIOD"],c = filtered_df["P_HABITABLE"])
plt.xlabel("Solar Mass")
plt.ylabel("Period of Orbit")
plt.yscale("log")
plt.title("Period of Orbit vs Solar Mass")

plt.show()
```



5 Part 4: Classification

5.1 Step 4.1

Implement train_test_split features and targets. Fix the random state to 3, You can use the default test size, which is 25%. This process will give you Xtrain, Xtest, ytrain, and ytest. Print the shapes of Xtrain and Xtest.

In [22]:
```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(filtered_df.iloc[:,:-1], filtered_df.iloc[:,-1], test_size=0.25,
```

Create a machine learning model by calling "KNeighborsClassifier". Remember that for the kNN algorithm, it is important to standardize the data since it relies on the notion of a metric. To this end, you can use the RobustScaler utility from sklearn.preprocessing. It is also recommended to construct a pipeline with the classifier so that the data is automatically scaled before given to the classifier algorithm. You can use Pipeline from sklearn.pipeline for this. The concept is explained in Chapter 3.4.

In [23]:
```
from sklearn.neighbors import KNeighborsClassifier
from sklearn.preprocessing import RobustScaler

scaler = RobustScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.fit_transform(X_test)

knn = KNeighborsClassifier()
knn.fit(X_train_scaled,y_train)
y_pred = knn.predict(X_test_scaled)
```

Let's see how many 1's ("Habitable") are predicted by our model. Count the number of ones in ytest, and then compare it to the number of ones in the y-array predicted from Xtest:

In [24]:
```
print("Number of planets classified as Habitable: " , bincount(y_pred)[1] )
print("Number of planets classified as  Uninhabitable: " , bincount(y_pred)[0] )
```

```
Number of planets classified as Habitable:  9
Number of planets classified as  Uninhabitable:  784
```

Let's check the performance of the classifier. Compute the accuracy, precision and recall scores using the **metrics** package for the test data. Also compute the performance of the "lazy" classifier that just assumes y=0 throughout. Any comments? What do these results mean for the success of our classifier? Can we improve our results by modifying the number of nearest neighbours in Step 4.1? Why or why not?

In [25]:
```python
from sklearn import metrics

lazy_pred = np.zeros(y_pred.shape[0])

accuracy = metrics.accuracy_score(y_test, y_pred)
precision = metrics.precision_score(y_test, y_pred)
recall = metrics.recall_score(y_test, y_pred)

lazy_accuracy = metrics.accuracy_score(y_test, lazy_pred)
lazy_precision = metrics.precision_score(y_test, lazy_pred)
lazy_recall = metrics.recall_score(y_test, lazy_pred)

print("Classifier Performance:")
print("Accuracy:", accuracy)
print("Precision:", precision)
print("Recall:", recall)

print("\nLazy Classifier Performance:")
print("Accuracy:", lazy_accuracy)
print("Precision:", lazy_precision)
print("Recall:", lazy_recall)
```

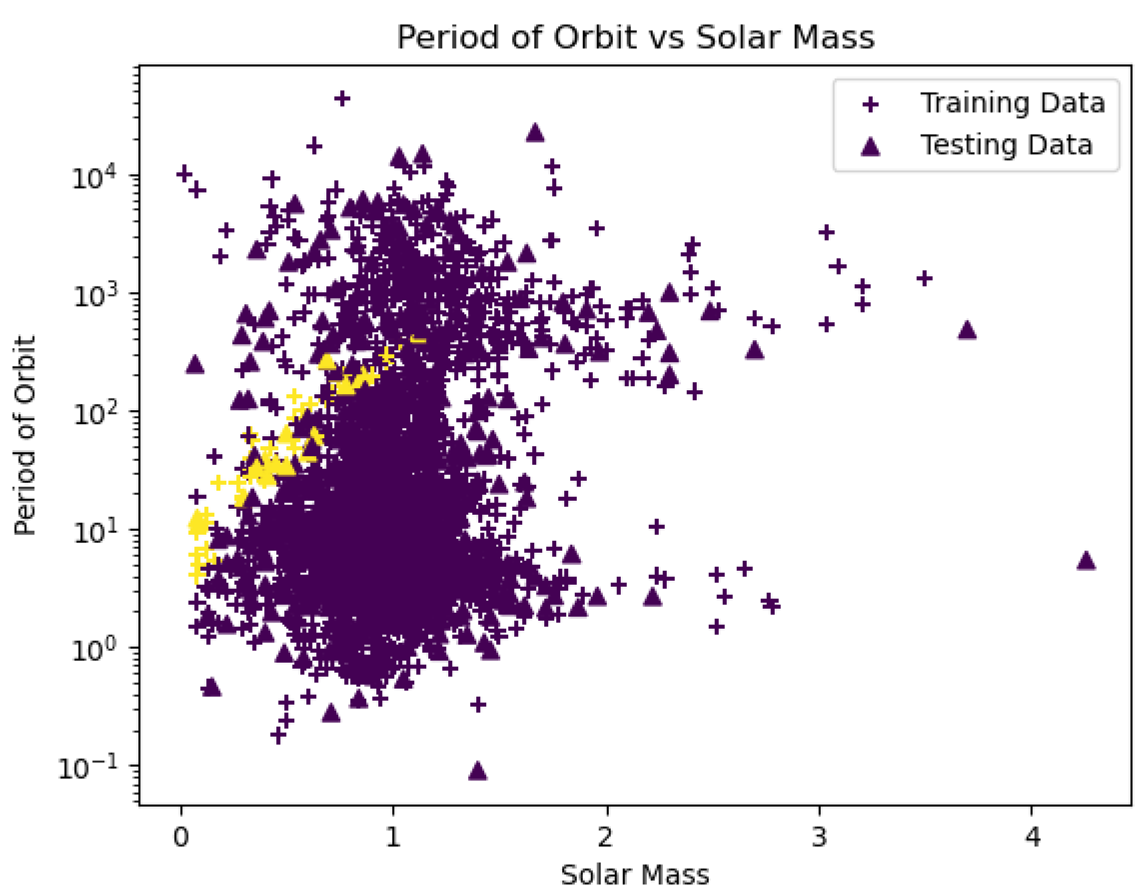
```
Classifier Performance:
Accuracy: 0.9836065573770492
Precision: 0.6666666666666666
Recall: 0.375

Lazy Classifier Performance:
Accuracy: 0.9798234552332913
Precision: 0.0
Recall: 0.0

C:\Users\kesha\miniconda3\envs\cs425\lib\site-packages\sklearn\metrics\_classification.py:1469: UndefinedMetricWarnin
g: Precision is ill-defined and being set to 0.0 due to no predicted samples. Use `zero_division` parameter to contro
l this behavior.
  _warn_prf(average, modifier, msg_start, len(result))
```

5.2 Step 4.2

Now let's plot the result. Use a scatter plot like in Step 3.3. Make sure that the training and testing points are represented by different shapes. Habitable and non-habitable points should be differentiated by colour as before. Label the axes and include a legend.

In [26]:
```python
fig = plt.figure()
legend_labels = filtered_df["P_HABITABLE"].unique()
plt.scatter(X_train["S_MASS"], X_train["P_PERIOD"], c = y_train, marker="+", label="Training Data")
plt.scatter(X_test["S_MASS"], X_test["P_PERIOD"], c = y_test, marker="^", label="Testing Data")

plt.legend()
plt.xlabel("Solar Mass")
plt.ylabel("Period of Orbit")
plt.yscale("log")
plt.title("Period of Orbit vs Solar Mass")
```

Out[26]: `Text(0.5, 1.0, 'Period of Orbit vs Solar Mass')`



5.3 Step 4.3

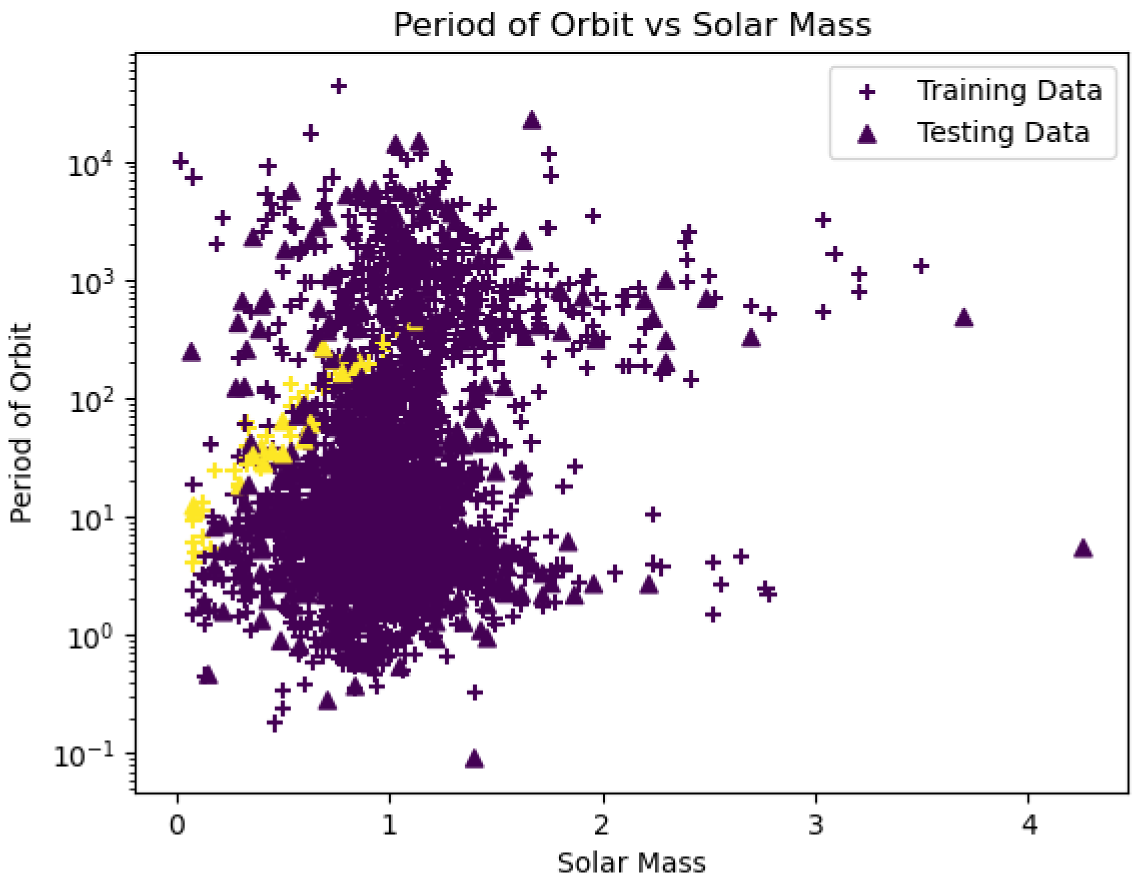
Repeat steps 4.1 and 4.2 with the DecisionTreeClassifier, random_state=42. Any comments on the results?

In [27]:
```
from sklearn.tree import DecisionTreeClassifier

dt = DecisionTreeClassifier(random_state=42)
dt.fit(X_train_scaled,y_train)
y_pred = dt.predict(X_test)

accuracy = metrics.accuracy_score(y_test, y_pred)
precision = metrics.precision_score(y_test, y_pred)
recall = metrics.recall_score(y_test, y_pred)

lazy_accuracy = metrics.accuracy_score(y_test, lazy_pred)
lazy_precision = metrics.precision_score(y_test, lazy_pred)
lazy_recall = metrics.recall_score(y_test, lazy_pred)

print("Classifier Performance:")
print("Accuracy:", accuracy)
print("Precision:", precision)
print("Recall:", recall)

print("\nLazy Classifier Performance:")
print("Accuracy:", lazy_accuracy)
print("Precision:", lazy_precision)
print("Recall:", lazy_recall)
```

```
Classifier Performance:
Accuracy: 0.9722572509457755
Precision: 0.0
Recall: 0.0

Lazy Classifier Performance:
Accuracy: 0.9798234552332913
Precision: 0.0
Recall: 0.0

C:\Users\kesha\miniconda3\envs\cs425\lib\site-packages\sklearn\base.py:457: UserWarning: X has feature names, but Dec
isionTreeClassifier was fitted without feature names
  warnings.warn(
C:\Users\kesha\miniconda3\envs\cs425\lib\site-packages\sklearn\metrics\_classification.py:1469: UndefinedMetricWarnin
g: Precision is ill-defined and being set to 0.0 due to no predicted samples. Use `zero_division` parameter to contro
l this behavior.
  _warn_prf(average, modifier, msg_start, len(result))
```

In [28]:
```
fig = plt.figure()
legend_labels = filtered_df["P_HABITABLE"].unique()
plt.scatter(X_train["S_MASS"], X_train["P_PERIOD"], c = y_train, marker="+", label="Training Data")
plt.scatter(X_test["S_MASS"], X_test["P_PERIOD"], c = y_test, marker="^", label="Testing Data")

plt.legend()
plt.xlabel("Solar Mass")
plt.ylabel("Period of Orbit")
plt.yscale("log")
plt.title("Period of Orbit vs Solar Mass")
```

Out[28]:
```
Text(0.5, 1.0, 'Period of Orbit vs Solar Mass')
```



6 Part 5: Cross-validation

Implement cross-validation on both models for the three metrics accuracy, precision, and recall with stratified k-folds, random shuffling, and 10 splits, random_state=10. What is k-fold validation, and what happens when stratification is applied? Report the mean CV score and the standard deviation.

In [32]:
```python
from sklearn.model_selection import cross_val_score, cross_validate, KFold

scoring = ['accuracy', 'precision_macro', 'recall_macro']
kfold = KFold(n_splits=10,shuffle= True, random_state=10)
scoring = ['accuracy', 'precision_macro', 'recall_macro']
scores = cross_validate(dt, X_train, y_train, cv=kfold, scoring=scoring)
results = {
    metric: {
        'average': round(scores[f"test_{metric}"].mean(), 3),
        'std_dev': round(scores[f"test_{metric}"].std(), 3)
    }
    for metric in scoring
}
print( "Decision Tree Metrics ")
print(results, "\n")

kfold = KFold(n_splits=10,shuffle= True, random_state=10)
scoring = ['accuracy', 'precision_macro', 'recall_macro']
scores = cross_validate(knn, X_train, y_train, cv=kfold, scoring=scoring)
results = {
    metric: {
        'average': round(scores[f"test_{metric}"].mean(), 3),
        'std_dev': round(scores[f"test_{metric}"].std(), 3)
    }
    for metric in scoring
}

print( "Knn Classifier Metrics ")
print(results)
```

```
Decision Tree Metrics
{'accuracy': {'average': 0.985, 'std_dev': 0.007}, 'precision_macro': {'average': 0.729, 'std_dev': 0.18}, 'recall_ma
cro': {'average': 0.743, 'std_dev': 0.21}}


C:\Users\kesha\miniconda3\envs\cs425\lib\site-packages\sklearn\metrics\_classification.py:1469: UndefinedMetricWarnin
g: Precision is ill-defined and being set to 0.0 in labels with no predicted samples. Use `zero_division` parameter t
o control this behavior.
  _warn_prf(average, modifier, msg_start, len(result))
C:\Users\kesha\miniconda3\envs\cs425\lib\site-packages\sklearn\metrics\_classification.py:1469: UndefinedMetricWarnin
g: Precision is ill-defined and being set to 0.0 in labels with no predicted samples. Use `zero_division` parameter t
o control this behavior.
  _warn_prf(average, modifier, msg_start, len(result))
C:\Users\kesha\miniconda3\envs\cs425\lib\site-packages\sklearn\metrics\_classification.py:1469: UndefinedMetricWarnin
g: Precision is ill-defined and being set to 0.0 in labels with no predicted samples. Use `zero_division` parameter t
o control this behavior.
  _warn_prf(average, modifier, msg_start, len(result))
C:\Users\kesha\miniconda3\envs\cs425\lib\site-packages\sklearn\metrics\_classification.py:1469: UndefinedMetricWarnin
g: Precision is ill-defined and being set to 0.0 in labels with no predicted samples. Use `zero_division` parameter t
o control this behavior.
  _warn_prf(average, modifier, msg_start, len(result))
C:\Users\kesha\miniconda3\envs\cs425\lib\site-packages\sklearn\metrics\_classification.py:1469: UndefinedMetricWarnin
g: Precision is ill-defined and being set to 0.0 in labels with no predicted samples. Use `zero_division` parameter t
o control this behavior.
  _warn_prf(average, modifier, msg_start, len(result))
C:\Users\kesha\miniconda3\envs\cs425\lib\site-packages\sklearn\metrics\_classification.py:1469: UndefinedMetricWarnin
g: Precision is ill-defined and being set to 0.0 in labels with no predicted samples. Use `zero_division` parameter t
o control this behavior.
  _warn_prf(average, modifier, msg_start, len(result))
C:\Users\kesha\miniconda3\envs\cs425\lib\site-packages\sklearn\metrics\_classification.py:1469: UndefinedMetricWarnin
g: Precision is ill-defined and being set to 0.0 in labels with no predicted samples. Use `zero_division` parameter t
o control this behavior.
  _warn_prf(average, modifier, msg_start, len(result))
C:\Users\kesha\miniconda3\envs\cs425\lib\site-packages\sklearn\metrics\_classification.py:1469: UndefinedMetricWarnin
g: Precision is ill-defined and being set to 0.0 in labels with no predicted samples. Use `zero_division` parameter t
o control this behavior.
  _warn_prf(average, modifier, msg_start, len(result))

Knn Classifier Metrics
{'accuracy': {'average': 0.985, 'std_dev': 0.009}, 'precision_macro': {'average': 0.492, 'std_dev': 0.004}, 'recall_m
acro': {'average': 0.5, 'std_dev': 0.0}}

C:\Users\kesha\miniconda3\envs\cs425\lib\site-packages\sklearn\metrics\_classification.py:1469: UndefinedMetricWarnin
g: Precision is ill-defined and being set to 0.0 in labels with no predicted samples. Use `zero_division` parameter t
o control this behavior.
  _warn_prf(average, modifier, msg_start, len(result))
C:\Users\kesha\miniconda3\envs\cs425\lib\site-packages\sklearn\metrics\_classification.py:1469: UndefinedMetricWarnin
g: Precision is ill-defined and being set to 0.0 in labels with no predicted samples. Use `zero_division` parameter t
o control this behavior.
  _warn_prf(average, modifier, msg_start, len(result))
```

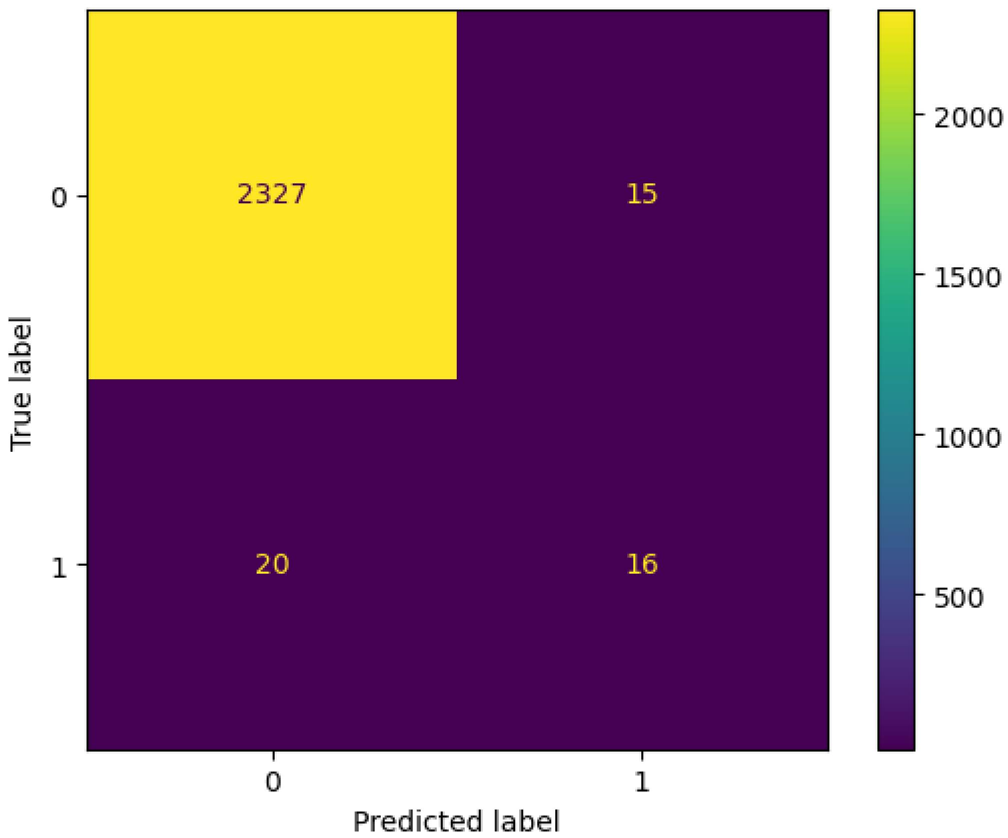
7 Part 6: Confusion Matrix

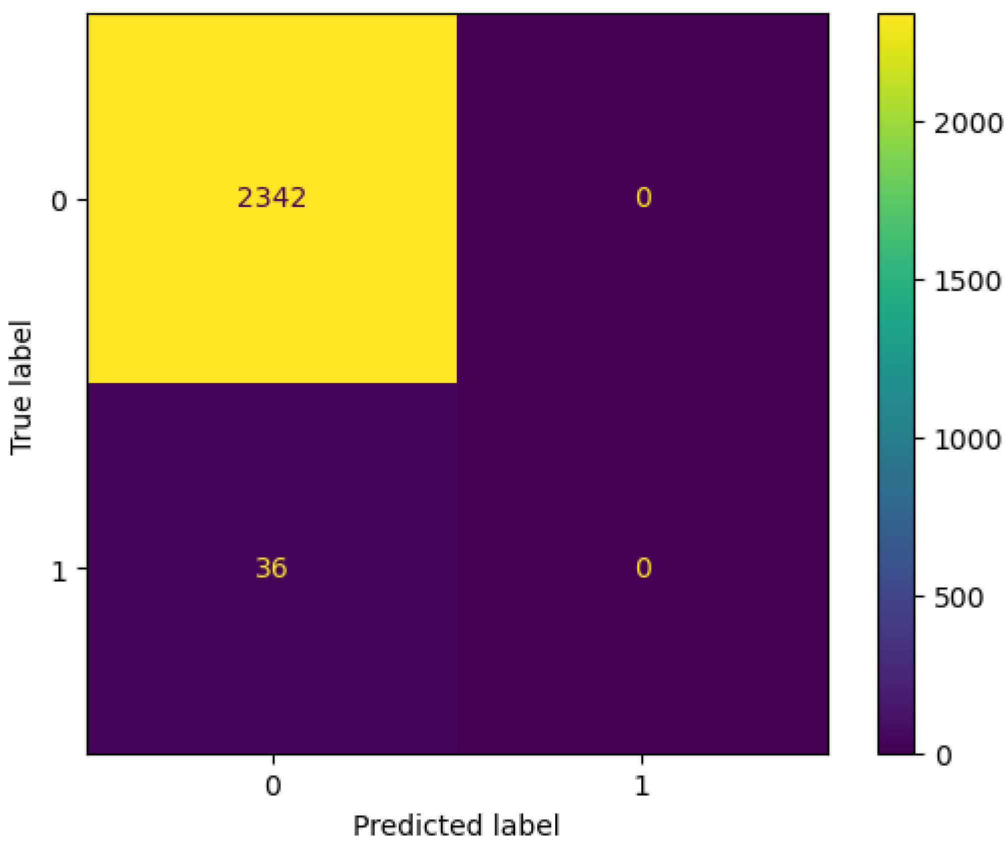
Construct the confusion matrices for the two classifiers from above. Please write code from scratch to compute the matrix elements, but feel free to check it against the function provided by sklearn.metrics. Use **cross_val_predict** to assemble the predictions of several folds. You can then

In [38]:

```
from sklearn.metrics import ConfusionMatrixDisplay
from sklearn.model_selection import cross_val_predict


def confusion_matrix_manual(y_true, y_pred):
    classes = np.unique(y_true)
    matrix = np.zeros((len(classes), len(classes)), dtype=int)

    for i, true_class in enumerate(classes):
        for j, pred_class in enumerate(classes):
            matrix[i, j] = np.sum((y_true == true_class) & (y_pred == pred_class))

    return matrix
####### Decision Tree ##########
y_pred_dt = cross_val_predict(dt, X_train, y_train,cv=10)
cm = confusion_matrix_manual(y_train,y_pred_dt)
plot = ConfusionMatrixDisplay(cm)
plot.plot()
####### Knn ####################
y_pred_knn = cross_val_predict(knn, X_train, y_train,cv=10)
cm = confusion_matrix_manual(y_train,y_pred_knn)
plot = ConfusionMatrixDisplay(cm)
plot.plot()
```

Out[38]: <sklearn.metrics._plot.confusion_matrix.ConfusionMatrixDisplay at 0x178f3e0ff10>



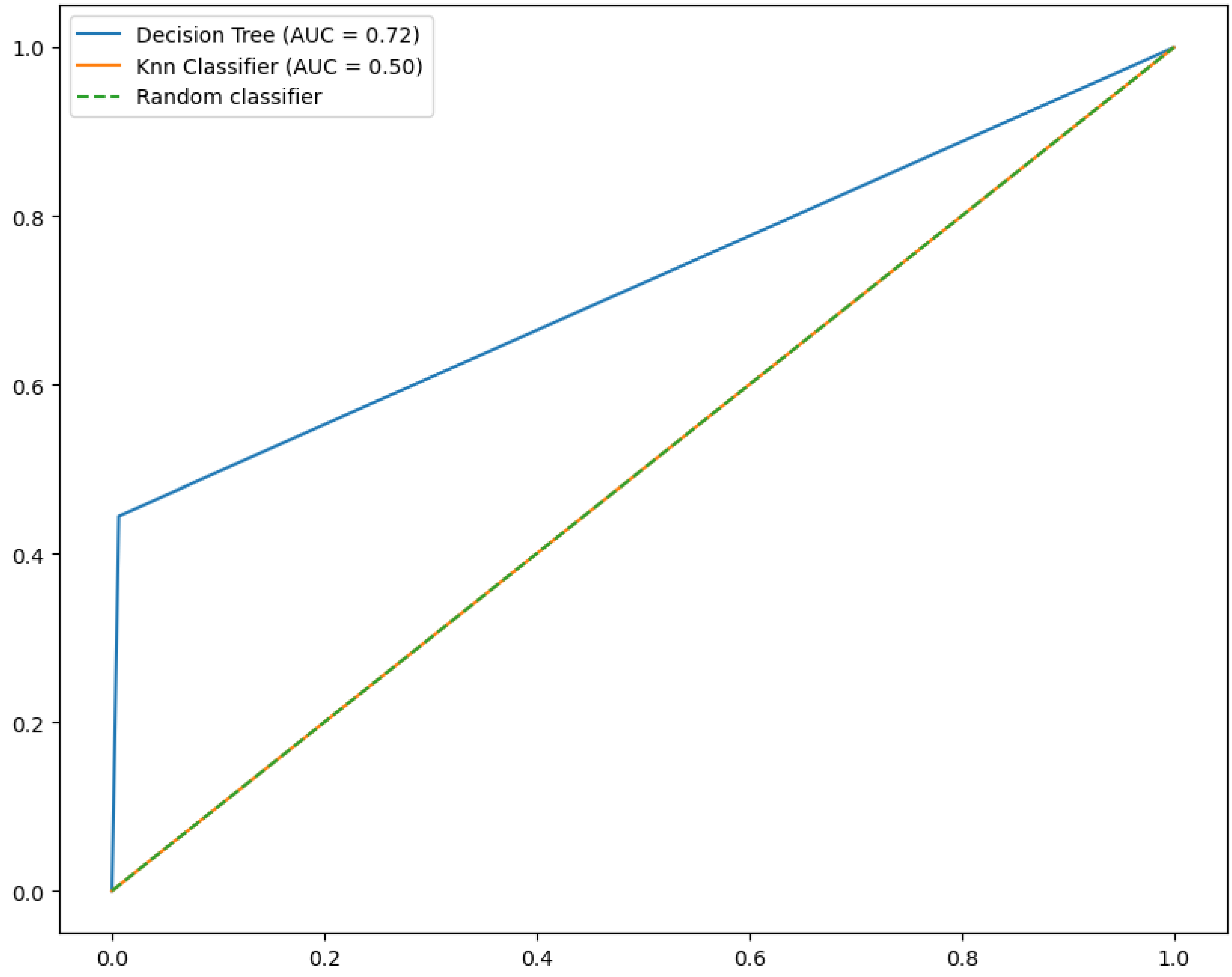


8 Part 7: ROC curve

Construct the ROC curves for both classifiers. Please use **sklearn's metrics.roc_curve()** to compute the points curve and then plot them. Also draw the ROC curve of a useless (i.e. random) classifier. Don't forget axis labels and a legend. Use cross_val_predict to obtain the probabilities of the predictions. Please also report the ROC AUC score for both classifiers as an average over the 10 folds, using **cross_val_score()**.

In [41]:
```
from sklearn.metrics import roc_curve, roc_auc_score

# Compute ROC curve points
fpr1, tpr1, _ = roc_curve(y_train, y_pred_dt)
fpr2, tpr2, _ = roc_curve(y_train, y_pred_knn)

# Compute ROC AUC scores
roc_auc1 = roc_auc_score(y_train, y_pred_dt)
roc_auc2 = roc_auc_score(y_train, y_pred_knn)

# Plot the ROC curves
plt.figure(figsize=(10, 8))
plt.plot(fpr1, tpr1, label=f'Decision Tree (AUC = {roc_auc1:.2f})')
plt.plot(fpr2, tpr2, label=f'Knn Classifier (AUC = {roc_auc2:.2f})')

# Plot the random classifier line
plt.plot([0, 1], [0, 1], linestyle='--', label='Random classifier')
plt.legend()
```

Out[41]:
```
<matplotlib.legend.Legend at 0x178f49b1900>
```



9 Part 8: Learning curves

With the help of the module **learning_curve** from sklearn.model_selection, construct the learning curves for both classifiers as shown in Fig. 3.7.

In [ ]:

In [ ]:

In [ ]:

10 Part 9: Should we exist?

Use the kNN and DT classifiers that you trained above to predict the habitability of the earth.

In [47]:
```
Earth_Features = np.array([1,365,1]).reshape(1,-1)

knn_earth = knn.predict(Earth_Features)
dt_earth = dt.predict(Earth_Features)

print("According to both the Decision Tree and the Knn Classifier, Earth would not be habitable")
```

```
According to both the Decision Tree and the Knn Classifier, Earth would not be habitable
```