In [51]:
```python
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

from sklearn.svm import SVC
from sklearn.tree import DecisionTreeClassifier
from sklearn.neighbors import KNeighborsClassifier

from sklearn.feature_selection import SelectKBest, mutual_info_classif
from mlxtend.feature_selection import SequentialFeatureSelector as SFS

from sklearn.preprocessing import StandardScaler, MinMaxScaler
import warnings
warnings.filterwarnings("ignore", category=FutureWarning)

from sklearn.preprocessing import LabelBinarizer
from sklearn.model_selection import train_test_split, KFold, StratifiedKFolc
from sklearn.metrics import ConfusionMatrixDisplay, balanced_accuracy_score,
from sklearn.dummy import DummyClassifier
from sklearn.preprocessing import LabelBinarizer
from itertools import combinations
```

```
[CV] END .................................................... total time=
0.4s
[CV] END .................................................... total time=
0.3s
[CV] END .................................................... total time=
4.0s
[CV] END .................................................... total time=
5.3s
[CV] END .................................................... total time=
0.1s
[CV] END .................................................... total time=
0.1s
[CV] END .................................................... total time=
0.1s
[CV] END .................................................... total time=
0.3s
[CV] END .................................................... total time=
0.3s
[CV] END .................................................... total time=
0.2s
[CV] END .................................................... total time=
0.1s
[CV] END .................................................... total time=
0.2s
[CV] END .................................................... total time=
0.3s
[CV] END .................................................... total time=
0.3s
[CV] END .................................................... total time=
0.3s
```

```
[CV] END ............................................ total time=  1
2.5s
[CV] END ............................................ total time=  1
8.4s
[CV] END ............................................ total time=  1
2.6s
[CV] END ............................................ total time=  3
7.9s
[CV] END ............................................ total time=  4
8.3s
```

Task 1: Data Quality Plan

In [2]:
```
df = pd.read_csv('24291444.csv')
df.head()
```

Out[2]:

	obj_ID	alpha	delta	u	g	r	i	
0	1.237663e+18	346.772076	0.798375	19.04310	18.74766	18.68006	18.49899	18.
1	1.237662e+18	215.945528	47.836140	23.82031	22.18570	20.93554	19.75348	19.
2	1.237661e+18	139.388332	35.051326	19.31936	19.10250	19.09940	18.90497	19.(
3	1.237655e+18	246.485139	47.140976	21.99707	21.67077	21.46923	21.49820	21.
4	1.237664e+18	116.662350	49.729022	22.23060	21.72280	21.47487	21.12132	20.

In [3]:
```
df.shape
```

Out[3]: (30000, 18)

In [4]:
```
df.dtypes
```

Out[4]:
```
obj_ID         float64
alpha          float64
delta          float64
u              float64
g              float64
r              float64
i              float64
z              float64
run_ID           int64
rerun_ID         int64
cam_col          int64
field_ID         int64
spec_obj_ID    float64
class           object
redshift       float64
plate            int64
MJD              int64
fiber_ID         int64
dtype: object
```

All the data feature types are numeric except the 'class' feature which is expected as class labels observations as ['Galaxy','STAR','QSO']

In [5]:
```
df.nunique()
```

Out[5]:
```
obj_ID         27431
alpha          30000
delta          30000
u              29404
g              29296
r              29220
i              29255
z              29232
run_ID           406
rerun_ID           1
cam_col            6
field_ID         817
spec_obj_ID    30000
class              3
redshift       29838
plate           5731
MJD             2124
fiber_ID        1000
dtype: int64
```

Mostly a unique value per observation for most of the photometric features, maybe due to the sensitivity of the data collection machines, additionally, there are duplicates present in the data, as obj_ID is a unique identifier and is less than the 30000 entries we have, we are going to have to remove the duplicates

In [6]:
```
df['class'].value_counts() / df.shape[0]
```

Out[6]:
```
class
GALAXY    0.592000
STAR      0.218167
QSO       0.189833
Name: count, dtype: float64
```

This data set is extremely unbalanced, as 59% of data are galaxies, 22% are stars and 19% are QSOs. For classification and evaluation, we will have to take this class imbalance into account

In [7]:
```python
df[df['obj_ID'].duplicated()]
```

Out[7]:

	obj_ID	alpha	delta	u	g	r	
425	1.237679e+18	20.288059	11.581189	21.65558	20.54378	20.30998	20.2110
737	1.237661e+18	164.050825	44.450899	20.34882	20.09278	19.94207	19.8999
961	1.237663e+18	341.846341	0.095160	20.79568	19.64293	18.87432	18.4387
1009	1.237664e+18	119.873909	51.729620	20.17974	18.46395	17.44095	16.9242
1233	1.237662e+18	176.465337	39.345863	20.39129	20.22710	20.01783	19.9064
...	...	...	...	...	...	...	.
29971	1.237658e+18	150.694800	45.849887	21.21550	20.86231	20.74509	20.3944
29982	1.237663e+18	339.218078	0.157844	25.55405	21.77067	20.18976	18.9573
29986	1.237663e+18	353.031463	0.649624	23.76088	22.15661	21.47430	20.7114
29988	1.237679e+18	28.577628	-2.252339	22.74284	21.17147	19.78127	19.2912
29997	1.237679e+18	8.784438	1.648707	23.10574	21.20415	20.97670	20.9801

2569 rows × 18 columns

There are 2569 duplicated items in this dataset, as sorted by obj_ID, which is supposed to be a unique identifier of these values. Therefore, will need to remove these duplicates

In [8]:
```python
df.describe()
```

Out[8]:

	obj_ID	alpha	delta	u	g	
count	3.000000e+04	30000.000000	30000.000000	30000.000000	30000.000000	30
mean	1.237665e+18	177.752891	24.032529	22.072917	20.622373	
std	8.433602e+12	96.608486	19.610344	2.250289	2.036687	
min	1.237646e+18	0.005528	-17.613056	12.262400	10.511390	
25%	1.237659e+18	127.616506	5.132893	20.335938	18.930343	
50%	1.237663e+18	181.090496	23.328810	22.180605	21.084350	
75%	1.237668e+18	234.268384	39.794247	23.674910	22.118523	
max	1.237681e+18	359.999615	83.000519	30.660390	30.607000	

Observations:

- All numerical data, except class which is categorical
- rerun_ID only has one value, irrelevant feature
- cal_col has 6 values potentially irrelevant
- runID only 406 values
- Modified Julian Date (MJD) values might not be relevant for this classification task, as stars do not change with respect to dates
- there exists duplicate data
- Dataset contains three identifier columns, which can lead to extreme overfitting

1. Drop duplicates
2. The spec object is different for all samples
3. check for outliers in redshift, u,g,r, i,z
4. normalize redshift and photometric

In [9]:

```
#keep raw df
df_raw = df

# drop all duplicates for the same object identifier
df = df[~df['obj_ID'].duplicated()]
```

In [10]:

```
# Removing irrelevant columns
df.drop(columns=['spec_obj_ID', 'obj_ID','run_ID','rerun_ID','field_ID','cam
df.head()
```

Out[10]:

	alpha	delta	u	g	r	i	z	class
0	346.772076	0.798375	19.04310	18.74766	18.68006	18.49899	18.24167	QSO
1	215.945528	47.836140	23.82031	22.18570	20.93554	19.75348	19.13016	GALAXY
2	139.388332	35.051326	19.31936	19.10250	19.09940	18.90497	19.00351	QSO
3	246.485139	47.140976	21.99707	21.67077	21.46923	21.49820	21.13974	QSO
4	116.662350	49.729022	22.23060	21.72280	21.47487	21.12132	20.75071	GALAXY

In [11]:

```python
df.describe().T
```

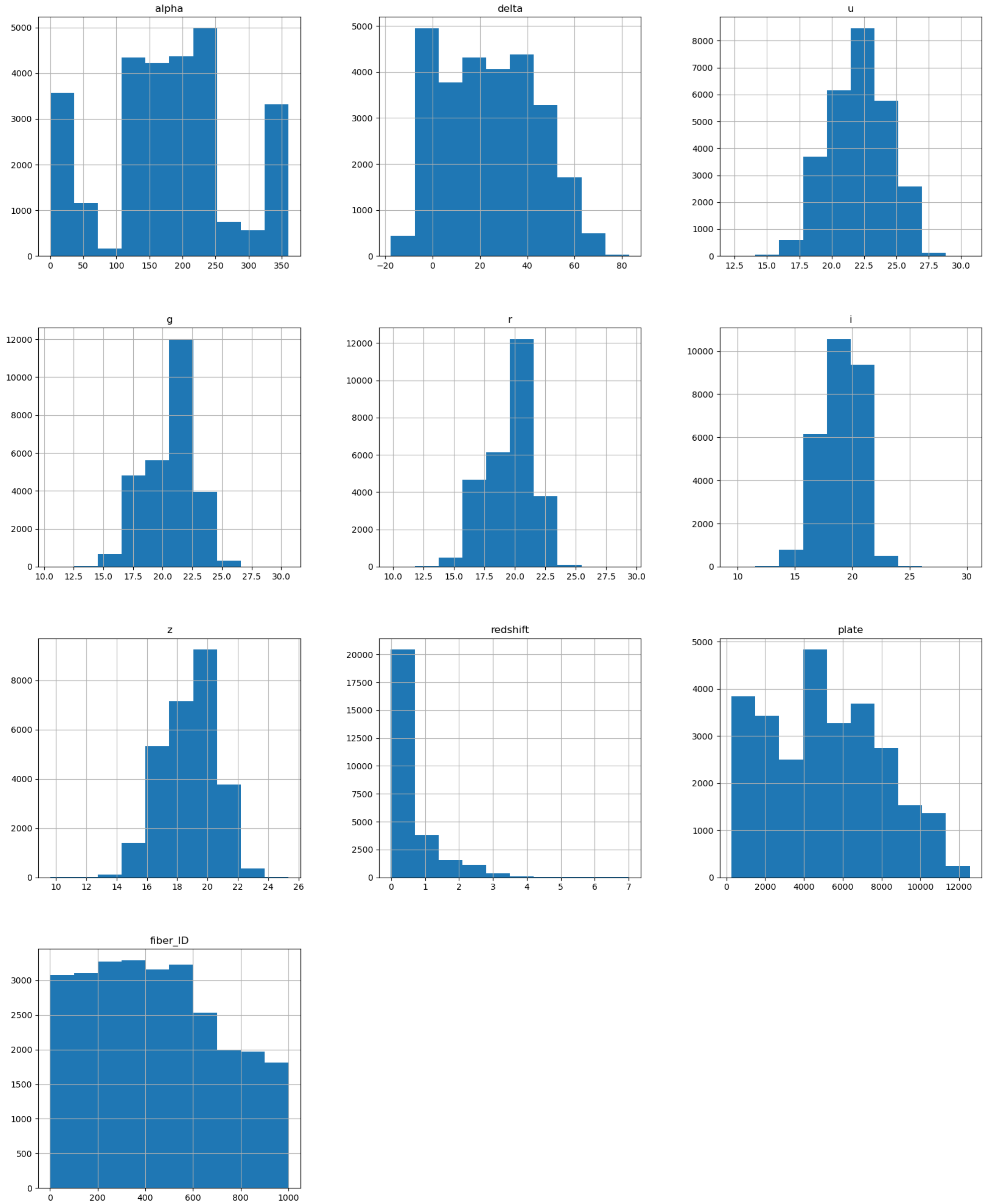
Out[11]:

	count	mean	std	min	25%	50%
alpha	27431.0	178.929725	96.075539	0.005528	128.557706	181.996685
delta	27431.0	24.075934	19.515966	-17.613056	5.606833	23.263707
u	27431.0	22.107274	2.256328	12.262400	20.360250	22.236840
g	27431.0	20.651527	2.037369	10.511390	18.959995	21.130090
r	27431.0	19.659585	1.847926	9.822070	18.144585	20.138530
i	27431.0	19.089482	1.743525	9.469903	17.738950	19.405670
z	27431.0	18.772540	1.750402	9.612333	17.463855	19.005920
redshift	27431.0	0.580647	0.740079	-0.009971	0.055789	0.428098
plate	27431.0	5124.311764	2912.718915	266.000000	2564.000000	4978.000000
fiber_ID	27431.0	449.189968	272.506128	1.000000	221.000000	432.000000

Presumed relevant feature exploration

In [12]:

```python
df.hist(figsize=(20, 25), layout=(4,3));
plt.grid(which='major', linestyle='-');
```



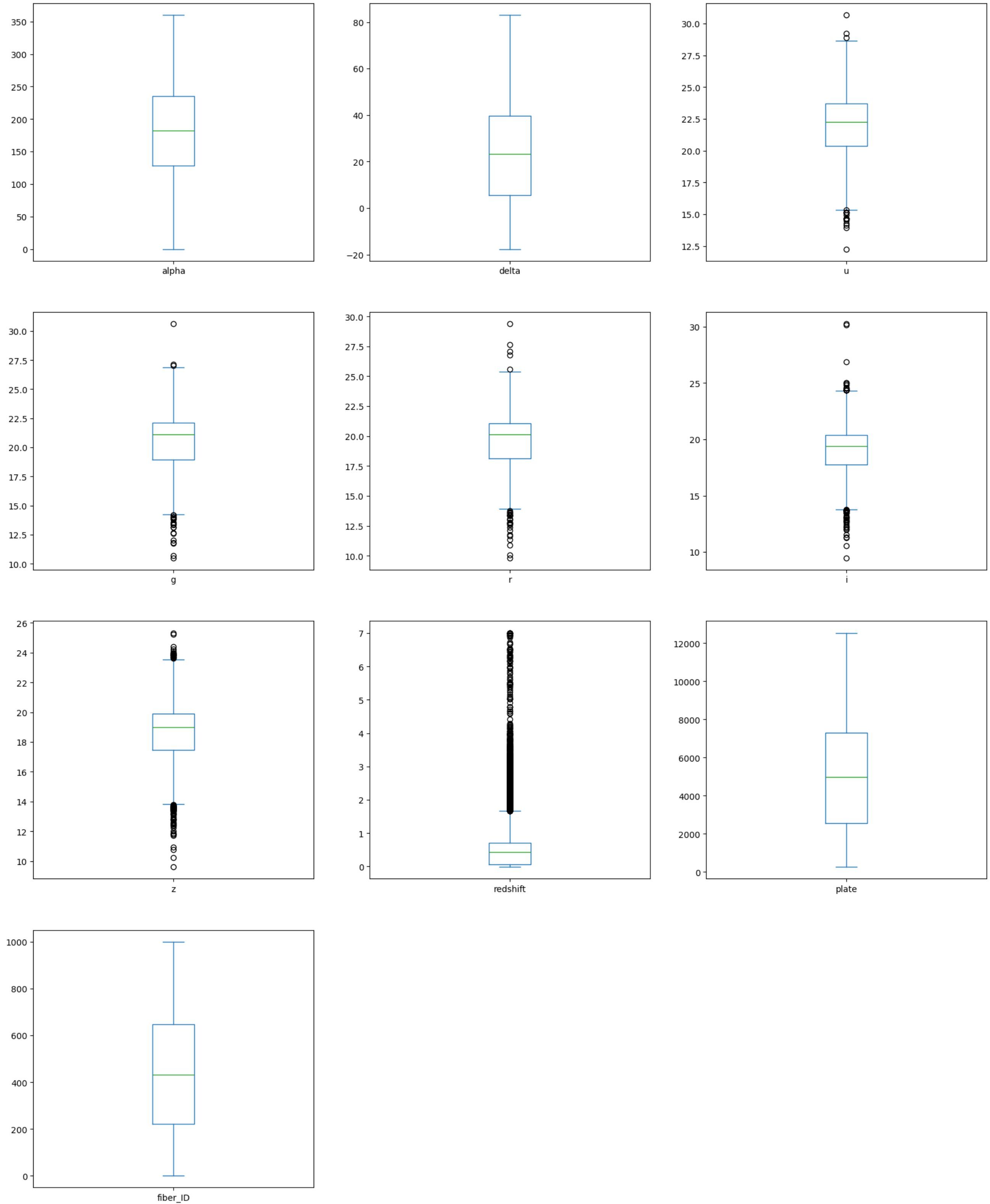
Observations

- alpha values seem 'clamped' at the extremes
- delta values seem normally distributed
- Photometric values: 'u','g','r','i', and 'z' seem psudo-normally distributed within the observations.

- Redshift values are severely right-skewed, a transformation might need to be done to normalize this distribution
- Plate, MJD, fiber do not seem to follow any distribution

In [13]:

```
numeric_columns = df.select_dtypes(['int64', 'float64']).columns
df[numeric_columns].plot(kind='box', subplots=True, figsize=(20,25), layout=
```



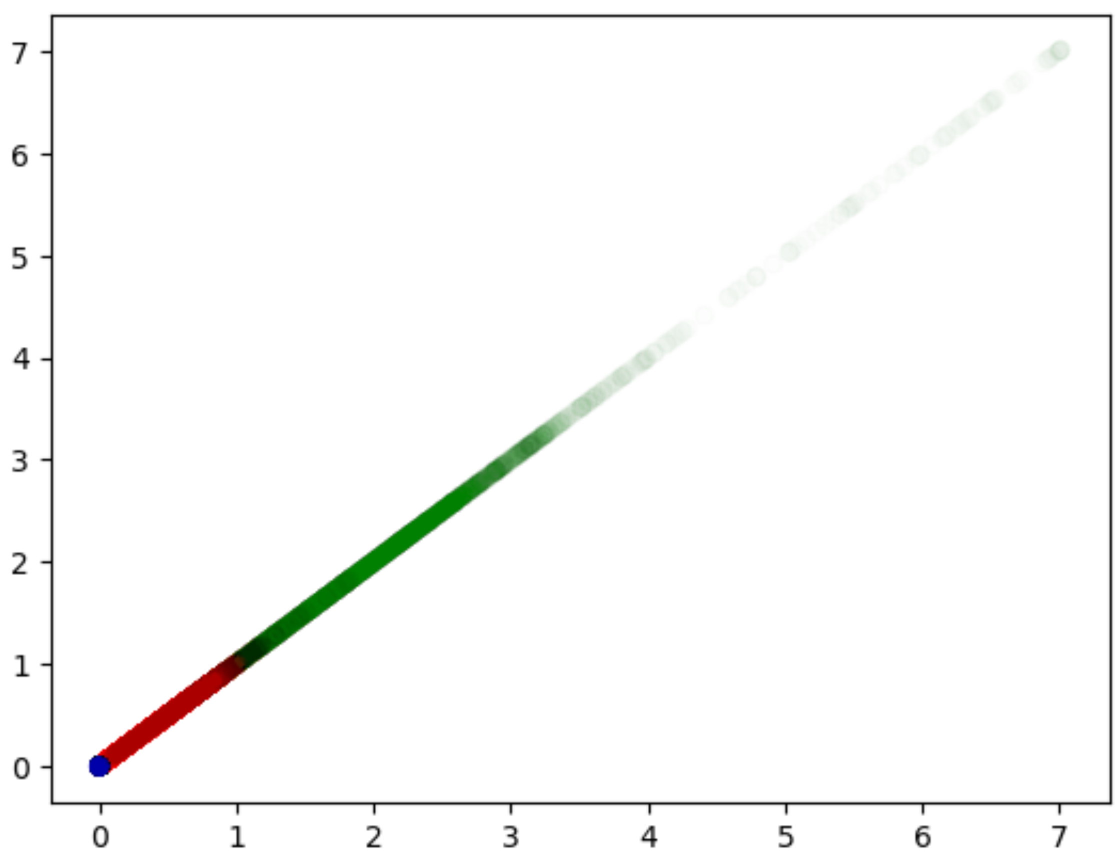
Observations

- alpha, delta, plate, MJD, and fiberID do not have any outliers
- photometric features 'u','g','r','i','z' have outliers on the high and lower end, but not very dense outliers region
- redshift is significanly right-skewed and will need a transformation as there is heavy high outliers in redshift distribution

Data Transformation

Investigation into redshift outliers

In [14]:
```
# plotting redshift against itself and seeing how redshift values might affe
colors = {'GALAXY': 'red', 'STAR': 'blue', 'QSO':'green'}
plt.scatter(df['redshift'],df['redshift'],c=df['class'].map(colors),label=df
```



From the plot above, we can see that Redshift values vs. class are almost linearly separable, this means that we could clamp outliers to 95th percentile as QSO observations get rarer in high redshift values.

REDSHIFT VALUES TRANSFORMATION

Because of the heavy-right skew in its distribution, I am going to log-scale this distribution to make it more normally distributed after clamping, as there still are outliers

In [15]:
```
percentile95 = np.percentile(df['redshift'],95)

redshiftOutliers = df[df['redshift'] > percentile95]
redshiftOutliers
```

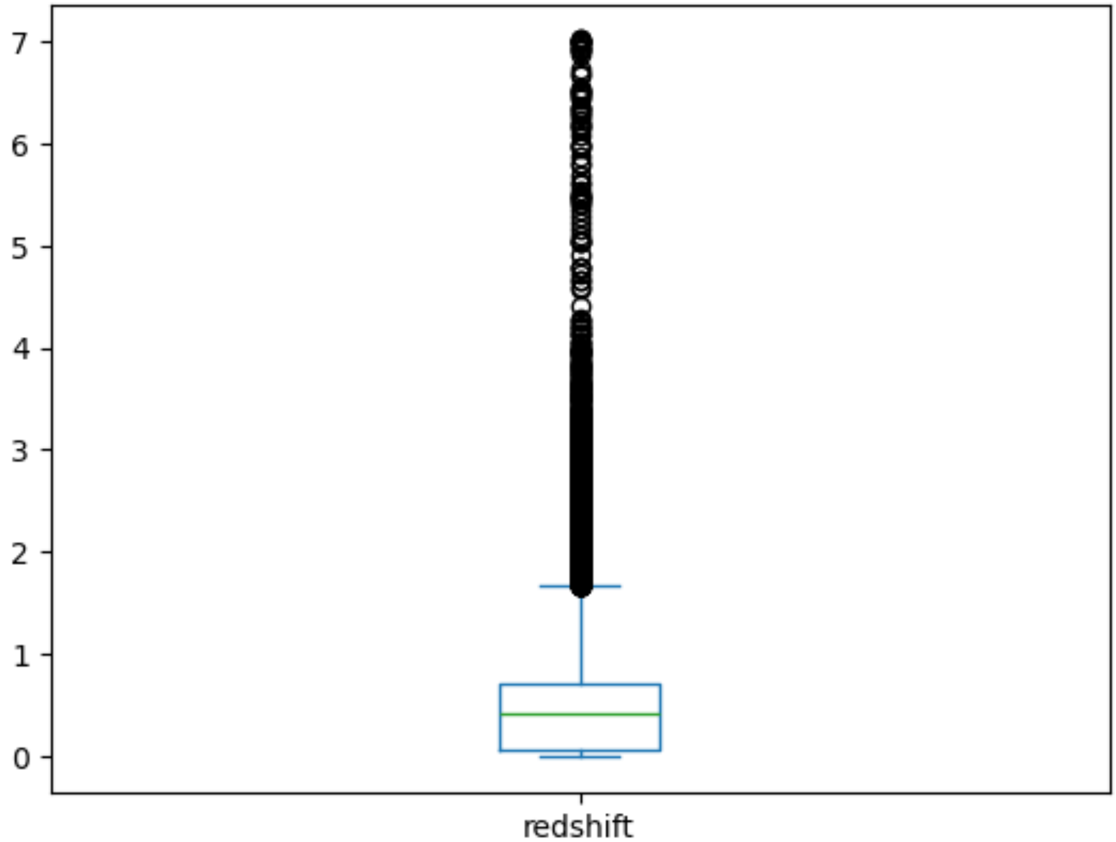
Out[15]:

	alpha	delta	u	g	r	i	z	c
52	184.641753	43.494613	22.95321	22.43908	22.01231	22.16328	22.24051	
66	236.821944	25.160369	23.70864	20.77560	19.71128	19.61595	19.82036	
135	355.897283	12.280289	20.69727	20.00488	19.75740	19.63135	19.41086	
147	20.370838	11.450133	18.48399	17.59546	17.25394	17.03100	16.70826	
186	9.589761	2.576984	23.73365	22.13621	22.31480	22.35993	21.93321	
...	...	...	...	...	...	...	...	
29907	190.187804	11.619725	21.66430	20.50776	20.26739	20.05617	19.95164	
29911	170.567628	41.293285	21.63541	21.22299	21.12925	21.03539	20.61155	
29945	155.517159	38.380584	21.03028	20.22529	20.26563	20.26219	20.27242	
29959	239.489542	44.440527	20.86616	19.68577	19.44610	19.34450	19.30692	
29989	239.222740	23.983202	21.75931	20.59475	20.34945	20.03026	19.55657	

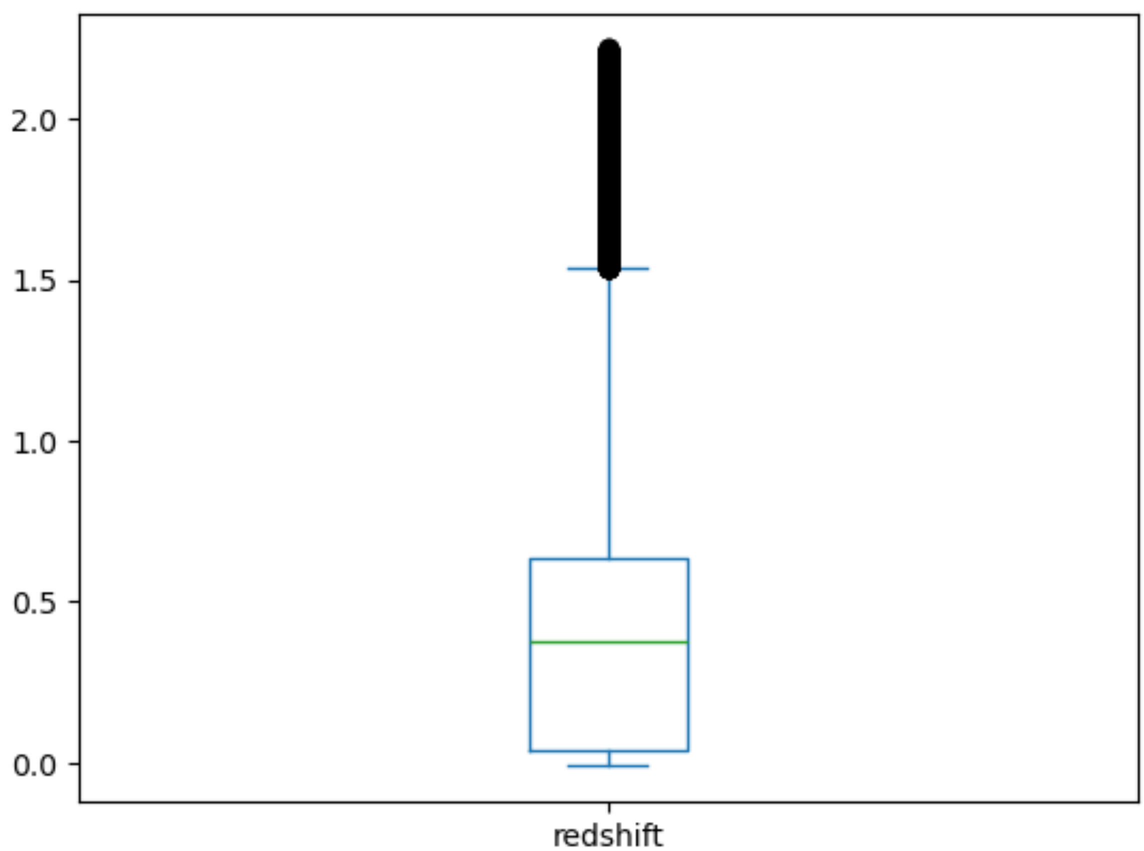
1372 rows × 11 columns

Clamping will affect 1500 rows of data

In [16]:
```
df['redshift'].plot.box();
```

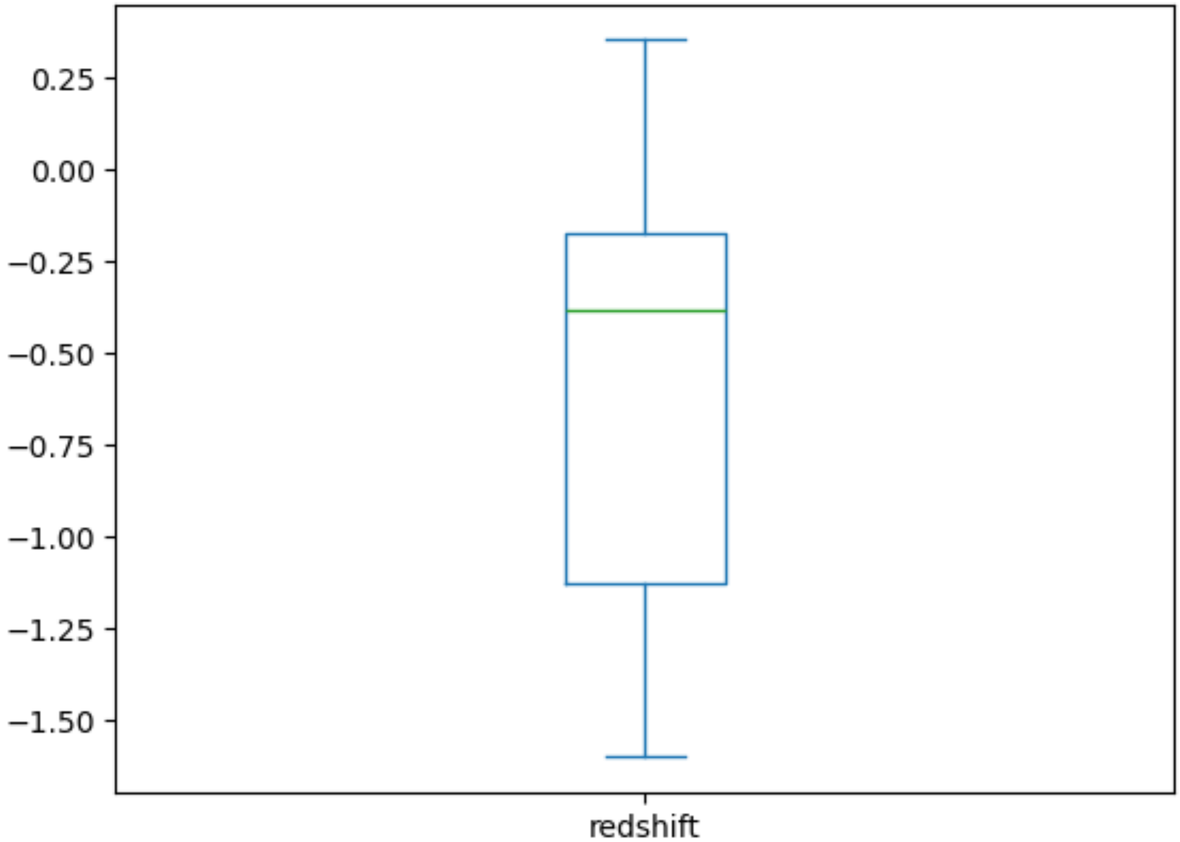


In [17]:
```
# show the box plot after clamping the redshift feature, still heavy outlier
df = df[df['redshift'] < percentile95]
df['redshift'].plot.box();
```



After clamping, distribution is still heavily right-skewed, will attempt to apply a log transformation to reduce skew

In [18]:
```
df['redshift'].transform(lambda x: np.log10(0.035+x)).plot.box();
```



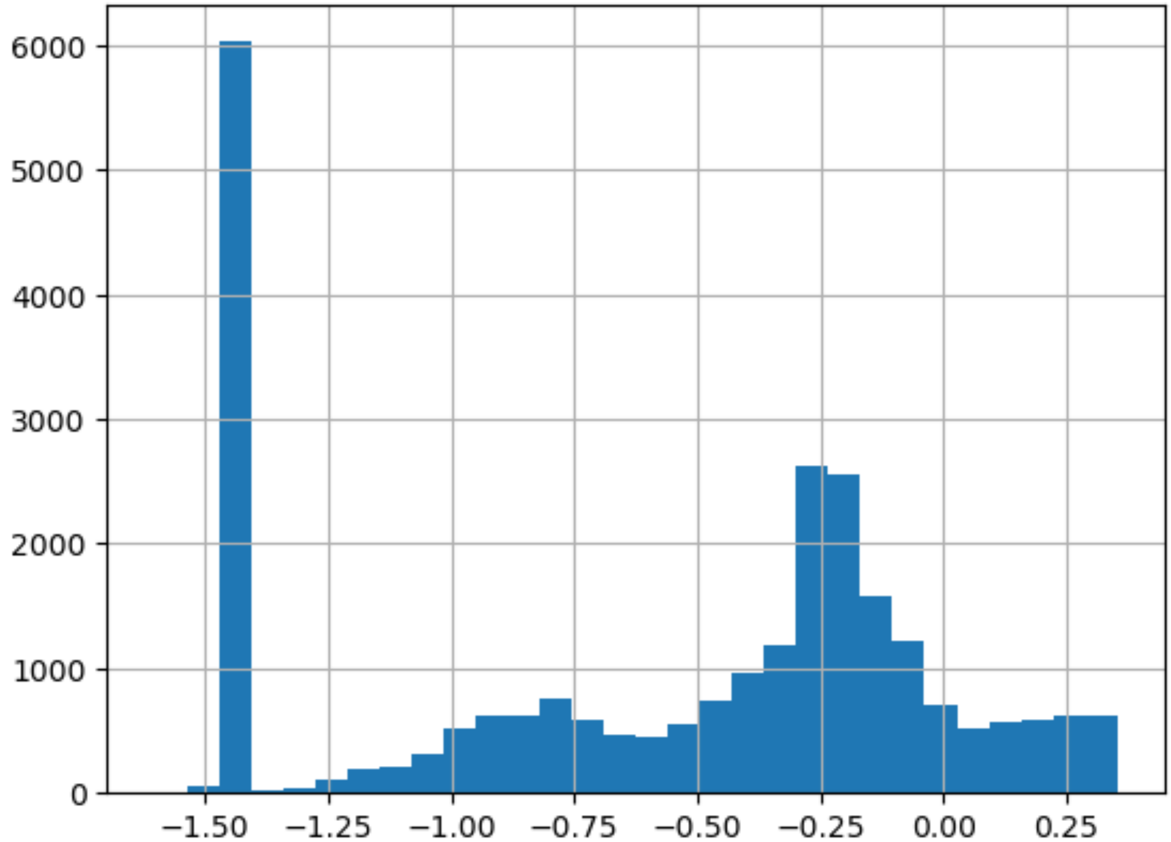
Much better distribution than previous, with no outliers. will place this in effect

In [19]:
```
# See class distribution for redshift outliers
redshiftOutliers['class'].value_counts()
```

Out[19]:
```
class
QSO    1372
Name: count, dtype: int64
```

In [20]:
```
# transform redshift
df.loc[:,'redshift'] =  df['redshift'].transform(lambda x: np.log10(0.035+x)
```

In [21]:
```
#plot new redshift distribution
df['redshift'].hist(bins=30);
```

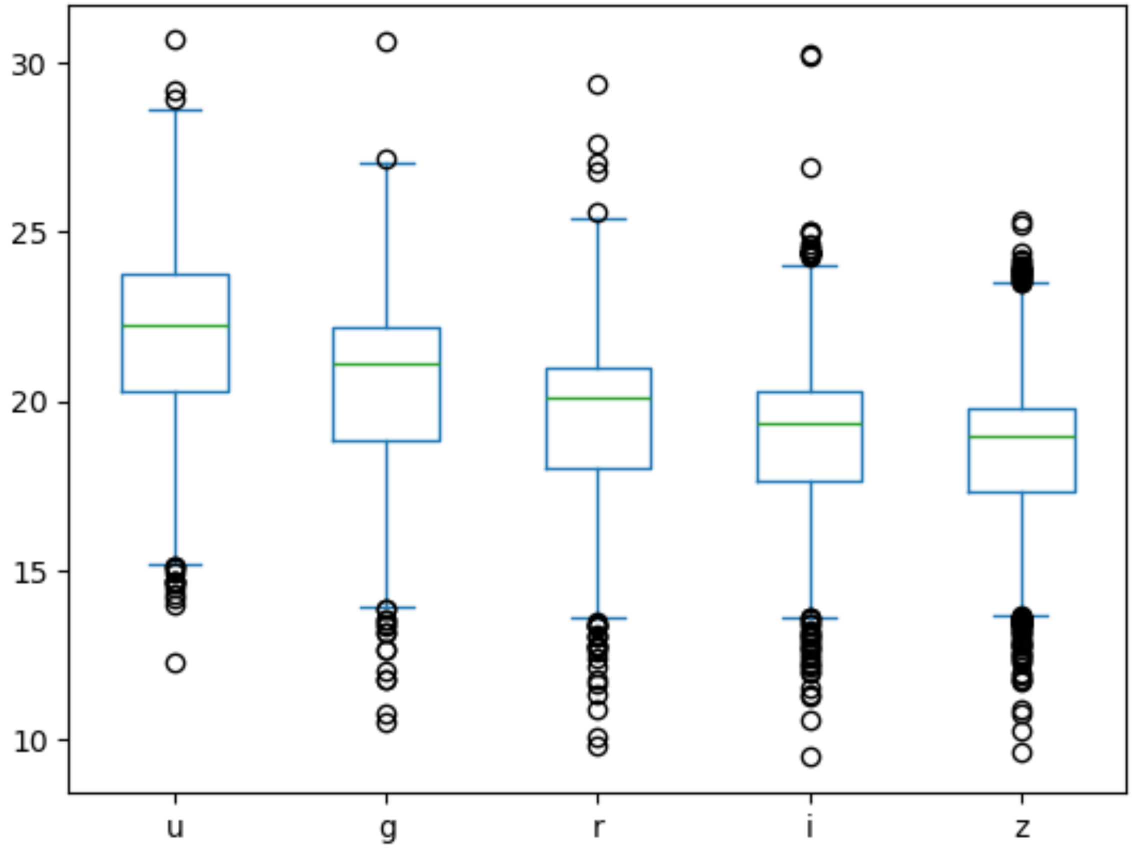


Conclusion for redshift values

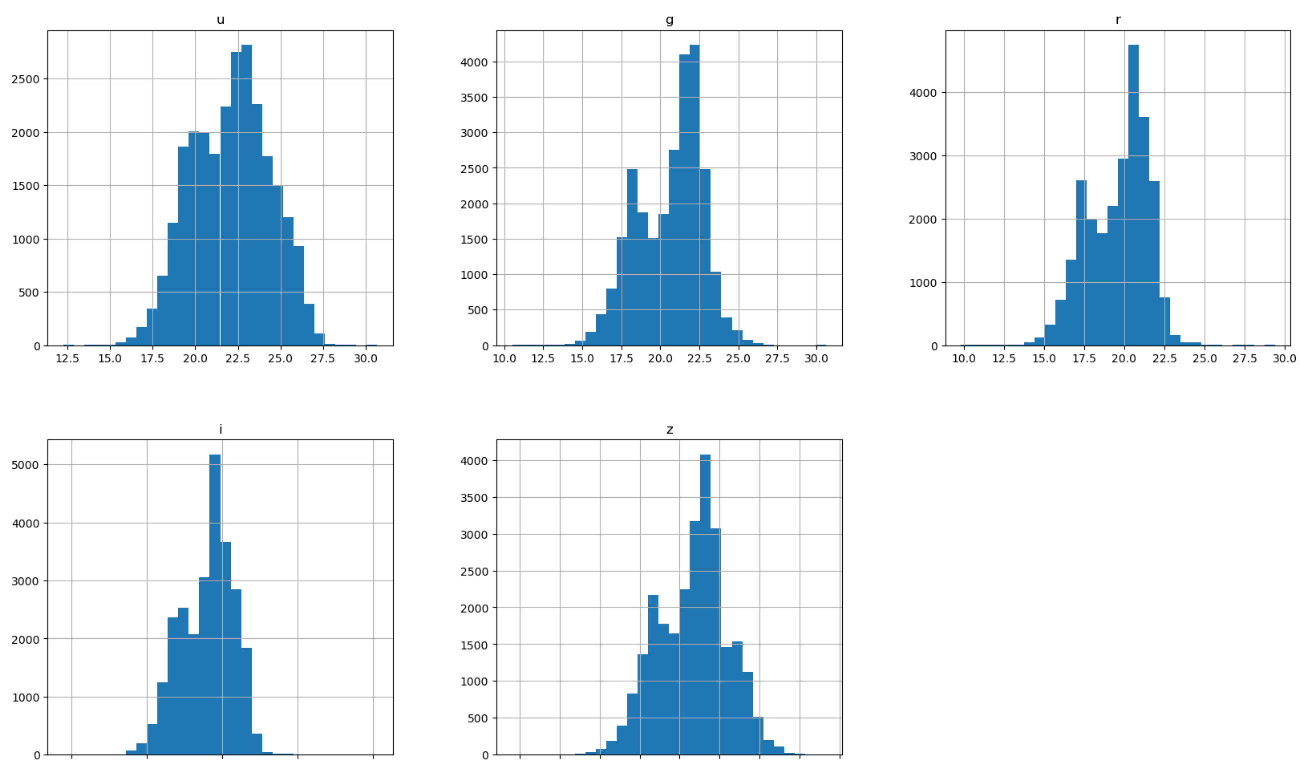
I was able to remove outliers by applying a log transformation in the redshift which left me with a more uniformly distrbuted distribution than the original redshift distribution, however, this distribution is still not normal, which means I will be scaling this distribution using min-max scaling.

Exploration into photometric values

In [22]:
```
df[['u','g','r','i','z']].plot.box();
```



In [23]:
```
df[['u','g','r','i','z']].hist(figsize=(20, 25), layout=(4,3), bins=30);
```



Some of these features have the same distribution shape and range, meaning they could possible be correlated, something that classifiers could pick on, will look at correlation matrices to prove this, and consider the feature as relevant

In [24]:
```python
df[['u','g','r','i','z']].corr()
```

Out[24]:

	u	g	r	i	z
u	1.000000	0.855146	0.738062	0.631404	0.556179
g	0.855146	1.000000	0.937479	0.856727	0.781739
r	0.738062	0.937479	1.000000	0.965055	0.920111
i	0.631404	0.856727	0.965055	1.000000	0.969735
z	0.556179	0.781739	0.920111	0.969735	1.000000

Observations

The photometric features appear to follow a pseudo-normal distribution. Although there exist outliers in this dataset and the range is fixed for this dataset (~10 - 30) we will standardize the distribution using z-score normalization because I believe that the outliers in these distributions still might hold valuable data and we cannot just clamp these distributions. Additionally some values such as i and z are extremely well correlated, which might be something to consider for classification.

Standarization of data

Because a large variety of these features excluding outliers seem normally (or semi normally distributed), we are going to use z-score normalization for photometric values U, G, R, I, Z, and for redshift values because of their distrbution, we are going to use min-max normalization.

In [25]:
```python
# save a copy of the df without scaling
df_without_scaling = df
```

In [26]:
```python
#create a dictionary to hold the scalers for each column to be scaled
scalers = {
    'alpha': MinMaxScaler(),
    'delta': MinMaxScaler(),
    'u': StandardScaler(),
    'g': StandardScaler(),
    'r': StandardScaler(),
    'i': StandardScaler(),
    'z': StandardScaler(),
    'redshift': MinMaxScaler(),
    'plate': MinMaxScaler(),
    'fiber_ID': MinMaxScaler()
}
```

```python
classes = df.pop('class')
for column in df.columns:
    df.loc[:,column] = scalers[column].fit_transform(df[[column]])
```

we are going to save the scalers we used in case we want to unscale the data back to its original form in a dictionary with the key being the column name

In [27]:
```python
# append the classes back
df.loc[:,'class'] = classes
```

In [28]:
```python
df
```

Out[28]:

	alpha	delta	u	g	r	i	z
0	0.963256	0.182992	-1.334728	-0.908000	-0.494452	-0.292998	-0.255032
1	0.599843	0.650501	0.753244	0.748867	0.714330	0.429296	0.256216
2	0.387181	0.523432	-1.213983	-0.736995	-0.269715	-0.059248	0.183340
3	0.684677	0.643591	-0.043638	0.500711	1.000351	1.433848	1.412552
4	0.324052	0.669314	0.058430	0.525786	1.003374	1.216853	1.188700
...	...	...	...	...	...	...	...
29994	0.062218	0.516598	1.426348	1.538015	1.103550	0.902282	0.654735
29995	0.354924	0.455543	1.779457	1.176506	0.833457	0.547017	0.391501
29996	0.664502	0.523679	0.106888	0.474032	0.216203	0.034538	-0.029804
29998	0.640316	0.681693	-1.636254	-1.652594	-1.533409	-1.415761	-1.287810
29999	0.621233	0.677615	0.538875	0.127651	0.330421	0.258034	0.603586

26059 rows × 11 columns

Choosing an Evaluation metric

This is an extremely large dataset, which makes leave-one-out cross validations computationally infeasible. Additionally, as highlighted in our data exploration, this dataset is very unbalanced. Therefore, we are going to use imbalanced evaluation meaures such as the Balance Accuracy Rate, a weighted F1 measure, precision, and recall, and a F1 measure to evaluate our classifier. This is the criterion and to further test our models robustness, we will use K-fold cross-validation using our F1 measure to evaluate the classifier.

I chose this because overall, we will be able to evaluate the relevance of retrieved results in a way that is guaranteed to test the robustness of our model due to the class imbalance in the data. Therefore having multiple measures such as our balance accuracy rate and our F1 measure will help us grasp how the models are performing despite the challenges on the dataset and areas they lack in the classification tasks.

In [29]:
```
y = df.pop('class')
X = df

X_train, X_test, y_train, y_test = train_test_split(X, y,test_size=0.30)
```

In [30]:
```
# Evaluation metrics
scoring = ('balanced_accuracy', 'f1_weighted', 'precision_weighted', 'recall
k = 5

# Setup K-Fold validation
kFold = KFold(n_splits = k, shuffle=True)
```

Training and finding the winning classifier

In [31]:
```
# Instantiating the models
decisionTree = DecisionTreeClassifier()
knn = KNeighborsClassifier()
SVMLinear = SVC(kernel='linear')
SVMPoly = SVC(kernel='poly')
SVM_RBF = SVC(kernel='rbf')
SVMSigmoid = SVC(kernel='sigmoid')
```

In [32]:
```
# Evaluate using Cross Validation

# cross validate decision tree
decisionTreeEvaluation = cross_validate(decisionTree, X, y, cv=kFold, scorin

# cross validate KNN
KNN_Evaluation = cross_validate(knn, X, y, cv=kFold, scoring=scoring, verbos

# cross validate SVM Linear
SVMLinearEvaluation = cross_validate(SVMLinear, X, y, cv=kFold, scoring=scor

# cross validate SVM Poly
SVMPolyEvaluation = cross_validate(SVMPoly, X, y, cv=kFold, scoring=scoring,

# cross validate SVM RBF
SVM_RBF_Evaluation = cross_validate(SVM_RBF, X, y, cv=kFold, scoring=scoring

# cross validate SVM Sigmoid
SVMSigmoidEvaluation = cross_validate(SVMSigmoid, X, y, cv=kFold, scoring=sc
```

```
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent workers.
[Parallel(n_jobs=-1)]: Done   2 out of   5 | elapsed:    1.5s remaining:    
2.3s
[Parallel(n_jobs=-1)]: Done   5 out of   5 | elapsed:    1.6s finished
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent workers.
[Parallel(n_jobs=-1)]: Done   2 out of   5 | elapsed:    0.3s remaining:    
0.5s
[Parallel(n_jobs=-1)]: Done   5 out of   5 | elapsed:    0.9s finished
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent workers.
[Parallel(n_jobs=-1)]: Done   2 out of   5 | elapsed:    4.0s remaining:    
6.0s
[Parallel(n_jobs=-1)]: Done   5 out of   5 | elapsed:    4.6s finished
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent workers.
[Parallel(n_jobs=-1)]: Done   2 out of   5 | elapsed:    6.8s remaining:   1
0.2s
[Parallel(n_jobs=-1)]: Done   5 out of   5 | elapsed:    7.5s finished
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent workers.
[Parallel(n_jobs=-1)]: Done   2 out of   5 | elapsed:    5.3s remaining:    
7.9s
[Parallel(n_jobs=-1)]: Done   5 out of   5 | elapsed:    5.8s finished
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent workers.
[Parallel(n_jobs=-1)]: Done   2 out of   5 | elapsed:   20.9s remaining:   3
1.4s
[Parallel(n_jobs=-1)]: Done   5 out of   5 | elapsed:   23.4s finished
```

In [33]:

```python
def printResultsEvaluation(evaluation: dict):
    for key in evaluation.keys():
        print(f'{key}: {round(evaluation[key].mean(),2)} +/- {round(np.std(e
```

In [34]:

```python
print(decisionTree)
printResultsEvaluation(decisionTreeEvaluation)
print()
print(knn)
printResultsEvaluation(KNN_Evaluation)
print()
print(SVMLinear)
printResultsEvaluation(SVMLinearEvaluation)
print()
print(SVMPoly)
printResultsEvaluation(SVMPolyEvaluation)
print()
print(SVM_RBF)
printResultsEvaluation(SVM_RBF_Evaluation)
print()
print(SVMSigmoid)
printResultsEvaluation(SVMSigmoidEvaluation)
```

```
DecisionTreeClassifier()
fit_time: 0.3 +/- 0.0055
score_time: 0.08 +/- 0.0017
test_balanced_accuracy: 0.95 +/- 0.0022
test_f1_weighted: 0.96 +/- 0.0016
test_precision_weighted: 0.96 +/- 0.0016
test_recall_weighted: 0.96 +/- 0.0017

KNeighborsClassifier()
fit_time: 0.02 +/- 0.0018
score_time: 0.24 +/- 0.0269
test_balanced_accuracy: 0.93 +/- 0.0027
test_f1_weighted: 0.95 +/- 0.0027
test_precision_weighted: 0.95 +/- 0.0027
test_recall_weighted: 0.95 +/- 0.0027

SVC(kernel='linear')
fit_time: 3.41 +/- 0.2552
score_time: 0.67 +/- 0.0297
test_balanced_accuracy: 0.95 +/- 0.0044
test_f1_weighted: 0.97 +/- 0.0028
test_precision_weighted: 0.97 +/- 0.0028
test_recall_weighted: 0.97 +/- 0.0027

SVC(kernel='poly')
fit_time: 5.81 +/- 0.43
score_time: 1.11 +/- 0.0762
test_balanced_accuracy: 0.94 +/- 0.0031
test_f1_weighted: 0.96 +/- 0.0018
test_precision_weighted: 0.96 +/- 0.0017
test_recall_weighted: 0.96 +/- 0.0018

SVC()
fit_time: 4.07 +/- 0.2955
score_time: 1.38 +/- 0.0501
test_balanced_accuracy: 0.95 +/- 0.0031
test_f1_weighted: 0.96 +/- 0.0025
test_precision_weighted: 0.97 +/- 0.0024
test_recall_weighted: 0.97 +/- 0.0025

SVC(kernel='sigmoid')
fit_time: 18.27 +/- 1.1574
score_time: 3.46 +/- 0.2257
test_balanced_accuracy: 0.51 +/- 0.0258
test_f1_weighted: 0.62 +/- 0.0113
test_precision_weighted: 0.64 +/- 0.0148
test_recall_weighted: 0.65 +/- 0.0179
```

Discussion on performance of models

The winning classifier after many rounds of playing with the parameters is the linear SVM and the RBF SVM, suggesting that some features/patterns in the data should be linearly separable, or at least in the same radial basis. Although a lot of the models came close to this performance benchmark (basically 97% on all measures). Something that surprised me the most was the training times for the polynomial SVM, making it a bad time/performance tradeoff, as the linear SVM trains much faster and is slightly more accurate. Additionally, the Sigmoid Kernel SVM performed poorly as it was not able to capture the patterns in the data too accurately.

Overall the performance of the linear SVM was expected as by a high-relevance feature such as redshift seem mostly linearly separable. The decision tree would probably pick up on this pattern too as it also performed quite well, probably because of the linear separability of redshift, making it an easy split rule inside the tree.

Side note

I realized that it is taking a lot of time to run these SVMs, especially the polynomial kernel SVM, therefore, I am going to look at the learning curves for all of the models and estimate about how many traning samples are needed to generate a 'generalizable' result.

In [35]:
```
## ***** Note to Grader ******
## Most of this code is modified and sourced from SciKit-Learn's Documentati
#Link: https://scikit-learn.org/dev/modules/generated/sklearn.model_selectio

from sklearn.model_selection import LearningCurveDisplay

fig, ax = plt.subplots(nrows=2, ncols=3, figsize=(10, 6), sharey=False)

common_params = {
    "X": X,
    "y": y,
    "train_sizes": np.linspace(0.1, 1.0, 10),
    "cv": kFold,
    "n_jobs": -1,
    "line_kw": {"marker": "o"},
    "std_display_style": "fill_between",
    "scoring": "f1_weighted",
}

axes = ax.flatten()

for idx, estimator in enumerate([decisionTree, knn, SVMLinear, SVMPoly,SVM_R
    LearningCurveDisplay.from_estimator(estimator, **common_params, ax=axes[
    handles, label = axes[idx].get_legend_handles_labels()
    axes[idx].legend(handles[:2], ["Training Score", "Test Score"])
    axes[idx].set_title(f"Learning Curve for {str(estimator)}")

plt.tight_layout()
plt.show()
```

```
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent workers.
[learning_curve] Training set sizes: [ 2084  4169  6254  8338 10423 12508 14
592 16677 18762 20847]
[Parallel(n_jobs=-1)]: Done  25 out of  50 | elapsed:    1.0s remaining:
1.0s
[Parallel(n_jobs=-1)]: Done  50 out of  50 | elapsed:    2.1s finished
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent workers.
[learning_curve] Training set sizes: [ 2084  4169  6254  8338 10423 12508 14
592 16677 18762 20847]
[Parallel(n_jobs=-1)]: Done  25 out of  50 | elapsed:    3.3s remaining:
3.3s
[Parallel(n_jobs=-1)]: Done  50 out of  50 | elapsed:    6.4s finished
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent workers.
[learning_curve] Training set sizes: [ 2084  4169  6254  8338 10423 12508 14
592 16677 18762 20847]
[Parallel(n_jobs=-1)]: Done  25 out of  50 | elapsed:   13.3s remaining:   1
3.3s
[Parallel(n_jobs=-1)]: Done  50 out of  50 | elapsed:   31.5s finished
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent workers.
[learning_curve] Training set sizes: [ 2084  4169  6254  8338 10423 12508 14
592 16677 18762 20847]
[Parallel(n_jobs=-1)]: Done  25 out of  50 | elapsed:   28.2s remaining:   2
8.2s
[Parallel(n_jobs=-1)]: Done  50 out of  50 | elapsed:  1.1min finished
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent workers.
[learning_curve] Training set sizes: [ 2084  4169  6254  8338 10423 12508 14
592 16677 18762 20847]
[Parallel(n_jobs=-1)]: Done  25 out of  50 | elapsed:   43.1s remaining:   4
3.1s
[Parallel(n_jobs=-1)]: Done  50 out of  50 | elapsed:  1.5min finished
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent workers.
[learning_curve] Training set sizes: [ 2084  4169  6254  8338 10423 12508 14
592 16677 18762 20847]
[Parallel(n_jobs=-1)]: Done  25 out of  50 | elapsed:  1.3min remaining:  1.
3min
[Parallel(n_jobs=-1)]: Done  50 out of  50 | elapsed:  3.1min finished
```

From this data, we can see that any where from 1000 to 5000 samples yield the best results for generalizable results, this is the number I will use

Filter Technique

In [36]:
```
X_sample_train, _, y_sample_train, _ = train_test_split(X, y, train_size=100
```

In [37]:
```
mi = {}

i_scores = mutual_info_classif(X_train, y_train)

for i, j in zip(X_train.columns, i_scores):
    mi[i] = j

columnInfoGaindf= pd.DataFrame.from_dict(mi, orient='index', columns=['I-Gai
columnInfoGaindf.sort_values(by=['I-Gain'],ascending=False,inplace=True)
```

In [38]:
```
columnInfoGaindf
```

Out[38]:

	I-Gain
redshift	0.756098
plate	0.251924
z	0.115435
u	0.105711
g	0.103883
i	0.084999
r	0.061213
alpha	0.045432
fiber_ID	0.042707
delta	0.040229

Running SVM Classifiers with the top three features

In [39]:

```
topThreeFeatures = columnInfoGaindf[0:3]

topThreeDf = df[topThreeFeatures.index]
```

In [40]:

```
#cross validate SVM Linear
topThreeLinearEval = cross_validate(SVMLinear, topThreeDf, y, cv=kFold, scor

# cross validate SVM Poly
topThreePolyEval = cross_validate(SVMPoly, topThreeDf, y, cv=kFold, scoring=

# cross validate SVM RBF
topThree_RBF_Eval = cross_validate(SVM_RBF, topThreeDf, y, cv=kFold, scoring

# cross validate SVM Sigmoid
topThreeSigmoidEval = cross_validate(SVMSigmoid, topThreeDf, y, cv=kFold, sc
```

```
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent workers.
[Parallel(n_jobs=-1)]: Done   2 out of   5 | elapsed:   11.1s remaining:   1
6.6s
[Parallel(n_jobs=-1)]: Done   5 out of   5 | elapsed:   12.6s finished
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent workers.
[Parallel(n_jobs=-1)]: Done   2 out of   5 | elapsed:   16.8s remaining:   2
5.2s
[Parallel(n_jobs=-1)]: Done   5 out of   5 | elapsed:   18.5s finished
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent workers.
[Parallel(n_jobs=-1)]: Done   2 out of   5 | elapsed:   11.7s remaining:   1
7.5s
[Parallel(n_jobs=-1)]: Done   5 out of   5 | elapsed:   12.6s finished
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent workers.
[Parallel(n_jobs=-1)]: Done   2 out of   5 | elapsed:   35.5s remaining:   5
3.2s
[Parallel(n_jobs=-1)]: Done   5 out of   5 | elapsed:   38.0s finished
```

In [41]:

```
print(SVMLinear)
printResultsEvaluation(topThreeLinearEval)
print()
print(SVMPoly)
printResultsEvaluation(topThreePolyEval)
print()
print(SVM_RBF)
printResultsEvaluation(topThree_RBF_Eval)
print()
print(SVMSigmoid)
printResultsEvaluation(topThreeSigmoidEval)
```

```
SVC(kernel='linear')
fit_time: 9.41 +/- 0.898
score_time: 2.22 +/- 0.1421
test_balanced_accuracy: 0.9 +/- 0.0078
test_f1_weighted: 0.94 +/- 0.0038
test_precision_weighted: 0.95 +/- 0.0028
test_recall_weighted: 0.95 +/- 0.0032

SVC(kernel='poly')
fit_time: 15.28 +/- 1.0837
score_time: 2.18 +/- 0.2639
test_balanced_accuracy: 0.9 +/- 0.0039
test_f1_weighted: 0.94 +/- 0.003
test_precision_weighted: 0.94 +/- 0.003
test_recall_weighted: 0.94 +/- 0.0028

SVC()
fit_time: 8.86 +/- 0.6108
score_time: 3.12 +/- 0.0975
test_balanced_accuracy: 0.91 +/- 0.0024
test_f1_weighted: 0.95 +/- 0.0017
test_precision_weighted: 0.95 +/- 0.0017
test_recall_weighted: 0.95 +/- 0.0017

SVC(kernel='sigmoid')
fit_time: 28.61 +/- 2.8965
score_time: 6.96 +/- 0.3471
test_balanced_accuracy: 0.36 +/- 0.0312
test_f1_weighted: 0.51 +/- 0.0308
test_precision_weighted: 0.54 +/- 0.0537
test_recall_weighted: 0.53 +/- 0.0522
```

Running SVM Classifiers with the bottom three features

In [42]:

```
bottomThreeFeatures = columnInfoGaindf[-3:]

bottomThreeDf = df[bottomThreeFeatures.index]
bottomThreeDfSample = X_sample_train[bottomThreeFeatures.index]

# cross validate SVM Linear
bottomThreeLinearEval = cross_validate(SVMLinear, bottomThreeDf, y, cv=kFolc

# cross validate SVM Poly
bottomThreePolyEval = cross_validate(SVMPoly, bottomThreeDfSample, y_sample_

# cross validate SVM RBF
bottomThree_RBF_Eval = cross_validate(SVM_RBF, bottomThreeDf, y, cv=kFold, s

# cross validate SVM Sigmoid
bottomThreeSigmoidEval = cross_validate(SVMSigmoid, bottomThreeDf, y, cv=kFc
```

```
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent workers.
[Parallel(n_jobs=-1)]: Done   2 out of   5 | elapsed:   44.4s remaining:  1.
1min
[Parallel(n_jobs=-1)]: Done   5 out of   5 | elapsed:   48.6s finished
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent workers.
[Parallel(n_jobs=-1)]: Done   2 out of   5 | elapsed:    0.1s remaining:    
0.2s
[Parallel(n_jobs=-1)]: Done   5 out of   5 | elapsed:    0.2s finished
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent workers.
[Parallel(n_jobs=-1)]: Done   2 out of   5 | elapsed:  1.2min remaining:  1.
8min
[Parallel(n_jobs=-1)]: Done   5 out of   5 | elapsed:  1.2min finished
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent workers.
[Parallel(n_jobs=-1)]: Done   2 out of   5 | elapsed:   38.5s remaining:   5
7.7s
[Parallel(n_jobs=-1)]: Done   5 out of   5 | elapsed:   41.1s finished
```

In [43]:

```
print(SVMLinear)
printResultsEvaluation(bottomThreeLinearEval)
print()
print(SVMPoly)
printResultsEvaluation(bottomThreePolyEval)
print()
print(SVM_RBF)
printResultsEvaluation(bottomThree_RBF_Eval)
print()
print(SVMSigmoid)
printResultsEvaluation(bottomThreeSigmoidEval)
```

```
SVC(kernel='linear')
fit_time: 39.76 +/- 2.9728
score_time: 6.17 +/- 0.9173
test_score: 0.48 +/- 0.0081

SVC(kernel='poly')
fit_time: 0.1 +/- 0.0202
score_time: 0.01 +/- 0.0022
test_score: 0.48 +/- 0.0424

SVC()
fit_time: 59.0 +/- 0.6537
score_time: 13.1 +/- 0.2782
test_score: 0.49 +/- 0.0033

SVC(kernel='sigmoid')
fit_time: 32.55 +/- 1.5061
score_time: 6.67 +/- 0.5076
test_score: 0.47 +/- 0.0203
```

Discussion

The results of running the SVMS with the top three most discriminative features and the whole dataset were almost as expected; we were able to capture a lot of the information as the f1-measure gave 95% compared to 97% on the whole dataset. This is surprising because not only did these models take less time to train, but they also managed to be almost as efficient.

While training the models on the bottom three features, results were also as expected as most of the models did not even achieve an f1-measure of 0.5 meaning they almost did not outperform random classification. Additiontally, something that surprised me was how slow the fitting times were for the polynomial kernel, as it took almost 3 hours for it to run with these features, therefore I had to sample the dataset and make it run with this sample and generalize the results. This sample was taken from the learning curve of this data vs quantity of training samples.

Wrapper Techniques

In [44]:
```
# Using Sequential Forward Selection with each model and 5-fold cross-valida
# look discriminative features per SVM

# define the SFS instances
sfs_Forward_Decision_Tree = SFS(decisionTree, forward=True,k_features=3, sco
sfs_Forward_KNN = SFS(knn, forward=True,k_features=3, scoring='f1_weighted',
sfs_SVM_Linear = SFS(SVMLinear, forward=True,k_features=3, scoring='f1_weigh
sfs_SVM_Poly = SFS(SVMPoly, forward=True,k_features=3, scoring='f1_weighted'
sfs_SVM_RBF = SFS(SVM_RBF, forward=True,k_features=3, scoring='f1_weighted',
sfs_SVM_Sigmoid = SFS(SVMSigmoid, forward=True,k_features=3, scoring='f1_wei
```

In [45]:
```
%%time
sfs_Forward_Decision_Tree.fit(X,y)
sfs_Forward_KNN.fit(X,y)
sfs_SVM_Linear.fit(X,y)
sfs_SVM_Poly.fit(X_sample_train,y_sample_train)
sfs_SVM_RBF.fit(X_sample_train,y_sample_train)
sfs_SVM_Sigmoid.fit(X_sample_train,y_sample_train)
```

```
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent workers.
/opt/anaconda3/envs/dataMining/lib/python3.9/site-packages/sklearn/model_sel
ection/_validation.py:73: FutureWarning: `fit_params` is deprecated and will
be removed in version 1.6. Pass parameters via `params` instead.
  warnings.warn(
/opt/anaconda3/envs/dataMining/lib/python3.9/site-packages/sklearn/model_sel
ection/_validation.py:73: FutureWarning: `fit_params` is deprecated and will
be removed in version 1.6. Pass parameters via `params` instead.
  warnings.warn(
/opt/anaconda3/envs/dataMining/lib/python3.9/site-packages/sklearn/model_sel
ection/_validation.py:73: FutureWarning: `fit_params` is deprecated and will
be removed in version 1.6. Pass parameters via `params` instead.
  warnings.warn(
/opt/anaconda3/envs/dataMining/lib/python3.9/site-packages/sklearn/model_sel
```

```
be removed in version 1.6. Pass parameters via `params` instead.
  warnings.warn(
/opt/anaconda3/envs/dataMining/lib/python3.9/site-packages/sklearn/model_sel
ection/_validation.py:73: FutureWarning: `fit_params` is deprecated and will
be removed in version 1.6. Pass parameters via `params` instead.
  warnings.warn(
CPU times: user 1.45 s, sys: 562 ms, total: 2.01 s
Wall time: 9min 48s
[Parallel(n_jobs=-1)]: Done   3 out of   8 | elapsed:    0.3s remaining:
0.6s
[Parallel(n_jobs=-1)]: Done   8 out of   8 | elapsed:    0.4s finished

[2024-10-31 18:37:42] Features: 3/3 -- score: 0.6721743945939638
```

Out[45]:

▸ **SequentialFeatureSelector**

▸ **estimator: SVC**

▸ SVC

In [46]:

```python
def printSFS_Results(sfsResult):
    print(f'Estimator: {str(sfsResult.estimator)}')
    print(f'Top 3 features: {sfsResult.k_feature_names_}')
    print(f'F1 measure of model with top 3 features: {sfsResult.k_score_}')
    print()
```

In [47]:

```python
printSFS_Results(sfs_Forward_Decision_Tree)
printSFS_Results(sfs_Forward_KNN)
printSFS_Results(sfs_SVM_Linear)
printSFS_Results(sfs_SVM_Poly)
printSFS_Results(sfs_SVM_RBF)
printSFS_Results(sfs_SVM_Sigmoid)
```

```
Estimator: DecisionTreeClassifier()
Top 3 features: ('g', 'i', 'redshift')
F1 measure of model with top 3 features: 0.9641583472902701

Estimator: KNeighborsClassifier()
Top 3 features: ('u', 'redshift', 'plate')
F1 measure of model with top 3 features: 0.9626760593211084

Estimator: SVC(kernel='linear')
Top 3 features: ('g', 'i', 'redshift')
F1 measure of model with top 3 features: 0.9662490242892936

Estimator: SVC(kernel='poly')
Top 3 features: ('alpha', 'u', 'redshift')
F1 measure of model with top 3 features: 0.9396693869842577

Estimator: SVC()
Top 3 features: ('delta', 'u', 'redshift')
F1 measure of model with top 3 features: 0.9416980987867076

Estimator: SVC(kernel='sigmoid')
Top 3 features: ('u', 'i', 'redshift')
F1 measure of model with top 3 features: 0.6721743945939638
```

Discussion

When using the filter technique, we got the result that the top three discriminative features are ['redshift','plate','z'], which is what we expected, as redshift is almost linearly separable and SVMs could take advantage of that. However, when we used the wrapper technique, the results for the three most discriminative features started to vary. Not to my surprise, all of the feature subsets with the wrapper technique included the redshift feature, indicating that it is a relevant feature for all models, most SVMS with high f1-measure included a plate, which is what we also see with the filter technique. However, then every SVM and other estimator looks at relevance with photometric values such as 'u' (count 5/6) 'g' (count 1/6), or 'z' (count 1/6). This might appeal to the individual kernels of the SVMs and how they learn from data. This behavior was also expected from using a Wrapper Technique.

Some differences indicate that in the filter technique 'u' was not shown to be a discriminative feature, however, most SVMs and estimators used it well in conjunction with the other two included in the filter subset.

Task 7 Discussion

When we look at the performance of the different classifiers from task 4, and the classifier's performance on tasks 5 and 6, results are as expected. When performing feature selection, you have can expect to have a dimensionality-accuracy tradeoff. From this tradeoff, you are trying to maximize your accuracy while minimizing your dimensionality. This is why when we used feature selection with the filter approach and with the wrapper approach, we get to see a decrease of about 0.01 in our f1 measure, which is expected, however, given that we went from 9 features to 3 features, and managed only to lose 0.01 on our f1 measure, this tradeoff is worth it. Especially when models take a long time to train like SVMs in this case.

The decrease in accuracy after feature selection also might signal that the other features also play a role in the models' learning accuracy, which might signify that most features in the dataset are relevant and that models can learn from them.

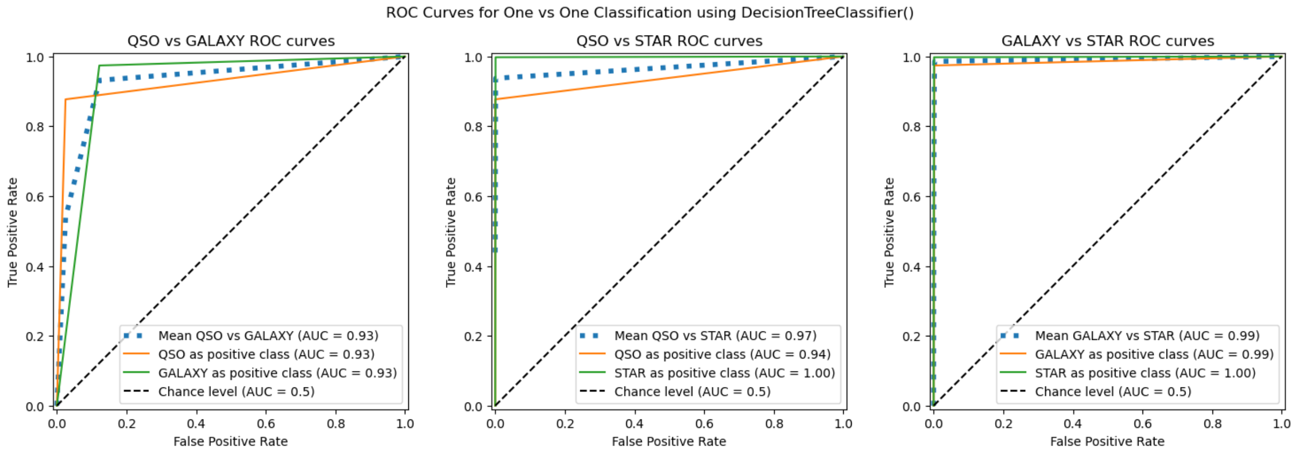
ROC Curves

In [48]:
```
pair_list = list(combinations(y.unique(),2))
label_binarizer = LabelBinarizer().fit(y_train)
y_onehot_test = label_binarizer.transform(y_test)
y_onehot_test.shape  # (n_samples, n_classes)
target_names = ['GALAXY','QSO','STAR']
```

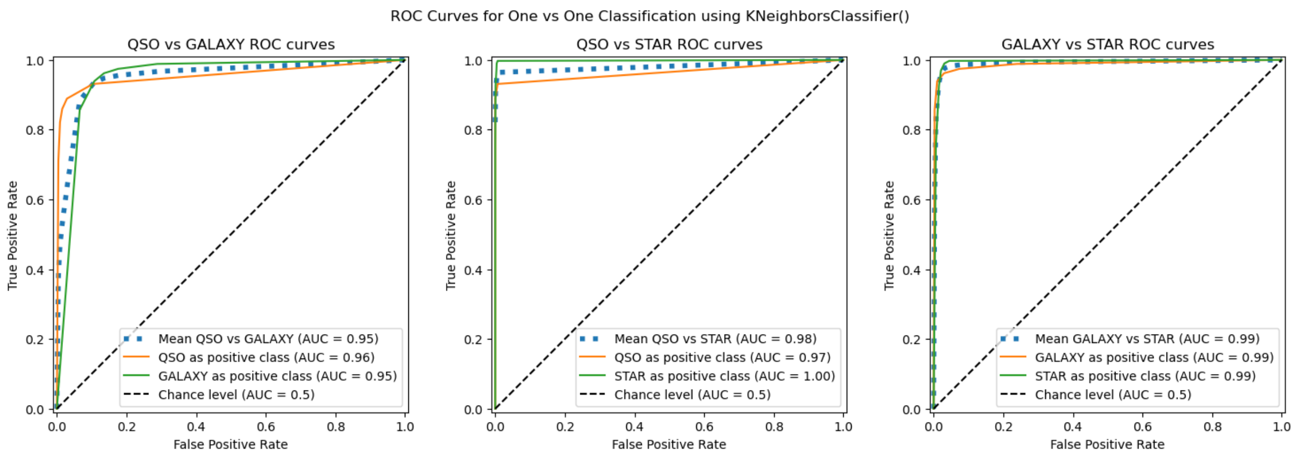
In [49]:
```
## ***** Note to Grader ******
## Most of this code is modified and sourced from SciKit-Learn's Documentati
## Link: https://scikit-learn.org/dev/auto_examples/model_selection/plot_roc

def plotROCCurve(classifier):
    y_score = classifier.fit(X_train, y_train).predict_proba(X_test)

    fpr_grid = np.linspace(0.0, 1.0, 1000)

    pair_scores = []
    mean_tpr = dict()

    fig, axs = plt.subplots(1, 3, figsize=(15, 5))

    for ix, (label_a, label_b) in enumerate(pair_list):
        a_mask = y_test == label_a
        b_mask = y_test == label_b
        ab_mask = np.logical_or(a_mask, b_mask)

        a_true = a_mask[ab_mask]
        b_true = b_mask[ab_mask]

        idx_a = np.flatnonzero(label_binarizer.classes_ == label_a)[0]
        idx_b = np.flatnonzero(label_binarizer.classes_ == label_b)[0]

        fpr_a, tpr_a, _ = roc_curve(a_true, y_score[ab_mask, idx_a])
        fpr_b, tpr_b, _ = roc_curve(b_true, y_score[ab_mask, idx_b])
```

```python
        mean_tpr[ix] = np.zeros_like(fpr_grid)
        mean_tpr[ix] += np.interp(fpr_grid, fpr_a, tpr_a)
        mean_tpr[ix] += np.interp(fpr_grid, fpr_b, tpr_b)
        mean_tpr[ix] /= 2
        mean_score = auc(fpr_grid, mean_tpr[ix])
        pair_scores.append(mean_score)

        ax = axs[ix]
        ax.plot(
            fpr_grid,
            mean_tpr[ix],
            label=f"Mean {label_a} vs {label_b} (AUC = {mean_score:.2f})",
            linestyle=":",
            linewidth=4,
        )
        RocCurveDisplay.from_predictions(
            a_true,
            y_score[ab_mask, idx_a],
            ax=ax,
            name=f"{label_a} as positive class",
        )
        RocCurveDisplay.from_predictions(
            b_true,
            y_score[ab_mask, idx_b],
            ax=ax,
            name=f"{label_b} as positive class",
            plot_chance_level=True,
        )
        ax.set(
            xlabel="False Positive Rate",
            ylabel="True Positive Rate",
            title=f"{label_a} vs {label_b} ROC curves",
        )
        ax.legend(loc="lower right")

    fig.suptitle('ROC Curves for One vs One Classification using ' + str(cla
    plt.tight_layout()
    plt.show()

    print(f"Macro-averaged One-vs-One ROC AUC score:\n{np.average(pair_score
```

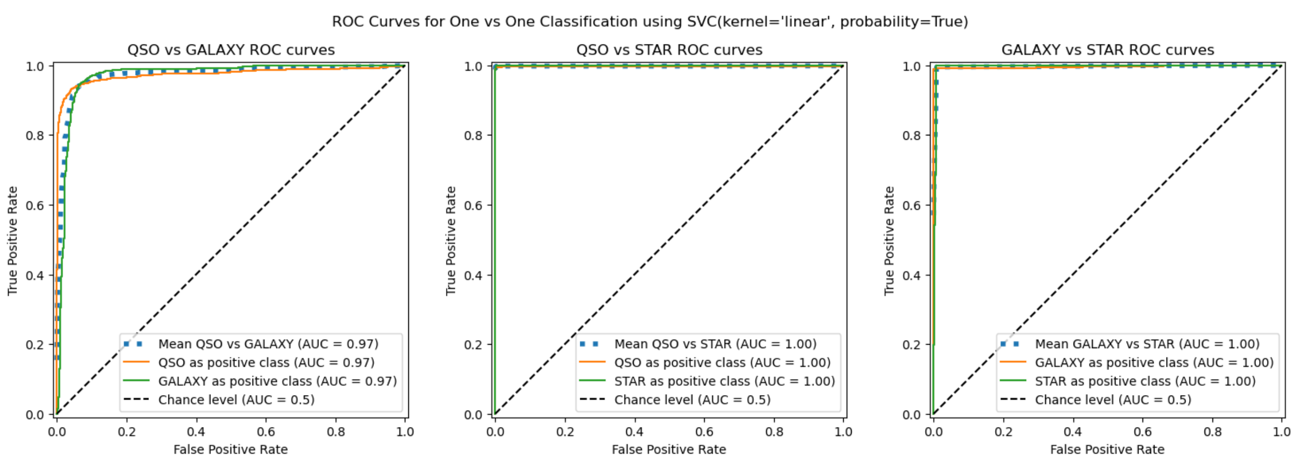
In [50]:
```python
plotROCCurve(decisionTree)
plotROCCurve(knn)
plotROCCurve(SVMLinear.set_params(probability=True))
plotROCCurve(SVMPoly.set_params(probability=True))
plotROCCurve(SVM_RBF.set_params(probability=True))
plotROCCurve(SVMSigmoid.set_params(probability=True))
```



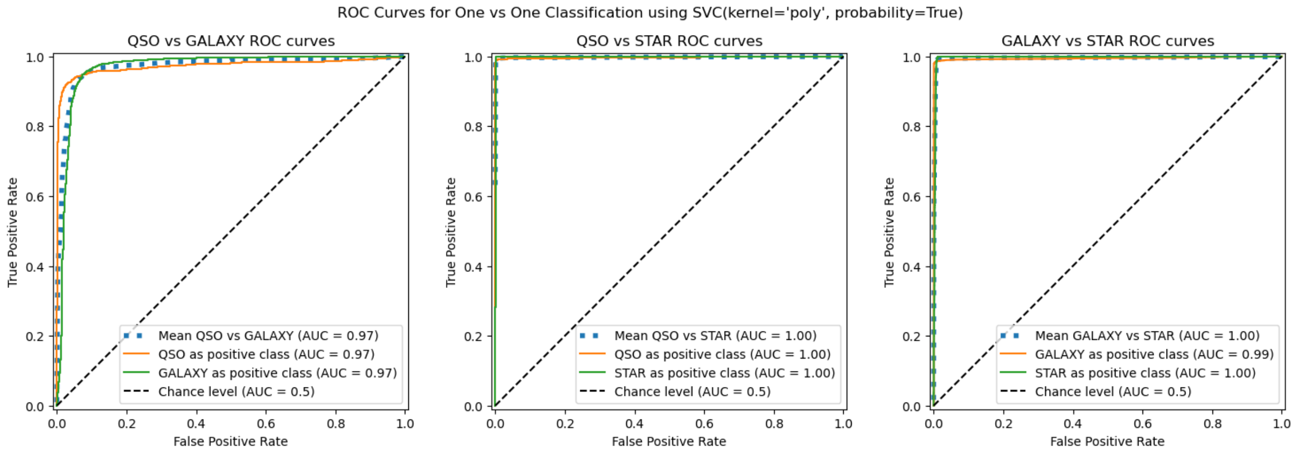
Macro-averaged One-vs-One ROC AUC score:
0.96



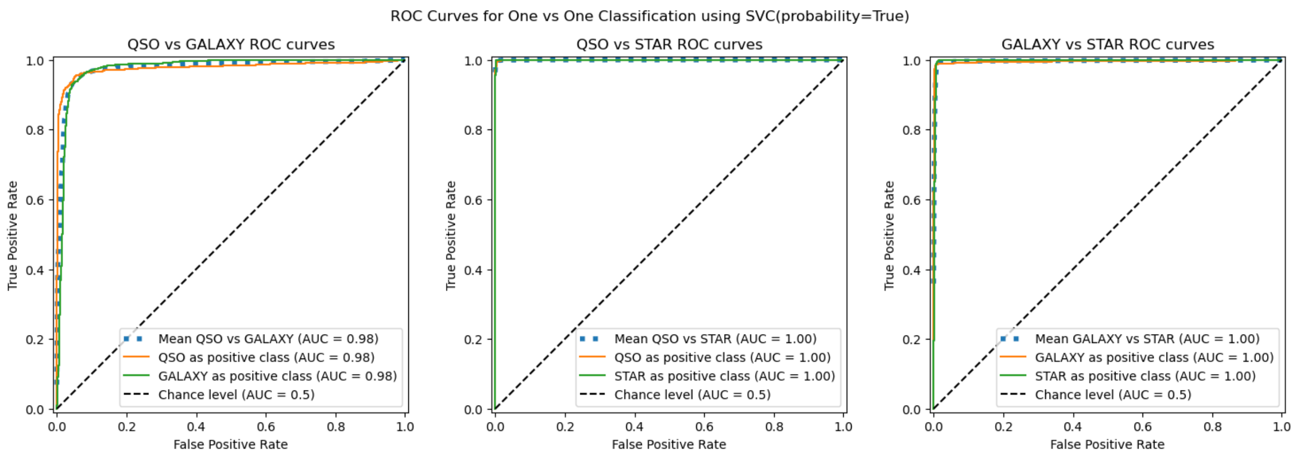
Macro-averaged One-vs-One ROC AUC score:
0.97



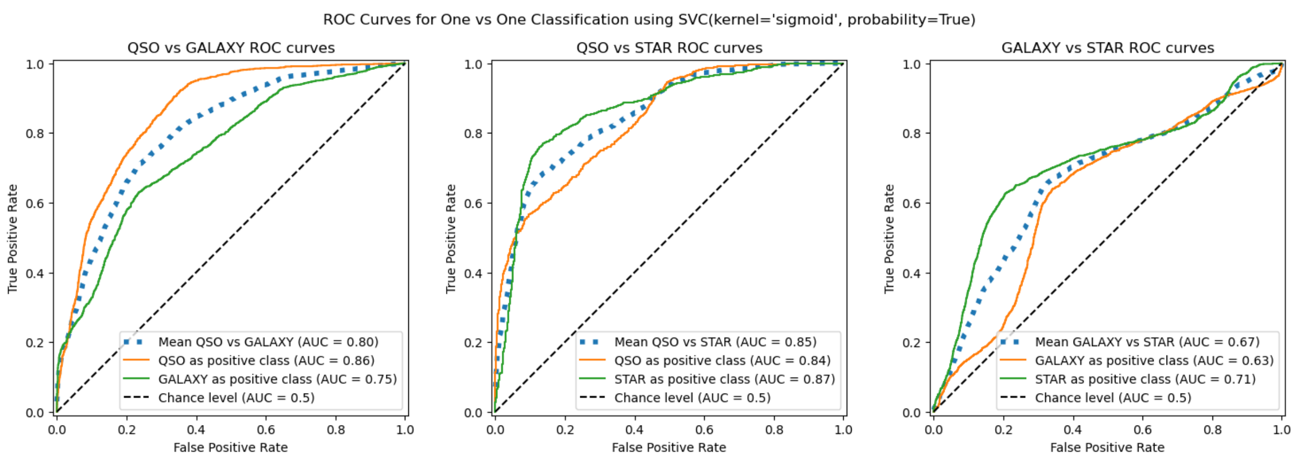
Macro-averaged One-vs-One ROC AUC score:
0.99




```
Macro-averaged One-vs-One ROC AUC score:
0.99
```




```
Macro-averaged One-vs-One ROC AUC score:
0.99
```




```
Macro-averaged One-vs-One ROC AUC score:
0.77
```

Discussion

The ROC curves for the three classes reveal something quite interesting: almost all of the models are great at correctly distinguishing QSOs from Stars and Galaxies from stars with almost 100% accuracy (This might be due to the linear separability of these two classes). However, when looking at the less linearly separable classes according to our redshift graph the areas where redshift tends to blend between QSO classification and galaxy classification, that is where most of the error for our classifiers come in.

Additionally, from the ROC curves, we can interpret that the linear SVM and the RBF Classifiers are the best models for all classification tasks for this dataset averaging almost 0.99 of the Area Under the Curve Measure. I am very satisfied with the performance of these classifiers, as they are correctly classifying about 97% of the time. To take research further, I would recommend looking at more significant features similar to redshift which are capable of distinguishing QSOs and Galaxies in a linearly separable way similar to how redshift can distinguish Stars from QSOs and Galaxies in a linearly separable way.

Correlations

I looked at the correlation values given by the wrapper technique and found that there are some very good correlations between features, as I mentioned in that part of the data exploration. This reveals that further feature engineering, in order to differentiate QSOs from Galaxies, can be done to further increase the accuracy of the models. This might be able to explain why all of the features performed better than using the top 3, as 3 top features are good, but maybe combinations of 3 photometric values are how the model explains QSOs vs Galaxies. Overall, something that surprised me was that the default for the SVM models performs best, as I tried to set the C value high and low but had no success, and performance just seemed to degrade. Overall, if I had more time, I would definitely look into the association with all the features to QSOs and Galaxies and how I might be able to find more feature combinations that explain their difference to improve classification accuracy. Overall I am very happy with this project and enjoyed solving it.