Predicting a Pulsar Star with Deep Learning

December 13, 2019

1 Predicting a Pulsar Star with Deep Learning

1.0.1 1. Motivation

Pulsar stars are a very rare type of Neutron star that produce radio emission detectable on Earth and they are of considerable scientific interest as probes of space-time and states of matter. Their emission spreads across the sky and produces a detectable pattern of broadband radio emission. However in practice almost all detections are caused by radio frequency interference and noise, making legitimate signals hard to find.

Having said that, the main purpose of this problem is to build a simple classifier using deep learning tools in order to predict wether a detected signal comes from a pulsar star or from other sources such as noises, interferences, etc.

All information and data related to this problem can be found here: https://www.kaggle.com/pavanraj159/predicting-a-pulsar-star

1.0.2 2. Data Information

Each signal is described by eight continuous variables, and a single class variable. The first four are simple statistics obtained from the integrated pulse profile and the remaining four variables are similarly obtained from the DM-SNR curve. These variables are:

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.
Class
The data set shared here contains 17898 total samples.

1.0.3 3. Dependences

Here we can find the libraries we will use in order to develop a solution for this problem: **numpy | pandas:** Will help us treat and explore the data, and execute vector and matrix operations. **matplotlib | seaborn:** Will help us plot the information so we can visualize it in different ways and have a better understanding of it. **keras:** Will provide us all the necessary deep learning tools to develop a solution for the problem. **sklearn:** We are using this library since it gives us access to some tools to divide our dataset into training and test and some metrics we can use to evaluate our models.

```
In [1]: import numpy as np
        import pandas as pd
        import matplotlib.pyplot as plt
        import seaborn as sns
        import keras
        from prettytable import PrettyTable
        from keras.models import Sequential
        from sklearn.metrics import confusion_matrix, accuracy_score, recall_score, f1_score
        from sklearn.model_selection import train_test_split
        from sklearn.preprocessing import StandardScaler
        from keras.layers import Dense, Dropout
```

```
Using TensorFlow backend.
C:\Users\Ferran\Anaconda3\lib\site-packages\tensorflow\python\framework\dtypes.py:526: FutureWa
  _np_qint8 = np.dtype([("qint8", np.int8, 1)])
C:\Users\Ferran\Anaconda3\lib\site-packages\tensorflow\python\framework\dtypes.py:527: FutureWa
  _np_quint8 = np.dtype([("quint8", np.uint8, 1)])
C:\Users\Ferran\Anaconda3\lib\site-packages\tensorflow\python\framework\dtypes.py:528: FutureWa
  _np_qint16 = np.dtype([("qint16", np.int16, 1)])
C:\Users\Ferran\Anaconda3\lib\site-packages\tensorflow\python\framework\dtypes.py:529: FutureWa
  _np_quint16 = np.dtype([("quint16", np.uint16, 1)])
C:\Users\Ferran\Anaconda3\lib\site-packages\tensorflow\python\framework\dtypes.py:530: FutureWa
  _np_qint32 = np.dtype([("qint32", np.int32, 1)])
C:\Users\Ferran\Anaconda3\lib\site-packages\tensorflow\python\framework\dtypes.py:535: FutureWa
  np_resource = np.dtype([("resource", np.ubyte, 1)])
```

1.0.4 4. Data Exploration

Since this is a deep learning solution we do not want to make any feature selection or special treatment to our data. However let's take a quick look to our data and understand it first!

```
In [2]: #read the csv that contains our data and print the first 5 rows of it
        df = pd.read_csv("pulsar_stars.csv")
        df.head()
```

```
Out[2]:     Mean of the integrated profile  \
        0                       140.562500
        1                       102.507812
        2                       103.015625
        3                       136.750000
        4                        88.726562

            Standard deviation of the integrated profile  \
        0                                      55.683782
```

```
1                                       58.882430
2                                       39.341649
3                                       57.178449
4                                       40.672225

    Excess kurtosis of the integrated profile  \
0                                   -0.234571
1                                    0.465318
2                                    0.323328
3                                   -0.068415
4                                    0.600866

    Skewness of the integrated profile   Mean of the DM-SNR curve  \
0                            -0.699648                   3.199833
1                            -0.515088                   1.677258
2                             1.051164                   3.121237
3                            -0.636238                   3.642977
4                             1.123492                   1.178930

    Standard deviation of the DM-SNR curve  \
0                                19.110426
1                                14.860146
2                                21.744669
3                                20.959280
4                                11.468720

    Excess kurtosis of the DM-SNR curve   Skewness of the DM-SNR curve  \
0                              7.975532                      74.242225
1                             10.576487                     127.393580
2                              7.735822                      63.171909
3                              6.896499                      53.593661
4                             14.269573                     252.567306

   target_class
0             0
1             0
2             0
3             0
4             0
```

First of all we observe the name of these columns is quite long and full of spaces, let's rename them:

```
In [3]: #Changing the name of some columns
        df.columns = ['mean_profile', 'std_profile', 'kurtosis_profile', 'skewness_profile', 'm
                      'std_dmsnr', 'kurtosis_dmsnr', 'skewness_dmsnr', 'target']
```

Now let's check for null values and object datatypes we might want to transform into numerical values:

```
In [4]: #Looking for null values
        df.isna().sum()
```

```
Out[4]: mean_profile        0
        std_profile         0
        kurtosis_profile    0
        skewness_profile    0
        mean_dmsnr          0
        std_dmsnr           0
        kurtosis_dmsnr      0
        skewness_dmsnr      0
        target              0
        dtype: int64
```

```
In [5]: #Looking for object datatypes
        df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 17898 entries, 0 to 17897
Data columns (total 9 columns):
mean_profile        17898 non-null float64
std_profile         17898 non-null float64
kurtosis_profile    17898 non-null float64
skewness_profile    17898 non-null float64
mean_dmsnr          17898 non-null float64
std_dmsnr           17898 non-null float64
kurtosis_dmsnr      17898 non-null float64
skewness_dmsnr      17898 non-null float64
target              17898 non-null int64
dtypes: float64(8), int64(1)
memory usage: 1.2 MB
```

After having checked there are no null values and all columns contain numerical values, let's take a look at some information about our data:

```
In [6]: #Show statistical information of our data
        df.describe()
```

```
Out[6]:        mean_profile   std_profile  kurtosis_profile  skewness_profile  \
        count  17898.000000  17898.000000      17898.000000      17898.000000
        mean     111.079968     46.549532          0.477857          1.770279
        std       25.652935      6.843189          1.064040          6.167913
        min        5.812500     24.772042         -1.876011         -1.791886
        25%      100.929688     42.376018          0.027098         -0.188572
        50%      115.078125     46.947479          0.223240          0.198710
        75%      127.085938     51.023202          0.473325          0.927783
        max      192.617188     98.778911          8.069522         68.101622
```

```
         mean_dmsnr     std_dmsnr  kurtosis_dmsnr  skewness_dmsnr  \
count  17898.000000  17898.000000    17898.000000    17898.000000
mean      12.614400     26.326515        8.303556      104.857709
std       29.472897     19.470572        4.506092      106.514540
min        0.213211      7.370432       -3.139270       -1.976976
25%        1.923077     14.437332        5.781506       34.960504
50%        2.801839     18.461316        8.433515       83.064556
75%        5.464256     28.428104       10.702959      139.309331
max      223.392140    110.642211       34.539844     1191.000837

             target
count  17898.000000
mean       0.091574
std        0.288432
min        0.000000
25%        0.000000
50%        0.000000
75%        0.000000
max        1.000000
```

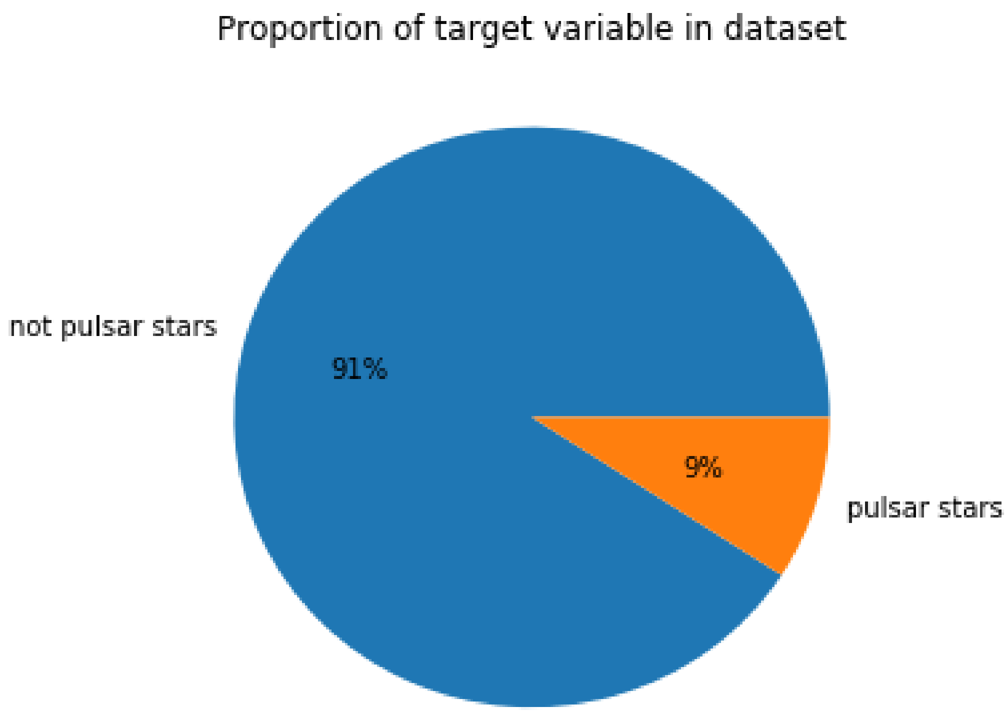
Just by looking at this table we can extract some important information of our data, for example if we take a look at the target column we can see the max value is 1 (pulse star) and the minimum value is 0 (not a star) while the mean of this column tends to 0, which lets us know there are more "false pulse stars" than actual stars. Additionally we clearly see this data needs some scaling since the difference between values is noticeable.

1.0.5 5. Data Visualization

At this point we are going to plot some things that might be of our interest from this dataset. First of all knowing our target variable let's see the difference between values, or better said, the proportion between them:

```
In [7]: #counting pulsars and not pulsars
        pulsar = df[df['target'] ==1]
        pulsar_count = pulsar["target"].value_counts()[1]
        not_pulsar = df[df['target'] == 0]
        not_pulsar_count = not_pulsar["target"].value_counts()[0]
```

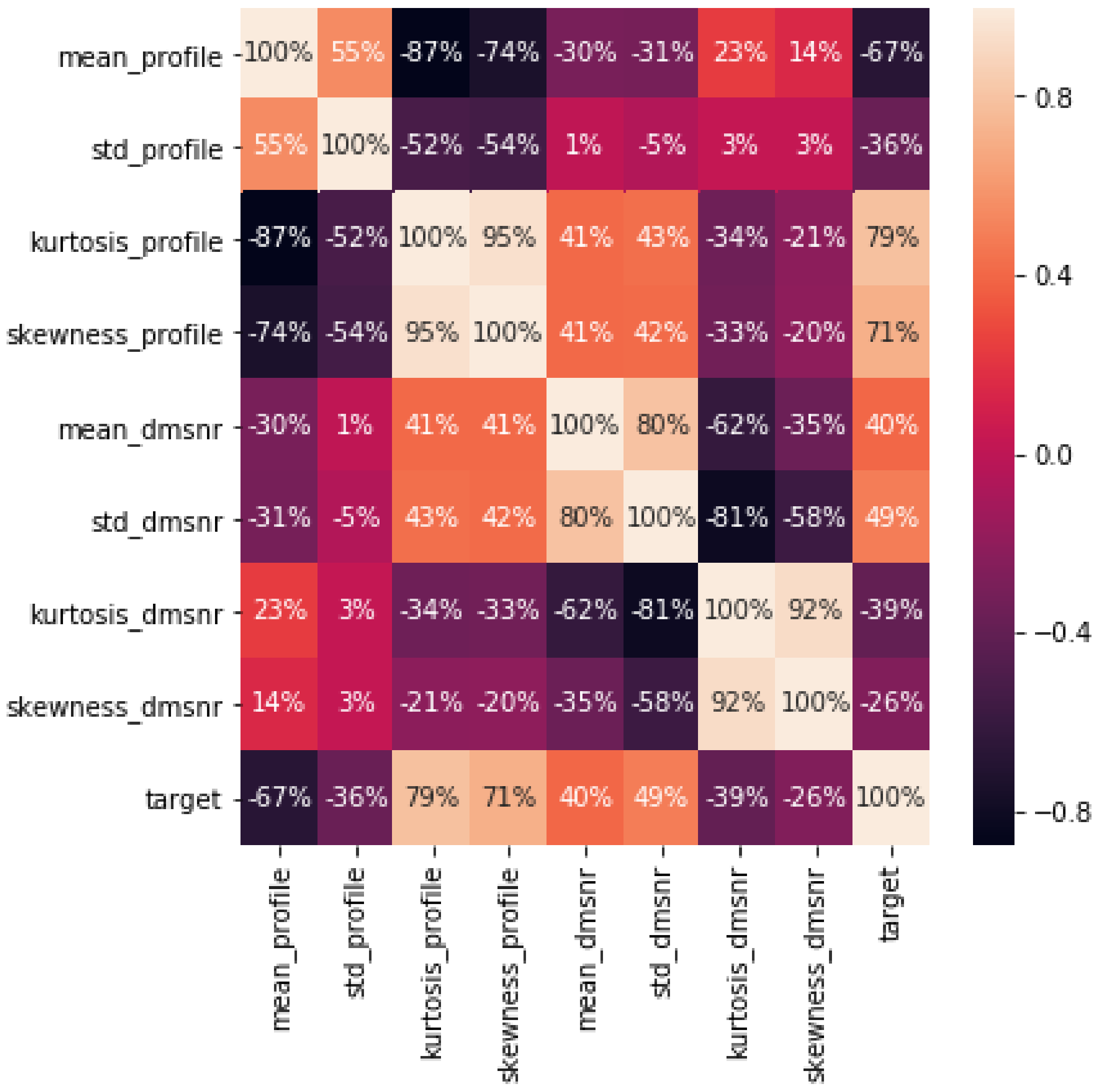
```
In [8]: #pie plotting the stats between pulsars and not pulsars
        plt.figure(figsize=(5,5))
        plt.pie(df["target"].value_counts().values,labels=["not pulsar stars","pulsar stars"],
        plt.title("Proportion of target variable in dataset")
        plt.show()
        print("There are " + str(pulsar_count) + " signals that belong to pulsar stars "
              + "and " + str(not_pulsar_count) + " signals that aren't from pulsars")
```




```
There are 1639 signals that belong to pulsar stars and 16259 signals that aren't from pulsars
```

From the graph above we see that approximately the 10% of our samples are real Pulsar Stars while the other 90% detected signals belong to something else such as noise. Now let's have a look at how features correlate with each other.

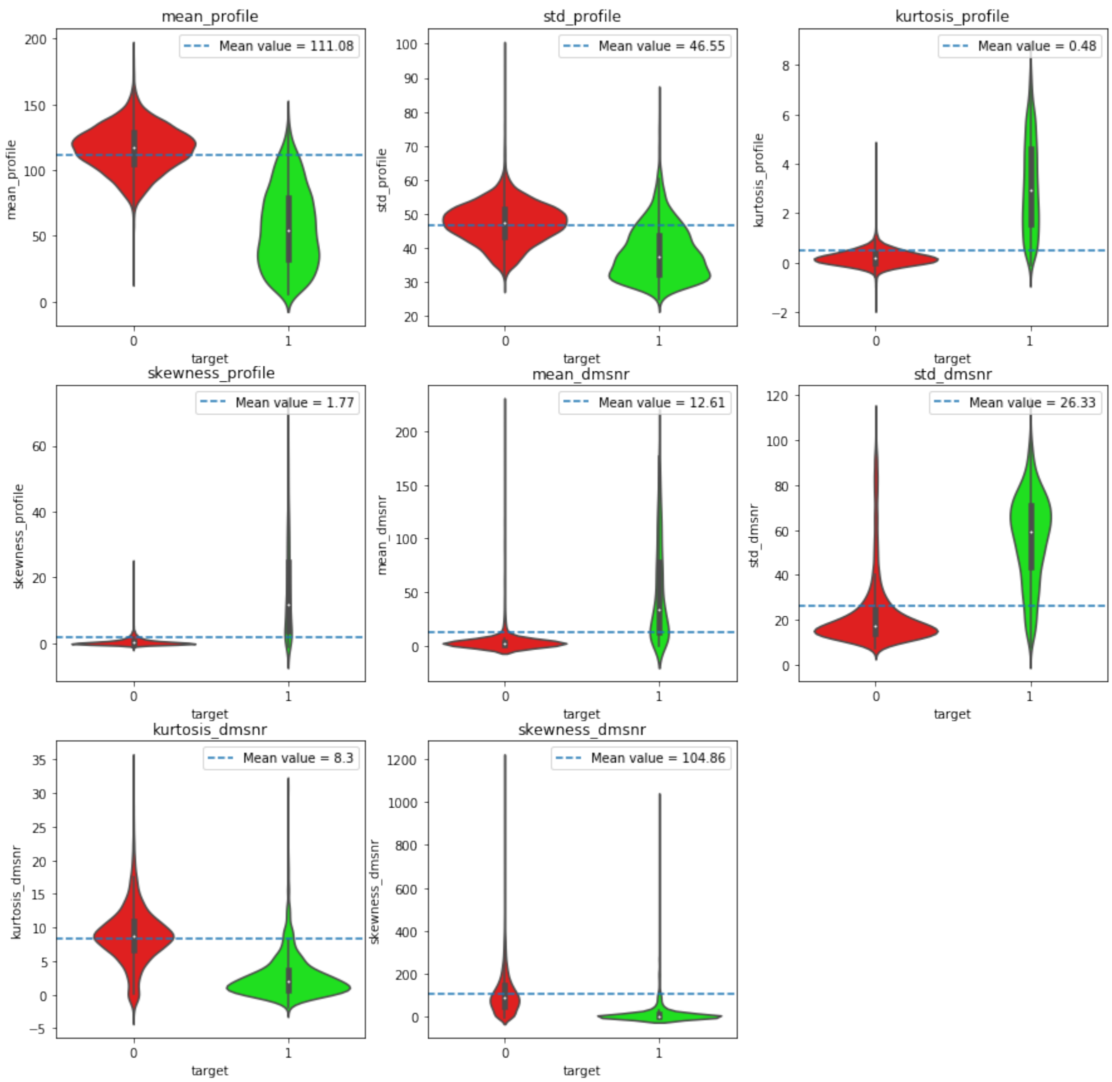
```
In [9]: #plot correlation matrix
        plt.figure(figsize=(6,6))
        sns.heatmap(df.iloc[:,0:9].corr(), annot=True, fmt='.0%')

Out[9]: <matplotlib.axes._subplots.AxesSubplot at 0x19b4f6fab38>
```



Something curious we can see from this correlation matrix is that four of the eight features we have in our dataset correlate positively with our target variable whilst the other four correlate negatively and this is really going to help when training our model since the separation between classes becomes clear. Having said that let's go deeper into investigating these features!

```
In [10]: #violinlpot of all features
         features = df.iloc[:,0:8]
         plt.figure(figsize=(15,20))
         j = 0
         for i in features:
             plt.subplot(4,3,j+1)
             sns.violinplot(x=df["target"],y=df[i],palette=["red","lime"])
             plt.title(i)
             plt.axhline(df[i].mean(),linestyle = "dashed", label ="Mean value = " + str(round
```

```
    plt.legend(loc="best")
    j = j + 1
```



Thanks to these violin plots we are able to extract information about the distribution of values for each of the features our database contain. Furthermore we can view these distributions for the different values our target variable has, plus having this dashed line being the mean value of the feature might help us understand this data better. That said, let's comment some of the features for which we see the separation between classes becomes more clear:

mean_profile. From the correlation matrix we observed the higher the values the less change of the signal coming from a pulsar star, in our violinplot we clearly see it. Additionally, by looking at the mean we could say that if a mean_profile value is above the mean value, that signal might probably to come from another source than a star.

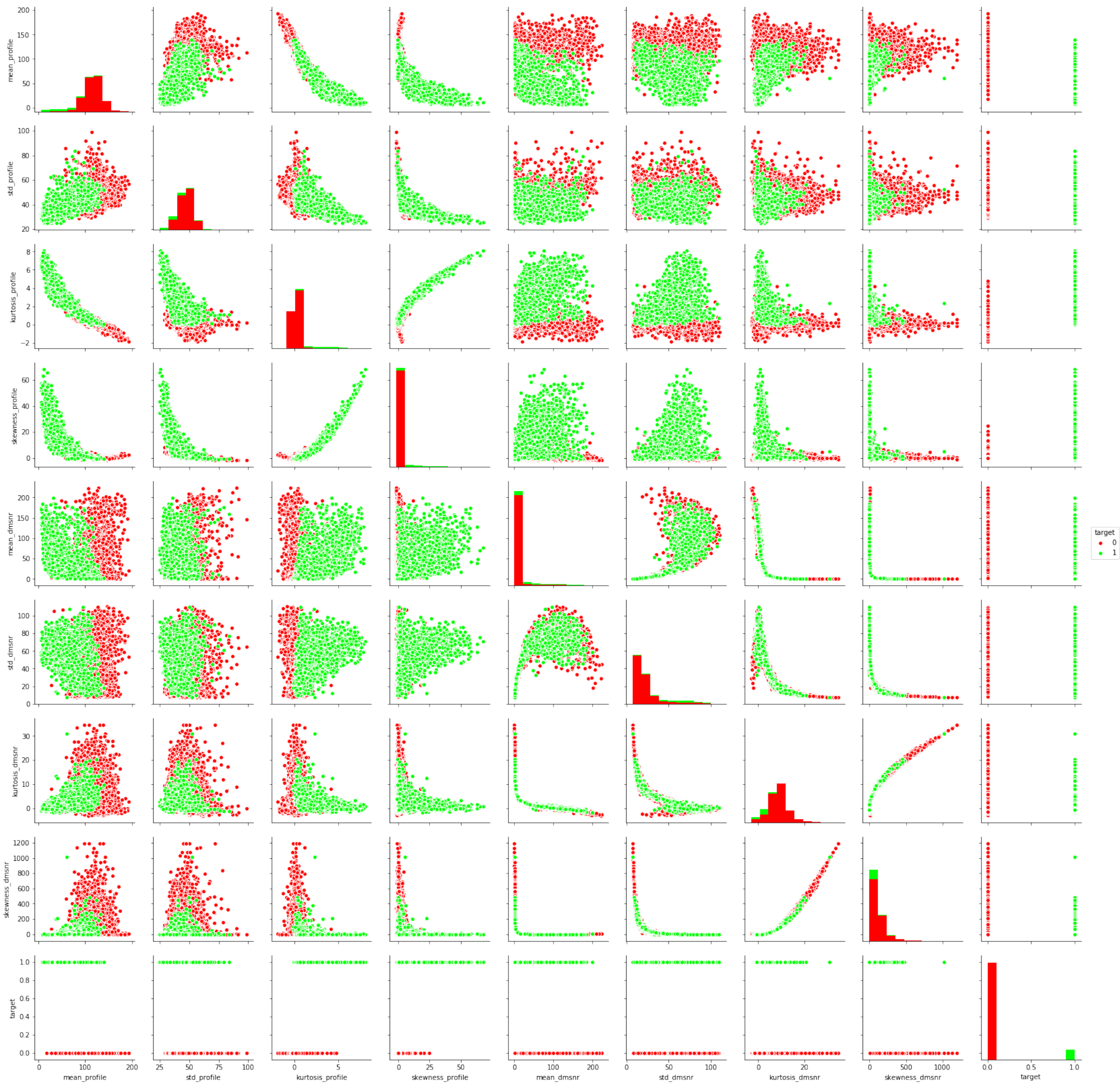
kurtosis_profile. Like mean profile, this feature is also pretty interesting. We clearly observe how the majority of samples whose kurtosis_profile value is above the mean value belong to the group of pulsar stars while, with some outliers that break the rules, lower values than the mean come from other signals. In addition, the distribution of values from the "non pulsar" group is pretty similar, meaning the range of kurtosis_profile values for those signals is quite narrow and the opposite happens to the pulsar group, values tend to be in a range between 0.48-8.

skewness_profile. From skewness_profile we can extract a quite interesting information. It actually seems weird the mean value is just 1.77 when we have values higher than 60 in our dataset. The reason for that is that in our dataset we have approximately 10 times more "non-pulsar" than actual pulsar stars and the majority of skewness values for the non pulsar are pretty close to 0; since the pulsar group is proportionally smaller the mean value is penalized. However that gives us a very important information, the majority of samples whoso skewness_profile value is higher than the mean will probably belong to the pulsar group and, we could say almost 100% samples whose value is higher than 23~ are stars!

mean_dmsnr/skewness_dmsnr. These features are pretty similar in terms of data distribution with the difference being that in mean_dmsn the vast majority of negative star values lay under the mean and in skewness_dmsnr it's exactly the oppositve, pulsar stars are located under the mean value. That said, these distributions look very similar to the one we commented before (skewness_profile) but with one exception: here we can't surely affirm that from "x" value above or below the mean value each sample will belong to a pulsar or not pulsar star, since the range of values for these features is pretty wide.

In order to make sure our hypotesis of data being easily separable for the majority of features is true, we are going to create a pairplot between columns and check if we can visually make that separation.

```
In [11]: #pairplot between features
         sns.pairplot(df, hue="target", palette=["red","lime"])

Out[11]: <seaborn.axisgrid.PairGrid at 0x19b4f9932e8>
```



Thanks to pairplotting it's clear that our data seems to be split in two huge separated groups that can be easily differentiated by just looking at the graphic above.

1.0.6 6. Preparing and Fitting Our Deep Learning Model

Now it's time to prepare our Deep Learning model and, unlike in Machine Learning, we are not going to select any specific features, we are going to set some parameters that we think are going to make the neural network perform and its best and then feed it with all our data. That said, we may still have to split our data into training and set (validation might be OK as well) and scale our data too, if needed.

```
In [12]: #Separing our features from target variable
         X = df.iloc[:,0:8].values
         y = df.iloc[:,8].values
```

In [13]:
```
#Scaling our data
sc = StandardScaler()
X = sc.fit_transform(X)
```

In [14]:
```
#Splitting data
X_train,X_test,y_train,y_test = train_test_split(X,y,test_size = 0.3)
```

Now that we have our data scaled and split into training and test, it's time to define our deep learning model. We are going to use a Sequential model since we want to create our model layer by layer. We are going to be creating a simple fully-connected neuronal network with 3 total layers (2 of them as hidden layers and 1 output layer). Between these layers we are going to use Dropout layers in order to reduce overfitting and we're setting the rate as 0.3.

The input dimension is 8 since we're using eight features and output dimension as 1 since we're only receiving one label. The number of neurons per layer has been chosen randomly without being excessively high. Also important to mention that we're using relu as activation function for the hidden layers since it usually provides better results and sigmoid for the output layer since this is a binary classification problem.

In [15]:
```
#define a sequential Model
model = Sequential()

#First hidden layer
model.add(Dense(16,activation='relu',input_dim=8))
model.add(Dropout(0.25))

#Second hidden Layer
model.add(Dense(8,activation = 'relu'))
model.add(Dropout(0.25))

#Output layer
model.add(Dense(1,activation='sigmoid'))
```

```
WARNING:tensorflow:From C:\Users\Ferran\Anaconda3\lib\site-packages\tensorflow\python\ops\resou
Instructions for updating:
Colocations handled automatically by placer.
```

Having already our model defined there comes the time to compile it. As loss function we're using binary_crossentropy since, as we've already said, this is a binary classification problem. We're aso using adam as optimizer since adapts the learning rate as the training progresses and as far as metrics is concerned we're using accuracy.

In [16]:
```
#Compiling the model
model.compile(loss='binary_crossentropy',optimizer='adam',metrics=['accuracy'])
```

In [17]:
```
#Printing a summary of our model
model.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 16)                144
_________________________________________________________________
dropout_1 (Dropout)          (None, 16)                0
_________________________________________________________________
dense_2 (Dense)              (None, 8)                 136
_________________________________________________________________
dropout_2 (Dropout)          (None, 8)                 0
_________________________________________________________________
dense_3 (Dense)              (None, 1)                 9
=================================================================
Total params: 289
Trainable params: 289
Non-trainable params: 0
_________________________________________________________________
```

Now it usually comes the time to fit our model. However before doing that it might be interesting to adjust the weight of our different output classes, in other words since our database has way more values that don't belong to pulsar stars, we really want to set a higher importance rate to those samples who really come from starts. That's something called handling imbalanced datasets and might help our model to predict, hopefully, our positive pulsar stars samples better.

```
In [18]: #Adjusting class weights
         weight = {0 : 1., 1 : 2.}
```

Having done that, let's fit our model!

```
In [19]: #Fitting our model with training data and, at the same time, using test data for vali
         history = model.fit(X_train, y_train, epochs=100, batch_size=64, class_weight=weight,
```

```
WARNING:tensorflow:From C:\Users\Ferran\Anaconda3\lib\site-packages\tensorflow\python\ops\math_
Instructions for updating:
Use tf.cast instead.
Train on 12528 samples, validate on 5370 samples
Epoch 1/100
12528/12528 [==============================] - 1s 53us/step - loss: 0.4612 - accuracy: 0.8724 -
Epoch 2/100
12528/12528 [==============================] - 0s 16us/step - loss: 0.1996 - accuracy: 0.9630 -
Epoch 3/100
12528/12528 [==============================] - 0s 15us/step - loss: 0.1668 - accuracy: 0.9690 -
Epoch 4/100
12528/12528 [==============================] - 0s 15us/step - loss: 0.1546 - accuracy: 0.9705 -
Epoch 5/100
12528/12528 [==============================] - 0s 15us/step - loss: 0.1507 - accuracy: 0.9729 -
Epoch 6/100
```

```
12528/12528 [==============================] - 0s 15us/step - loss: 0.1433 - accuracy: 0.9751 -
Epoch 7/100
12528/12528 [==============================] - 0s 15us/step - loss: 0.1424 - accuracy: 0.9755 -
Epoch 8/100
12528/12528 [==============================] - 0s 15us/step - loss: 0.1365 - accuracy: 0.9757 -
Epoch 9/100
12528/12528 [==============================] - 0s 15us/step - loss: 0.1357 - accuracy: 0.9769 -
Epoch 10/100
12528/12528 [==============================] - 0s 14us/step - loss: 0.1358 - accuracy: 0.9763 -
Epoch 11/100
12528/12528 [==============================] - 0s 14us/step - loss: 0.1295 - accuracy: 0.9778 -
Epoch 12/100
12528/12528 [==============================] - 0s 14us/step - loss: 0.1317 - accuracy: 0.9771 -
Epoch 13/100
12528/12528 [==============================] - 0s 14us/step - loss: 0.1297 - accuracy: 0.9787 -
Epoch 14/100
12528/12528 [==============================] - 0s 14us/step - loss: 0.1295 - accuracy: 0.9777 -
Epoch 15/100
12528/12528 [==============================] - 0s 14us/step - loss: 0.1284 - accuracy: 0.9773 -
Epoch 16/100
12528/12528 [==============================] - 0s 14us/step - loss: 0.1316 - accuracy: 0.9777 -
Epoch 17/100
12528/12528 [==============================] - 0s 14us/step - loss: 0.1233 - accuracy: 0.9775 -
Epoch 18/100
12528/12528 [==============================] - 0s 14us/step - loss: 0.1250 - accuracy: 0.9780 -
Epoch 19/100
12528/12528 [==============================] - 0s 14us/step - loss: 0.1289 - accuracy: 0.9771 -
Epoch 20/100
12528/12528 [==============================] - 0s 14us/step - loss: 0.1264 - accuracy: 0.9774 -
Epoch 21/100
12528/12528 [==============================] - 0s 14us/step - loss: 0.1270 - accuracy: 0.9770 -
Epoch 22/100
12528/12528 [==============================] - 0s 14us/step - loss: 0.1240 - accuracy: 0.9779 -
Epoch 23/100
12528/12528 [==============================] - 0s 14us/step - loss: 0.1216 - accuracy: 0.9773 -
Epoch 24/100
12528/12528 [==============================] - 0s 14us/step - loss: 0.1245 - accuracy: 0.9772 -
Epoch 25/100
12528/12528 [==============================] - 0s 14us/step - loss: 0.1244 - accuracy: 0.9777 -
Epoch 26/100
12528/12528 [==============================] - 0s 14us/step - loss: 0.1251 - accuracy: 0.9785 -
Epoch 27/100
12528/12528 [==============================] - 0s 14us/step - loss: 0.1269 - accuracy: 0.9785 -
Epoch 28/100
12528/12528 [==============================] - 0s 14us/step - loss: 0.1237 - accuracy: 0.9772 -
Epoch 29/100
12528/12528 [==============================] - 0s 14us/step - loss: 0.1188 - accuracy: 0.9788 -
Epoch 30/100
```

```
12528/12528 [==============================] - 0s 14us/step - loss: 0.1183 - accuracy: 0.9780 -
Epoch 31/100
12528/12528 [==============================] - 0s 14us/step - loss: 0.1214 - accuracy: 0.9771 -
Epoch 32/100
12528/12528 [==============================] - 0s 14us/step - loss: 0.1182 - accuracy: 0.9790 -
Epoch 33/100
12528/12528 [==============================] - 0s 14us/step - loss: 0.1190 - accuracy: 0.9782 -
Epoch 34/100
12528/12528 [==============================] - 0s 14us/step - loss: 0.1223 - accuracy: 0.9778 -
Epoch 35/100
12528/12528 [==============================] - 0s 14us/step - loss: 0.1205 - accuracy: 0.9784 -
Epoch 36/100
12528/12528 [==============================] - 0s 14us/step - loss: 0.1201 - accuracy: 0.9784 -
Epoch 37/100
12528/12528 [==============================] - 0s 14us/step - loss: 0.1144 - accuracy: 0.9792 -
Epoch 38/100
12528/12528 [==============================] - 0s 14us/step - loss: 0.1174 - accuracy: 0.9781 -
Epoch 39/100
12528/12528 [==============================] - 0s 14us/step - loss: 0.1202 - accuracy: 0.9774 -
Epoch 40/100
12528/12528 [==============================] - 0s 14us/step - loss: 0.1202 - accuracy: 0.9783 -
Epoch 41/100
12528/12528 [==============================] - 0s 14us/step - loss: 0.1187 - accuracy: 0.9784 -
Epoch 42/100
12528/12528 [==============================] - 0s 14us/step - loss: 0.1215 - accuracy: 0.9768 -
Epoch 43/100
12528/12528 [==============================] - 0s 14us/step - loss: 0.1172 - accuracy: 0.9781 -
Epoch 44/100
12528/12528 [==============================] - 0s 14us/step - loss: 0.1145 - accuracy: 0.9780 -
Epoch 45/100
12528/12528 [==============================] - 0s 14us/step - loss: 0.1173 - accuracy: 0.9781 -
Epoch 46/100
12528/12528 [==============================] - 0s 14us/step - loss: 0.1197 - accuracy: 0.9782 -
Epoch 47/100
12528/12528 [==============================] - 0s 14us/step - loss: 0.1161 - accuracy: 0.9780 -
Epoch 48/100
12528/12528 [==============================] - 0s 14us/step - loss: 0.1188 - accuracy: 0.9782 -
Epoch 49/100
12528/12528 [==============================] - 0s 14us/step - loss: 0.1212 - accuracy: 0.9780 -
Epoch 50/100
12528/12528 [==============================] - 0s 14us/step - loss: 0.1168 - accuracy: 0.9787 -
Epoch 51/100
12528/12528 [==============================] - 0s 14us/step - loss: 0.1191 - accuracy: 0.9770 -
Epoch 52/100
12528/12528 [==============================] - 0s 14us/step - loss: 0.1184 - accuracy: 0.9777 -
Epoch 53/100
12528/12528 [==============================] - 0s 14us/step - loss: 0.1096 - accuracy: 0.9793 -
Epoch 54/100
```

```
12528/12528 [==============================] - 0s 14us/step - loss: 0.1185 - accuracy: 0.9780 
Epoch 55/100
12528/12528 [==============================] - 0s 14us/step - loss: 0.1197 - accuracy: 0.9785 
Epoch 56/100
12528/12528 [==============================] - 0s 14us/step - loss: 0.1146 - accuracy: 0.9782 
Epoch 57/100
12528/12528 [==============================] - 0s 14us/step - loss: 0.1149 - accuracy: 0.9785 
Epoch 58/100
12528/12528 [==============================] - 0s 14us/step - loss: 0.1188 - accuracy: 0.9788 
Epoch 59/100
12528/12528 [==============================] - 0s 14us/step - loss: 0.1157 - accuracy: 0.9780 
Epoch 60/100
12528/12528 [==============================] - 0s 14us/step - loss: 0.1144 - accuracy: 0.9783 
Epoch 61/100
12528/12528 [==============================] - 0s 14us/step - loss: 0.1184 - accuracy: 0.9773 
Epoch 62/100
12528/12528 [==============================] - 0s 15us/step - loss: 0.1153 - accuracy: 0.9784 
Epoch 63/100
12528/12528 [==============================] - 0s 20us/step - loss: 0.1173 - accuracy: 0.9775 
Epoch 64/100
12528/12528 [==============================] - 0s 14us/step - loss: 0.1188 - accuracy: 0.9789 
Epoch 65/100
12528/12528 [==============================] - 0s 14us/step - loss: 0.1145 - accuracy: 0.9797 
Epoch 66/100
12528/12528 [==============================] - 0s 15us/step - loss: 0.1181 - accuracy: 0.9786 
Epoch 67/100
12528/12528 [==============================] - 0s 16us/step - loss: 0.1168 - accuracy: 0.9788 
Epoch 68/100
12528/12528 [==============================] - 0s 14us/step - loss: 0.1163 - accuracy: 0.9788 
Epoch 69/100
12528/12528 [==============================] - 0s 15us/step - loss: 0.1144 - accuracy: 0.9789 
Epoch 70/100
12528/12528 [==============================] - 0s 15us/step - loss: 0.1151 - accuracy: 0.9792 
Epoch 71/100
12528/12528 [==============================] - 0s 14us/step - loss: 0.1171 - accuracy: 0.9792 
Epoch 72/100
12528/12528 [==============================] - 0s 14us/step - loss: 0.1170 - accuracy: 0.9774 
Epoch 73/100
12528/12528 [==============================] - 0s 15us/step - loss: 0.1154 - accuracy: 0.9786 
Epoch 74/100
12528/12528 [==============================] - 0s 14us/step - loss: 0.1202 - accuracy: 0.9769 
Epoch 75/100
12528/12528 [==============================] - 0s 14us/step - loss: 0.1177 - accuracy: 0.9782 
Epoch 76/100
12528/12528 [==============================] - 0s 14us/step - loss: 0.1144 - accuracy: 0.9784 
Epoch 77/100
12528/12528 [==============================] - 0s 14us/step - loss: 0.1147 - accuracy: 0.9792 
Epoch 78/100
```

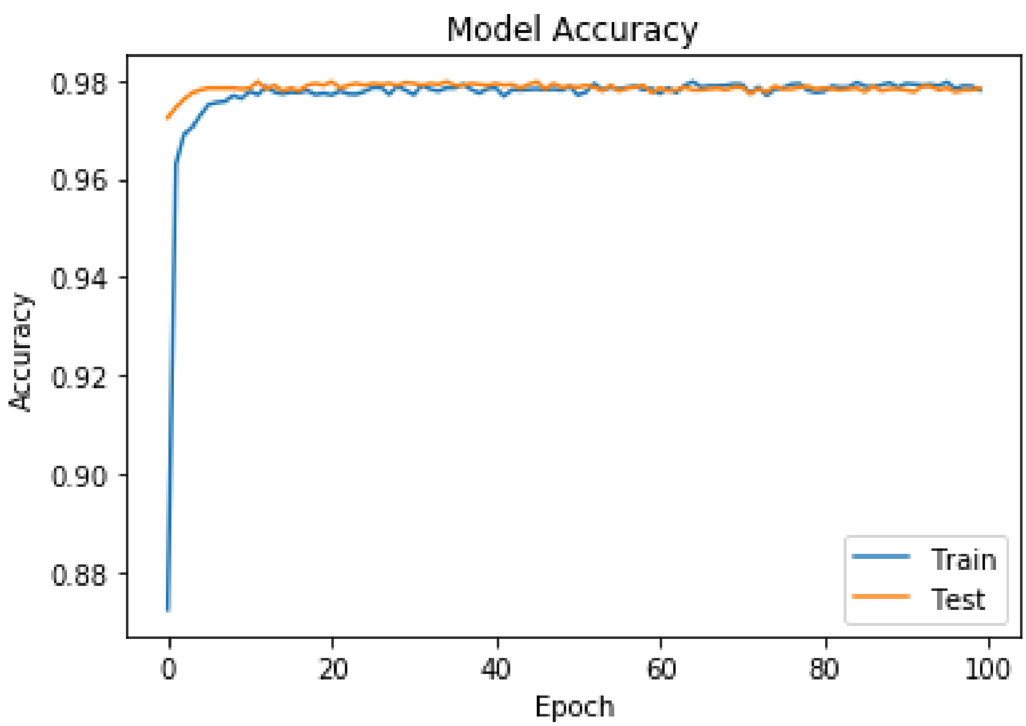
```
12528/12528 [==============================] - 0s 15us/step - loss: 0.1145 - accuracy: 0.9793 -
Epoch 79/100
12528/12528 [==============================] - 0s 15us/step - loss: 0.1174 - accuracy: 0.9780 -
Epoch 80/100
12528/12528 [==============================] - 0s 14us/step - loss: 0.1171 - accuracy: 0.9776 -
Epoch 81/100
12528/12528 [==============================] - 0s 15us/step - loss: 0.1180 - accuracy: 0.9777 -
Epoch 82/100
12528/12528 [==============================] - 0s 15us/step - loss: 0.1128 - accuracy: 0.9788 -
Epoch 83/100
12528/12528 [==============================] - 0s 15us/step - loss: 0.1152 - accuracy: 0.9785 -
Epoch 84/100
12528/12528 [==============================] - 0s 14us/step - loss: 0.1159 - accuracy: 0.9786 -
Epoch 85/100
12528/12528 [==============================] - 0s 16us/step - loss: 0.1177 - accuracy: 0.9796 -
Epoch 86/100
12528/12528 [==============================] - 0s 14us/step - loss: 0.1165 - accuracy: 0.9788 -
Epoch 87/100
12528/12528 [==============================] - 0s 14us/step - loss: 0.1174 - accuracy: 0.9786 -
Epoch 88/100
12528/12528 [==============================] - 0s 14us/step - loss: 0.1144 - accuracy: 0.9790 -
Epoch 89/100
12528/12528 [==============================] - 0s 14us/step - loss: 0.1138 - accuracy: 0.9795 -
Epoch 90/100
12528/12528 [==============================] - 0s 14us/step - loss: 0.1161 - accuracy: 0.9785 -
Epoch 91/100
12528/12528 [==============================] - 0s 15us/step - loss: 0.1137 - accuracy: 0.9793 -
Epoch 92/100
12528/12528 [==============================] - 0s 14us/step - loss: 0.1131 - accuracy: 0.9792 -
Epoch 93/100
12528/12528 [==============================] - 0s 14us/step - loss: 0.1179 - accuracy: 0.9790 -
Epoch 94/100
12528/12528 [==============================] - 0s 14us/step - loss: 0.1144 - accuracy: 0.9792 -
Epoch 95/100
12528/12528 [==============================] - 0s 14us/step - loss: 0.1133 - accuracy: 0.9788 -
Epoch 96/100
12528/12528 [==============================] - 0s 14us/step - loss: 0.1155 - accuracy: 0.9796 -
Epoch 97/100
12528/12528 [==============================] - 0s 15us/step - loss: 0.1149 - accuracy: 0.9783 -
Epoch 98/100
12528/12528 [==============================] - 0s 15us/step - loss: 0.1144 - accuracy: 0.9788 -
Epoch 99/100
12528/12528 [==============================] - 0s 15us/step - loss: 0.1167 - accuracy: 0.9788 -
Epoch 100/100
12528/12528 [==============================] - 0s 14us/step - loss: 0.1161 - accuracy: 0.9780 -
```

As we can see, our model works pretty well! We get an accuracy score of 98% approximately for both our training and test data

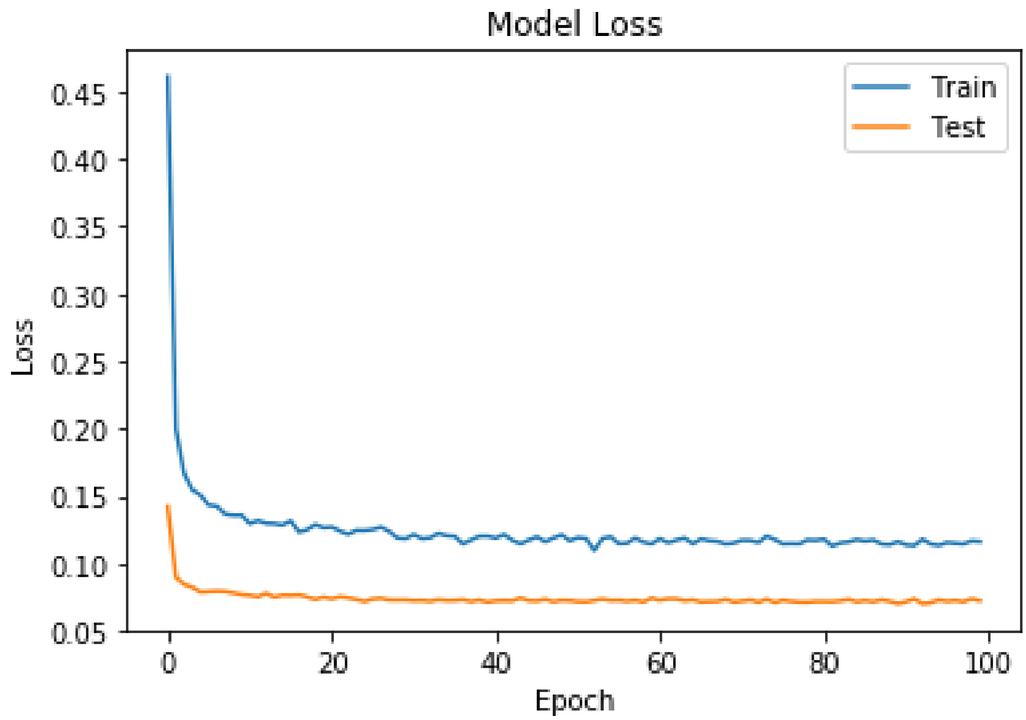
1.0.7 7. Evaluating Our Model

In order to do this evaluation we're going to do two things. First of all we're going to plot the model accuracy and loss for each the training and validation data, and afterwards we're going to make a prediction and, through a confussion matrix, see how it goes.

In [20]:
```
#Visualizing accuracy for both training and test data
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.title('Model Accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend(['Train', 'Test'], loc='lower right')
plt.show()
```

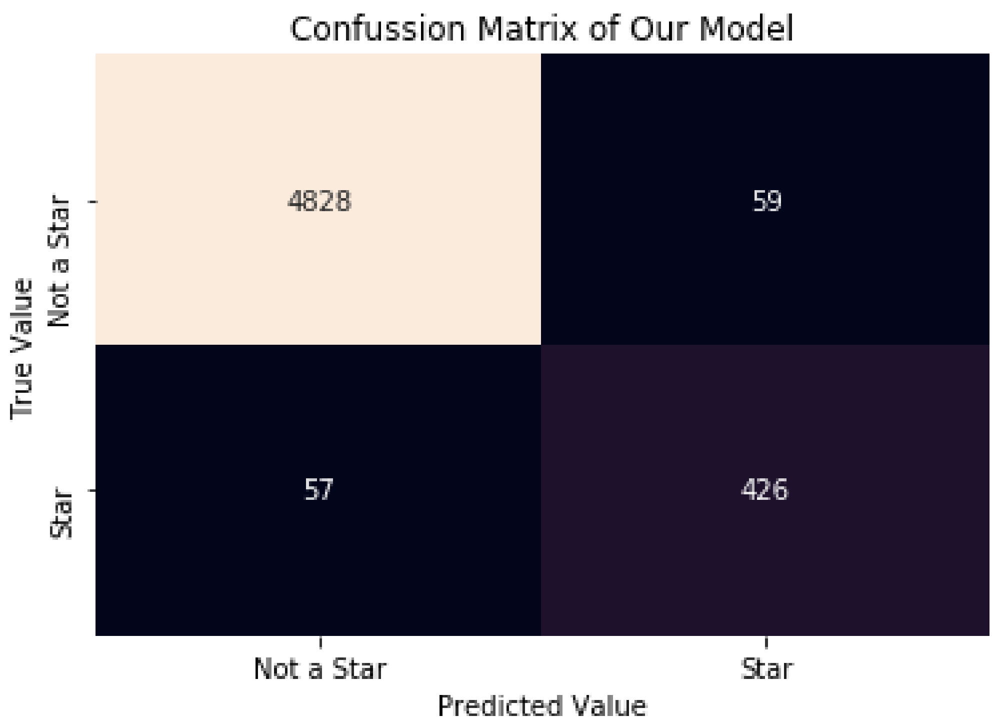


In [21]:
```
#Visualizing loss for both training and test data
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('Model Loss')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(['Train', 'Test'], loc='upper right')
plt.show()
```



With our two graphics already here let's make a prediction and see the confussion matrix:

```
In [22]: #Confussion Matrix for the Random Forest
         label_aux = plt.subplot()
         prediction = model.predict(X_test)
         cm_rf = confusion_matrix(y_test, np.round(prediction))
         cm_rf_m = pd.DataFrame(cm_rf, index = ['Not a Star','Star'], columns = ['Not a Star',
         sns.heatmap(cm_rf_m,annot=True,fmt="d", cbar=False)
         label_aux.set_title("Confussion Matrix of Our Model")
         label_aux.set_xlabel('Predicted Value');label_aux.set_ylabel('True Value');
```




```
In [23]: #Printing some metrics
         ptbl = PrettyTable()
         ptbl.field_names = ["Accuracy", "Recall", "F1Score"]
         ptbl.add_row([accuracy_score(y_test,np.round(prediction)),recall_score(y_test, np.rou
                       f1_score(y_test, np.round(prediction))])
         print(ptbl)
```

```
+--------------------+--------------------+--------------------+
|      Accuracy      |       Recall       |      F1Score       |
+--------------------+--------------------+--------------------+
| 0.9783985102420857 | 0.8819875776397516 | 0.8801652892561982 |
+--------------------+--------------------+--------------------+
```

1.0.8 8. Conclusions

After solving this problem using Deep Learning we have come to the following conclusions: - Our Deep Learning model predicts very well wether a signal comes from a pulsar star or another external source. - Thanks to data being clearly separable in two groups (pulsar and not pulsar), our model has worked very well. - Had we happened to have used a Machine Learning algorithm instead of Deep Learning techniques, feature engineering would have been clear since there is a strong correlation on this dataset. - The number of Epochs (iterations) in our Neural Network could have been reduced since results were pretty much the same after the first 40 iterations. - There are some features that are really well separated as we have explained when plotting the violins. It could be easy to predict the target class just by looking at a sample if some values where below or above the mean value. - This dataset is quite imbalance since there are only 10% of samples that belong to pulsar class; adjusting weights before feeding data to our neural network might help us detecting True Positives, which is what we're interested in.

In case someone wants to know how to solve this particular problem using different Machine Learning algorithms, this following link will take you to a very interesting Kernel where the author solves it using different techniques explaining the algorithms he uses as well as what he is doing each step.