Import libraries

In [1]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
%matplotlib inline

import warnings
warnings.filterwarnings('ignore')
```

Import dataset

In [2]:
```
df = pd.read_csv("dataset.csv")
```

Exploratory data analysis

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

Out[3]:
```
(17898, 9)
```

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

Out[4]:

	Mean of the integrated profile	Standard deviation of the integrated profile	Excess kurtosis of the integrated profile	Skewness of the integrated profile	Mean of the DM-SNR curve	Standard deviation of the DM-SNR curve	Excess kurtosis of the DM-SNR curve	Skewness of the DM-SNR curve	tar
0	140.562500	55.683782	-0.234571	-0.699648	3.199833	19.110426	7.975532	74.242225	
1	102.507812	58.882430	0.465318	-0.515088	1.677258	14.860146	10.576487	127.393580	
2	103.015625	39.341649	0.323328	1.051164	3.121237	21.744669	7.735822	63.171909	
3	136.750000	57.178449	-0.068415	-0.636238	3.642977	20.959280	6.896499	53.593661	
4	88.726562	40.672225	0.600866	1.123492	1.178930	11.468720	14.269573	252.567306	

Now, We will view the column names to check for leading and trailing spaces.

In [5]:
```
col_names = df.columns
col_names
```

Out[5]:
```
Index([' Mean of the integrated profile',
       ' Standard deviation of the integrated profile',
       ' Excess kurtosis of the integrated profile',
       ' Skewness of the integrated profile', ' Mean of the DM-SNR curve',
       ' Standard deviation of the DM-SNR curve',
       ' Excess kurtosis of the DM-SNR curve', ' Skewness of the DM-SNR curve',
       'target_class'],
      dtype='object')
```

We can see that there are leading spaces (spaces at the start of the string name) in the dataframe. So, We will remove these leading spaces.

In [6]:
```
df.columns = df.columns.str.strip()
df.columns
```

Out[6]:
```
Index(['Mean of the integrated profile',
       'Standard deviation of the integrated profile',
       'Excess kurtosis of the integrated profile',
       'Skewness of the integrated profile', 'Mean of the DM-SNR curve',
       'Standard deviation of the DM-SNR curve',
       'Excess kurtosis of the DM-SNR curve', 'Skewness of the DM-SNR curve',
       'target_class'],
      dtype='object')
```

We can see that the leading spaces are removed from the column name. But the column names are very long. So, We will make them short by renaming them.

In [7]:
```
df.columns = ['IP Mean', 'IP Sd', 'IP Kurtosis', 'IP Skewness',
              'DM-SNR Mean', 'DM-SNR Sd', 'DM-SNR Kurtosis', 'DM-SNR Skewness', 'ta

df.columns
```

Out[7]:
```
Index(['IP Mean', 'IP Sd', 'IP Kurtosis', 'IP Skewness', 'DM-SNR Mean',
       'DM-SNR Sd', 'DM-SNR Kurtosis', 'DM-SNR Skewness', 'target_class'],
      dtype='object')
```

Our target variable is the target_class column. So, We will check its distribution.

In [8]:
```
df['target_class'].value_counts()
```

Out[8]:
```
0    16259
1     1639
Name: target_class, dtype: int64
```

In [9]:
```
df['target_class'].value_counts()/np.float(len(df))
```

Out[9]:
```
0    0.908426
1    0.091574
Name: target_class, dtype: float64
```

We can see that percentage of observations of the class label 0 and 1 is 90.84% and 9.16%. So, this is a class imbalanced problem. We will deal with that in later section.

In [10]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 17898 entries, 0 to 17897
Data columns (total 9 columns):
 #   Column            Non-Null Count  Dtype  
---  ------            --------------  -----  
 0   IP Mean           17898 non-null  float64
 1   IP Sd             17898 non-null  float64
 2   IP Kurtosis       17898 non-null  float64
 3   IP Skewness       17898 non-null  float64
 4   DM-SNR Mean       17898 non-null  float64
 5   DM-SNR Sd         17898 non-null  float64
 6   DM-SNR Kurtosis   17898 non-null  float64
 7   DM-SNR Skewness   17898 non-null  float64
 8   target_class      17898 non-null  int64  
dtypes: float64(8), int64(1)
memory usage: 1.2 MB
```

We can see that there are no missing values in the dataset and all the variables are numerical variables.

Outliers in numerical variables

In [11]:
```
round(df.describe(),2)
```

Out[11]:

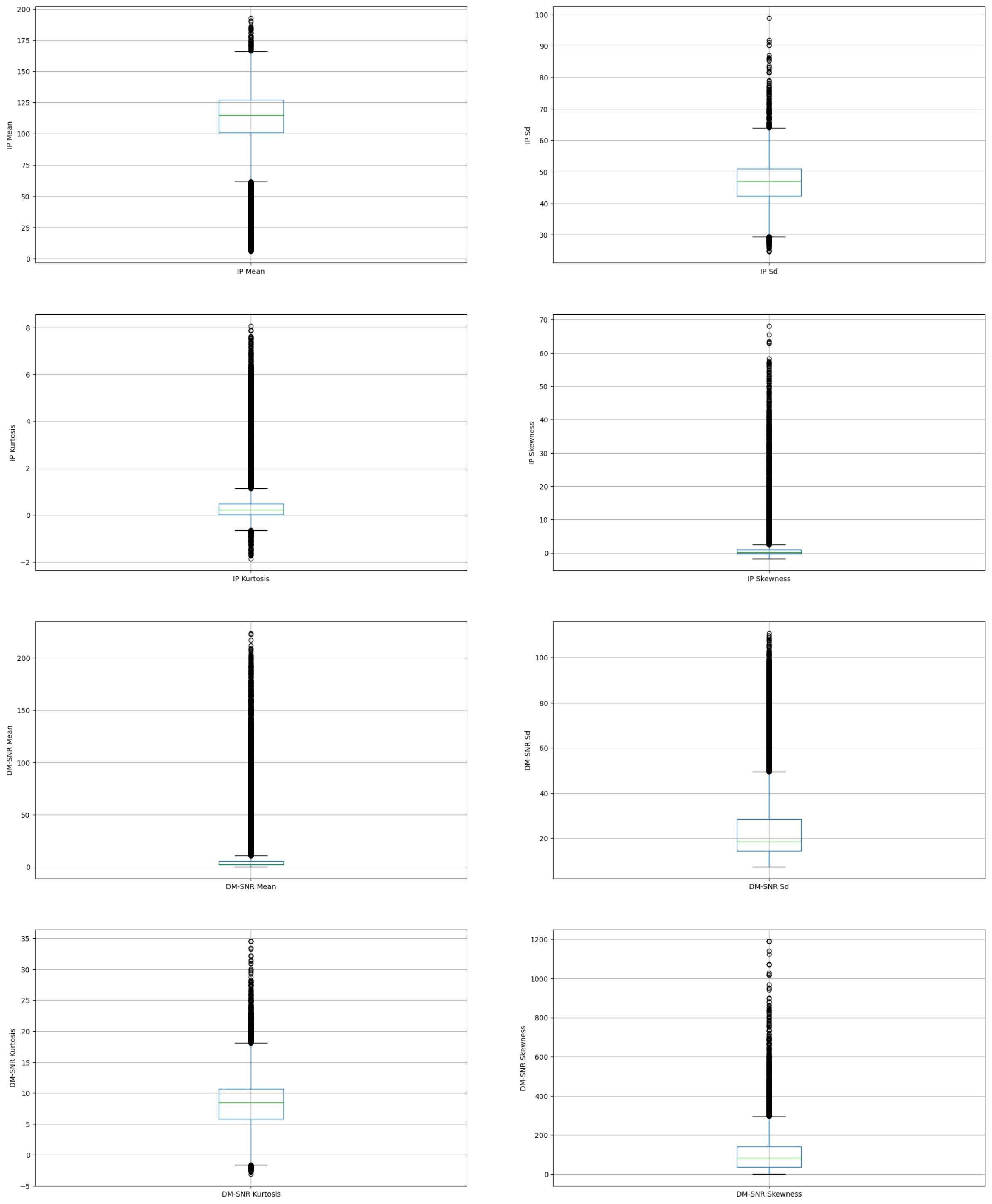
	IP Mean	IP Sd	IP Kurtosis	IP Skewness	DM-SNR Mean	DM-SNR Sd	DM-SNR Kurtosis	DM-SNR Skewness	target_cla
count	17898.00	17898.00	17898.00	17898.00	17898.00	17898.00	17898.00	17898.00	17898.
mean	111.08	46.55	0.48	1.77	12.61	26.33	8.30	104.86	0.
std	25.65	6.84	1.06	6.17	29.47	19.47	4.51	106.51	0.
min	5.81	24.77	-1.88	-1.79	0.21	7.37	-3.14	-1.98	0.
25%	100.93	42.38	0.03	-0.19	1.92	14.44	5.78	34.96	0.
50%	115.08	46.95	0.22	0.20	2.80	18.46	8.43	83.06	0.
75%	127.09	51.02	0.47	0.93	5.46	28.43	10.70	139.31	0.
max	192.62	98.78	8.07	68.10	223.39	110.64	34.54	1191.00	1.

We will draw boxplots to visualise outliers in the above variables.

In [12]:
```
plt.figure(figsize=(24,30))


plt.subplot(4, 2, 1)
fig = df.boxplot(column='IP Mean')
fig.set_title('')
fig.set_ylabel('IP Mean')


plt.subplot(4, 2, 2)
fig = df.boxplot(column='IP Sd')
fig.set_title('')
fig.set_ylabel('IP Sd')
```

```
plt.subplot(4, 2, 3)
fig = df.boxplot(column='IP Kurtosis')
fig.set_title('')
fig.set_ylabel('IP Kurtosis')


plt.subplot(4, 2, 4)
fig = df.boxplot(column='IP Skewness')
fig.set_title('')
fig.set_ylabel('IP Skewness')


plt.subplot(4, 2, 5)
fig = df.boxplot(column='DM-SNR Mean')
fig.set_title('')
fig.set_ylabel('DM-SNR Mean')


plt.subplot(4, 2, 6)
fig = df.boxplot(column='DM-SNR Sd')
fig.set_title('')
fig.set_ylabel('DM-SNR Sd')


plt.subplot(4, 2, 7)
fig = df.boxplot(column='DM-SNR Kurtosis')
fig.set_title('')
fig.set_ylabel('DM-SNR Kurtosis')


plt.subplot(4, 2, 8)
fig = df.boxplot(column='DM-SNR Skewness')
fig.set_title('')
fig.set_ylabel('DM-SNR Skewness')
```

```
Out[12]: Text(0, 0.5, 'DM-SNR Skewness')
```

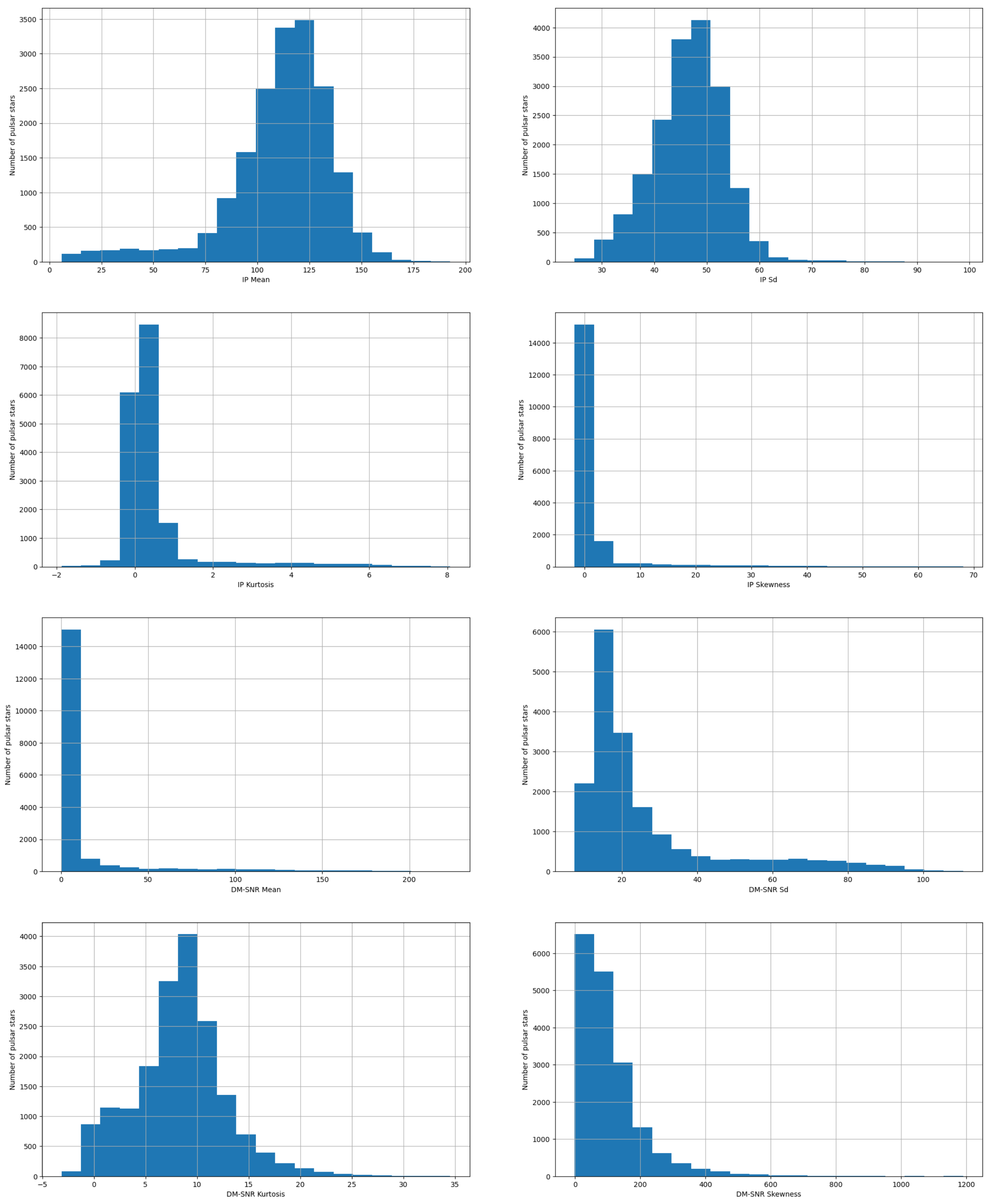


The above boxplots confirm that there are lot of outliers in these variables.

Now, We will plot the histograms to check distributions to find out if they are normal or skewed.

In [13]:
```
plt.figure(figsize=(24,30))

plt.subplot(4, 2, 1)
fig = df['IP Mean'].hist(bins=20)
fig.set_xlabel('IP Mean')
fig.set_ylabel('Number of pulsar stars')


plt.subplot(4, 2, 2)
fig = df['IP Sd'].hist(bins=20)
fig.set_xlabel('IP Sd')
fig.set_ylabel('Number of pulsar stars')


plt.subplot(4, 2, 3)
fig = df['IP Kurtosis'].hist(bins=20)
fig.set_xlabel('IP Kurtosis')
fig.set_ylabel('Number of pulsar stars')



plt.subplot(4, 2, 4)
fig = df['IP Skewness'].hist(bins=20)
fig.set_xlabel('IP Skewness')
fig.set_ylabel('Number of pulsar stars')



plt.subplot(4, 2, 5)
fig = df['DM-SNR Mean'].hist(bins=20)
fig.set_xlabel('DM-SNR Mean')
fig.set_ylabel('Number of pulsar stars')



plt.subplot(4, 2, 6)
fig = df['DM-SNR Sd'].hist(bins=20)
fig.set_xlabel('DM-SNR Sd')
fig.set_ylabel('Number of pulsar stars')



plt.subplot(4, 2, 7)
fig = df['DM-SNR Kurtosis'].hist(bins=20)
fig.set_xlabel('DM-SNR Kurtosis')
fig.set_ylabel('Number of pulsar stars')


plt.subplot(4, 2, 8)
fig = df['DM-SNR Skewness'].hist(bins=20)
fig.set_xlabel('DM-SNR Skewness')
fig.set_ylabel('Number of pulsar stars')
```

Out[13]:
```
Text(0, 0.5, 'Number of pulsar stars')
```



We can see that all the 8 continuous variables are skewed.

In [14]:
```
X = df.drop(['target_class'], axis=1)
y = df['target_class']
```

Split data into separate training and test set

In [15]:
```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2, random_
```

In [16]:
```
X_train.shape, X_test.shape
```

Out[16]:
```
((14318, 8), (3580, 8))
```

Feature Scaling

In [17]:
```
cols = X_train.columns
```

In [18]:
```
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()

X_train = scaler.fit_transform(X_train)

X_test = scaler.transform(X_test)
```

In [19]:
```
X_train = pd.DataFrame(X_train, columns=[cols])
X_test = pd.DataFrame(X_test, columns=[cols])
```

In [20]:
```
X_train.describe()
```

Out[20]:

	IP Mean	IP Sd	IP Kurtosis	IP Skewness	DM-SNR Mean	DM-SNR Sd
count	1.431800e+04	1.431800e+04	1.431800e+04	1.431800e+04	1.431800e+04	1.431800e+04
mean	1.908113e-16	-6.550610e-16	1.042143e-17	3.870815e-17	-8.734147e-17	-1.617802e-16
std	1.000035e+00	1.000035e+00	1.000035e+00	1.000035e+00	1.000035e+00	1.000035e+00
min	-4.035499e+00	-3.181033e+00	-2.185946e+00	-5.744051e-01	-4.239001e-01	-9.733707e-01
25%	-3.896291e-01	-6.069473e-01	-4.256221e-01	-3.188054e-01	-3.664918e-01	-6.125457e-01
50%	1.587461e-01	5.846646e-02	-2.453172e-01	-2.578142e-01	-3.372294e-01	-4.067482e-01
75%	6.267059e-01	6.501017e-01	-1.001238e-02	-1.419621e-01	-2.463724e-01	1.078934e-01
max	3.151882e+00	7.621116e+00	7.008906e+00	1.054430e+01	7.025568e+00	4.292181e+00

We now have X_train dataset ready to be fed into the SVM classifier. We will do it as follows.

Run SVM with default hyperparameters

In [21]:
```
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score

svc=SVC()
svc.fit(X_train,y_train)
y_pred=svc.predict(X_test)
```

```
print('Model accuracy score with default hyperparameters: {0:0.4f}'. format(accurac
```

```
Model accuracy score with default hyperparameters: 0.9827
```

Run SVM with rbf kernel and C=100.0

We have seen that there are outliers in our dataset. So, we should increase the value of C as higher C means fewer outliers. So, We will run SVM with kernel=rbf and C=100.0.

In [22]:
```
svc=SVC(C=100.0)
svc.fit(X_train,y_train)
y_pred=svc.predict(X_test)

print('Model accuracy score with rbf kernel and C=100.0 : {0:0.4f}'. format(accurac
```

```
Model accuracy score with rbf kernel and C=100.0 : 0.9832
```

We can see that we obtain a higher accuracy with C=100.0 as higher C means less outliers.

Now, We will further increase the value of C=1000.0 and check accuracy.

Run SVM with rbf kernel and C=1000.0

In [23]:
```
svc=SVC(C=1000.0)
svc.fit(X_train,y_train)
y_pred=svc.predict(X_test)

print('Model accuracy score with rbf kernel and C=1000.0 : {0:0.4f}'. format(accura
```

```
Model accuracy score with rbf kernel and C=1000.0 : 0.9816
```

Run SVM with linear kernel

Run SVM with linear kernel and C=1.0

In [24]:
```
linear_svc=SVC(kernel='linear', C=1.0)
linear_svc.fit(X_train,y_train)
y_pred_test=linear_svc.predict(X_test)

print('Model accuracy score with linear kernel and C=1.0 : {0:0.4f}'. format(accura
```

```
Model accuracy score with linear kernel and C=1.0 : 0.9830
```

Run SVM with linear kernel and C=100.0

In [25]:
```
linear_svc=SVC(kernel='linear', C=100.0)
linear_svc.fit(X_train,y_train)
y_pred_test=linear_svc.predict(X_test)

print('Model accuracy score with linear kernel and C=100.0 : {0:0.4f}'. format(accu
```

```
Model accuracy score with linear kernel and C=100.0 : 0.9832
```

Run SVM with linear kernel and C=1000.0

In [26]:
```
linear_svc=SVC(kernel='linear', C=1000.0)
linear_svc.fit(X_train,y_train)
y_pred_test=linear_svc.predict(X_test)
```

```
print('Model accuracy score with linear kernel and C=1000.0 : {0:0.4f}'. format(acc
```

```
Model accuracy score with linear kernel and C=1000.0 : 0.9832
```

We can see that we can obtain higher accuracy with C=100.0 and C=1000.0 as compared to C=1.0.

Here, y_test are the true class labels and y_pred are the predicted class labels in the test-set.

Compare the train-set and test-set accuracy

Now, We will compare the train-set and test-set accuracy to check for overfitting.

In [27]:
```
y_pred_train = linear_svc.predict(X_train)
y_pred_train
```

Out[27]:
```
array([0, 0, 1, ..., 0, 0, 0], dtype=int64)
```

In [28]:
```
print('Training-set accuracy score: {0:0.4f}'. format(accuracy_score(y_train, y_pre
```

```
Training-set accuracy score: 0.9785
```

We can see that the training set and test-set accuracy are very much comparable.

Check for overfitting and underfitting

In [29]:
```
print('Training set score: {:.4f}'.format(linear_svc.score(X_train, y_train)))

print('Test set score: {:.4f}'.format(linear_svc.score(X_test, y_test)))
```

```
Training set score: 0.9785
Test set score: 0.9832
```

The training-set accuracy score is 0.9783 while the test-set accuracy to be 0.9830. These two values are quite comparable. So, there is no question of overfitting.

Compare model accuracy with null accuracy

So, the model accuracy is 0.9832. But, we cannot say that our model is very good based on the above accuracy. We must compare it with the null accuracy. Null accuracy is the accuracy that could be achieved by always predicting the most frequent class.

So, we should first check the class distribution in the test set.

In [30]:
```
y_test.value_counts()
```

Out[30]:
```
0    3306
1     274
Name: target_class, dtype: int64
```

We can see that the occurences of most frequent class 0 is 3306. So, we can calculate null accuracy by dividing 3306 by total number of occurences.

In [31]:
```
null_accuracy = (3306/(3306+274))

print('Null accuracy score: {0:0.4f}'. format(null_accuracy))
```

```
Null accuracy score: 0.9235
```

We can see that our model accuracy score is 0.9830 but null accuracy score is 0.9235. So, we can conclude that our SVM classifier is doing a very good job in predicting the class labels.

Run SVM with polynomial kernel

Run SVM with polynomial kernel and C=1.0

In [32]:
```
poly_svc=SVC(kernel='poly', C=1.0)
poly_svc.fit(X_train,y_train)
y_pred=poly_svc.predict(X_test)

print('Model accuracy score with polynomial kernel and C=1.0 : {0:0.4f}'. format(ac
```

```
Model accuracy score with polynomial kernel and C=1.0 : 0.9807
```

Run SVM with polynomial kernel and C=100.0

In [33]:
```
poly_svc=SVC(kernel='poly', C=100.0)
poly_svc.fit(X_train,y_train)
y_pred=poly_svc.predict(X_test)

print('Model accuracy score with polynomial kernel and C=100.0 : {0:0.4f}'. format(
```

```
Model accuracy score with polynomial kernel and C=100.0 : 0.9824
```

Polynomial kernel gives poor performance. It may be overfitting the training set.

Run SVM with sigmoid kernel

Run SVM with sigmoid kernel and C=1.0

In [34]:
```
sigmoid_svc=SVC(kernel='sigmoid', C=1.0)
sigmoid_svc.fit(X_train,y_train)
y_pred=sigmoid_svc.predict(X_test)

print('Model accuracy score with sigmoid kernel and C=1.0 : {0:0.4f}'. format(accur
```

```
Model accuracy score with sigmoid kernel and C=1.0 : 0.8858
```

Run SVM with sigmoid kernel and C=100.0

In [35]:
```
sigmoid_svc=SVC(kernel='sigmoid', C=100.0)
sigmoid_svc.fit(X_train,y_train)
y_pred=sigmoid_svc.predict(X_test)

print('Model accuracy score with sigmoid kernel and C=100.0 : {0:0.4f}'. format(acc
```

```
Model accuracy score with sigmoid kernel and C=100.0 : 0.8855
```

We can see that sigmoid kernel is also performing poorly just like with polynomial kernel.

Comments

We get maximum accuracy with rbf and linear kernel with C=100.0. and the accuracy is 0.9832. Based on the above analysis we can conclude that our classification model accuracy is very good. Our model is doing a very good job in terms of predicting the class labels.

But, this is not true. Here, we have an imbalanced dataset. The problem is that accuracy is an inadequate measure for quantifying predictive performance in the imbalanced dataset problem.

So, we must explore alternative metrices that provide better guidance in selecting models. In particular, we would like to know the underlying distribution of values and the type of errors our classifer is making.

One such metric to analyze the model performance in imbalanced classes problem is Confusion matrix.

Confusion matrix

A confusion matrix is a tool for summarizing the performance of a classification algorithm. A confusion matrix will give us a clear picture of classification model performance and the types of errors produced by the model. It gives us a summary of correct and incorrect predictions broken down by each category. The summary is represented in a tabular form.

Four types of outcomes are possible while evaluating a classification model performance. These four outcomes are described below:-

True Positives (TP) – True Positives occur when we predict an observation belongs to a certain class and the observation actually belongs to that class.

True Negatives (TN) – True Negatives occur when we predict an observation does not belong to a certain class and the observation actually does not belong to that class.

False Positives (FP) – False Positives occur when we predict an observation belongs to a certain class but the observation actually does not belong to that class. This type of error is called Type I error.

False Negatives (FN) – False Negatives occur when we predict an observation does not belong to a certain class but the observation actually belongs to that class. This is a very serious error and it is called Type II error.

In [36]:
```python
from sklearn.metrics import confusion_matrix

cm = confusion_matrix(y_test, y_pred_test)

print('Confusion matrix\n\n', cm)

print('\nTrue Positives(TP) = ', cm[0,0])

print('\nTrue Negatives(TN) = ', cm[1,1])
print('\nFalse Positives(FP) = ', cm[0,1])

print('\nFalse Negatives(FN) = ', cm[1,0])
```

```
Confusion matrix

 [[3289   17]
 [  43  231]]

True Positives(TP) =  3289

True Negatives(TN) =  231

False Positives(FP) =  17

False Negatives(FN) =  43
```

The confusion matrix shows 3289 + 230 = 3519 correct predictions and 17 + 44 = 61 incorrect predictions.

In this case, we have

- True Positives (Actual Positive:1 and Predict Positive:1) - 3289
- True Negatives (Actual Negative:0 and Predict Negative:0) - 230
- False Positives (Actual Negative:0 but Predict Positive:1) - 17 (Type I error)
- False Negatives (Actual Positive:1 but Predict Negative:0) - 44 (Type II error)

Classification metrices

Classification Report

Classification report is another way to evaluate the classification model performance. It displays the precision, recall, f1 and support scores for the model. I have described these terms in later.

In [37]:
```
from sklearn.metrics import classification_report

print(classification_report(y_test, y_pred_test))
```

```
              precision    recall  f1-score   support

           0       0.99      0.99      0.99      3306
           1       0.93      0.84      0.89       274

    accuracy                           0.98      3580
   macro avg       0.96      0.92      0.94      3580
weighted avg       0.98      0.98      0.98      3580
```

Classification accuracy

In [38]:
```
TP = cm[0,0]
TN = cm[1,1]
```

```
FP = cm[0,1]
FN = cm[1,0]
```

In [39]:
```
classification_accuracy = (TP + TN) / float(TP + TN + FP + FN)

print('Classification accuracy : {0:0.4f}'.format(classification_accuracy))
```

```
Classification accuracy : 0.9832
```

Classification error

In [40]:
```
classification_error = (FP + FN) / float(TP + TN + FP + FN)

print('Classification error : {0:0.4f}'.format(classification_error))
```

```
Classification error : 0.0168
```

Precision

Precision can be defined as the percentage of correctly predicted positive outcomes out of all the predicted positive outcomes. It can be given as the ratio of true positives (TP) to the sum of true and false positives (TP + FP).

So, Precision identifies the proportion of correctly predicted positive outcome. It is more concerned with the positive class than the negative class.

Mathematically, precision can be defined as the ratio of TP to (TP + FP).

In [41]:
```
precision = TP / float(TP + FP)


print('Precision : {0:0.4f}'.format(precision))
```

```
Precision : 0.9949
```

Recall

Recall can be defined as the percentage of correctly predicted positive outcomes out of all the actual positive outcomes. It can be given as the ratio of true positives (TP) to the sum of true positives and false negatives (TP + FN). Recall is also called Sensitivity.

Recall identifies the proportion of correctly predicted actual positives.

Mathematically, recall can be defined as the ratio of TP to (TP + FN).

In [42]:
```
recall = TP / float(TP + FN)

print('Recall or Sensitivity : {0:0.4f}'.format(recall))
```

```
Recall or Sensitivity : 0.9871
```

True Positive Rate

True Positive Rate is synonymous with Recall.

```
In [43]: true_positive_rate = TP / float(TP + FN)


         print('True Positive Rate : {0:0.4f}'.format(true_positive_rate))
```

```
True Positive Rate : 0.9871
```

False Positive Rate

```
In [44]: false_positive_rate = FP / float(FP + TN)


         print('False Positive Rate : {0:0.4f}'.format(false_positive_rate))
```

```
False Positive Rate : 0.0685
```

Specificity

```
In [45]: specificity = TN / (TN + FP)

         print('Specificity : {0:0.4f}'.format(specificity))
```

```
Specificity : 0.9315
```

f1-score

f1-score is the weighted harmonic mean of precision and recall. The best possible f1-score would be 1.0 and the worst would be 0.0. f1-score is the harmonic mean of precision and recall. So, f1-score is always lower than accuracy measures as they embed precision and recall into their computation. The weighted average of f1-score should be used to compare classifier models, not global accuracy.

Support

Support is the actual number of occurrences of the class in our dataset.

ROC - AUC

ROC Curve

Another tool to measure the classification model performance visually is ROC Curve. ROC Curve stands for Receiver Operating Characteristic Curve. An ROC Curve is a plot which shows the performance of a classification model at various classification threshold levels.

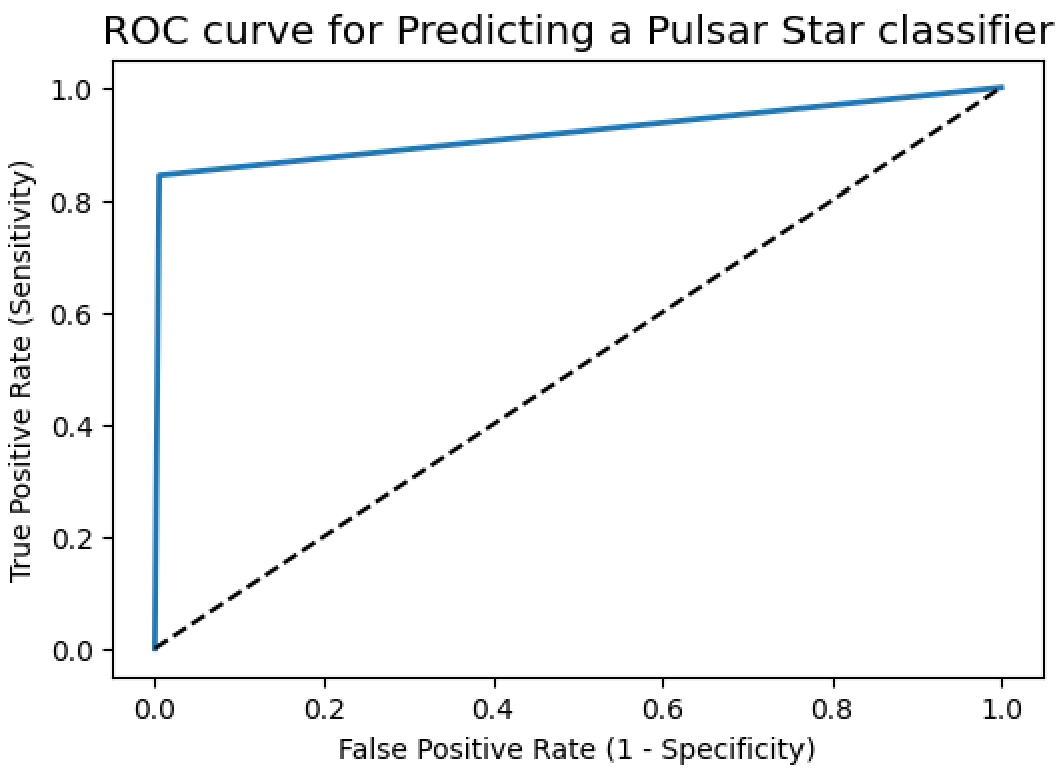
The ROC Curve plots the True Positive Rate (TPR) against the False Positive Rate (FPR) at various threshold levels.

True Positive Rate (TPR) is also called Recall. It is defined as the ratio of TP to (TP + FN).

False Positive Rate (FPR) is defined as the ratio of FP to (FP + TN).

In the ROC Curve, we will focus on the TPR (True Positive Rate) and FPR (False Positive Rate) of a single point. This will give us the general performance of the ROC curve which consists of the TPR and FPR at various threshold levels. So, an ROC Curve plots TPR vs FPR at different classification threshold levels. If we lower the threshold levels, it may result in more items being classified as positve. It will increase both True Positives (TP) and False Positives (FP).

In [46]:
```
from sklearn.metrics import roc_curve

fpr, tpr, thresholds = roc_curve(y_test, y_pred_test)

plt.figure(figsize=(6,4))

plt.plot(fpr, tpr, linewidth=2)

plt.plot([0,1], [0,1], 'k--' )

plt.rcParams['font.size'] = 12

plt.title('ROC curve for Predicting a Pulsar Star classifier')

plt.xlabel('False Positive Rate (1 - Specificity)')

plt.ylabel('True Positive Rate (Sensitivity)')

plt.show()
```



ROC curve help us to choose a threshold level that balances sensitivity and specificity for a particular context.

ROC AUC

ROC AUC stands for Receiver Operating Characteristic - Area Under Curve. It is a technique to compare classifier performance. In this technique, we measure the area under the curve (AUC). A perfect classifier will have a ROC AUC equal to 1, whereas a purely random classifier will have a ROC AUC equal to 0.5.

So, ROC AUC is the percentage of the ROC plot that is underneath the curve.

In [47]:
```
from sklearn.metrics import roc_auc_score

ROC_AUC = roc_auc_score(y_test, y_pred_test)

print('ROC AUC : {:.4f}'.format(ROC_AUC))
```

```
ROC AUC : 0.9190
```

Comments

- ROC AUC is a single number summary of classifier performance. The higher the value, the better the classifier.
- ROC AUC of our model approaches towards 1. So, we can conclude that our classifier does a good job in classifying the pulsar star.

In [48]:
```
from sklearn.model_selection import cross_val_score

Cross_validated_ROC_AUC = cross_val_score(linear_svc, X_train, y_train, cv=10, scor

print('Cross validated ROC AUC : {:.4f}'.format(Cross_validated_ROC_AUC))
```

```
Cross validated ROC AUC : 0.9757
```

Stratified k-fold Cross Validation with shuffle split

k-fold cross-validation is a very useful technique to evaluate model performance. But, it fails here because we have a imbalnced dataset. So, in the case of imbalanced dataset, I will use another technique to evaluate model performance. It is called stratified k-fold cross-validation.

In stratified k-fold cross-validation, we split the data such that the proportions between classes are the same in each fold as they are in the whole dataset.

Moreover, I will shuffle the data before splitting because shuffling yields much better result.

Stratified k-Fold Cross Validation with shuffle split with linear kernel

In [49]:
```
from sklearn.model_selection import KFold


kfold=KFold(n_splits=5, shuffle=True, random_state=0)
```

```
linear_svc=SVC(kernel='linear')


linear_scores = cross_val_score(linear_svc, X, y, cv=kfold)
```

In [50]:
```
print('Stratified cross-validation scores with linear kernel:\n\n{}'.format(linear_
```

```
Stratified cross-validation scores with linear kernel:

[0.98296089 0.97458101 0.97988827 0.97876502 0.97848561]
```

In [51]:
```
print('Average stratified cross-validation score with linear kernel:{:.4f}'.format(
```

```
Average stratified cross-validation score with linear kernel:0.9789
```

Stratified k-Fold Cross Validation with shuffle split with rbf kernel

In [52]:
```
rbf_svc=SVC(kernel='rbf')


rbf_scores = cross_val_score(rbf_svc, X, y, cv=kfold)
```

In [53]:
```
print('Stratified Cross-validation scores with rbf kernel:\n\n{}'.format(rbf_scores
```

```
Stratified Cross-validation scores with rbf kernel:

[0.97849162 0.97011173 0.97318436 0.9709416  0.96982397]
```

In [54]:
```
print('Average stratified cross-validation score with rbf kernel:{:.4f}'.format(rbf
```

```
Average stratified cross-validation score with rbf kernel:0.9725
```

Comments

I obtain higher average stratified k-fold cross-validation score of 0.9789 with linear kernel but the model accuracy is 0.9832. So, stratified cross-validation technique does not help to improve the model performance.

Hyperparameter Optimization using GridSearch CV

In [55]:
```
from sklearn.model_selection import GridSearchCV

from sklearn.svm import SVC

svc=SVC()

parameters = [ {'C':[1, 10, 100, 1000], 'kernel':['linear']},
               {'C':[1, 10, 100, 1000], 'kernel':['rbf'], 'gamma':[0.1, 0.2, 0.3, 0
               {'C':[1, 10, 100, 1000], 'kernel':['poly'], 'degree': [2,3,4] ,'gamm
              ]
```

```
grid_search = GridSearchCV(estimator = svc,
                           param_grid = parameters,
                           scoring = 'accuracy',
                           cv = 5,
                           verbose=0)

grid_search.fit(X_train, y_train)
```

Out[55]:

```
▸ GridSearchCV
▸ estimator: SVC
      ▸ SVC
```

In [56]:

```
print('GridSearch CV best score : {:.4f}\n\n'.format(grid_search.best_score_))
print('Parameters that give the best results :','\n\n', (grid_search.best_params_)
print('\n\nEstimator that was chosen by the search :','\n\n', (grid_search.best_est
```

```
GridSearch CV best score : 0.9793


Parameters that give the best results :

 {'C': 10, 'gamma': 0.3, 'kernel': 'rbf'}


Estimator that was chosen by the search :

 SVC(C=10, gamma=0.3)
```

In [57]:

```
print('GridSearch CV score on test set: {0:0.4f}'.format(grid_search.score(X_test,
```

```
GridSearch CV score on test set: 0.9835
```

Comments

- Our original model test accuracy is 0.9832 while GridSearch CV score on test-set is 0.9835.
- So, GridSearch CV helps to identify the parameters that will improve the performance for this particular model.
- Here, we should not confuse best_score_ attribute of grid_search with the score method on the test-set.
- The score method on the test-set gives the generalization performance of the model. Using the score method, we employ a model trained on the whole training set.
- The best_score_ attribute gives the mean cross-validation accuracy, with cross-validation performed on the training set.

Results and conclusion

- There are outliers in our dataset. So, as I increase the value of C to limit fewer outliers, the accuracy increased. This is true with different kinds of kernels.

- We get maximum accuracy with rbf and linear kernel with C=100.0 and the accuracy is 0.9832. So, we can conclude that our model is doing a very good job in terms of predicting the class labels. But, this is not true. Here, we have an imbalanced dataset. Accuracy is an inadequate measure for quantifying predictive performance in the imbalanced dataset problem. So, we must explore confusion matrix that provide better guidance in selecting models.
- ROC AUC of our model is very close to 1. So, we can conclude that our classifier does a good job in classifying the pulsar star.
- We obtain higher average stratified k-fold cross-validation score of 0.9789 with linear kernel but the model accuracy is 0.9832. So, stratified cross-validation technique does not help to improve the model performance.
- Our original model test accuracy is 0.9832 while GridSearch CV score on test-set is 0.9835. So, GridSearch CV helps to identify the parameters that will improve the performance for this particular model.