Exoplanet prediction using feedforward net

We will use NASA's Kepler open exoplanet archive dataset (cumulative)

[Source (https://exoplanetarchive.ipac.caltech.edu/cgi-bin/TblView/nph-tblView?app=ExoTbls&config=cumulative)](https://exoplanetarchive.ipac.caltech.edu/cgi-bin/TblView/nph-tblView?app=ExoTbls&config=cumulative)

This will be a binary classification task (i.e, planet or false positive aka not planet). The table contains useful properties of the planetary transit and other features like mass, orbital period etc. More details below.

Importing the Dataset

In [1]:

```
import numpy as np # linear algebra
import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv)
import os

koi_cumm_path = os.path.join('../input', 'koi-cummulative/koi_cummulativ
e.csv')
```

In [2]:

```
dfc = pd.read_csv(koi_cumm_path)
dfc.shape
```

Out[2]:

```
(9564, 49)
```

The Cumulative data has 9564 data points, however, we will soon see that we wont be able to use all of them

For now, lets see what our Categorizations might be.

Each observation in the KOI dataset has a disposition value which tells us what that Object is confirmed to be.

1. CONFIRMED - confirmed planets, these are confirmed to be exoplanets. These are our positive examples.
2. FALSE POSITIVE - as the name suggests, these were thought to be exoplanets but turned out to be false. These will serve us as negative examples.
3. CANDIDATE - these are potential candidates for exoplanets. These will be used for prediction

The KOI dataset also has a `koi-pdisposition` value which tells us the most probable explanation. `koi-disposition` values are finalized after further observations and analyses.

In [120]:

```
dfc = pd.read_csv(koi_cumm_path)
dfc['koi_disposition'].unique()
```

Out[120]:

```
array(['CONFIRMED', 'CANDIDATE', 'FALSE POSITIVE'], dtype=object)
```

In [123]:

```
dfc['koi_pdisposition'].unique()
```

Out[123]:

```
array(['CANDIDATE', 'FALSE POSITIVE'], dtype=object)
```

In [5]:

```
# 2418 candidates

(dfc['koi_disposition'] == "CANDIDATE").value_counts()
```

Out[5]:

```
False    7146
True     2418
Name: koi_disposition, dtype: int64
```

Lets take a look at our dataset before deconstructing it.

In [6]:
```
dfc.head(10)    # first 20 samples
```

Out[6]:

i_pdisposition	koi_score	koi_fpflag_nt	koi_fpflag_ss	koi_fpflag_co	koi_fpflag_ec	...	koi_steff_e
ANDIDATE	1.000	0	0	0	0	...	-81.0
ANDIDATE	0.969	0	0	0	0	...	-81.0
ANDIDATE	0.000	0	0	0	0	...	-176.0
LSE)SITIVE	0.000	0	1	0	0	...	-174.0
ANDIDATE	1.000	0	0	0	0	...	-211.0
ANDIDATE	1.000	0	0	0	0	...	-232.0
ANDIDATE	1.000	0	0	0	0	...	-232.0
ANDIDATE	0.992	0	0	0	0	...	-232.0
LSE)SITIVE	0.000	0	1	1	0	...	-124.0
ANDIDATE	1.000	0	0	0	0	...	-83.0

10 rows × 49 columns

In [7]:

```
# the columns

dfc.columns
```

Out[7]:

```
Index(['kepid', 'kepoi_name', 'kepler_name', 'koi_disposition',
       'koi_pdisposition', 'koi_score', 'koi_fpflag_nt', 'koi_fpflag_s
s',
       'koi_fpflag_co', 'koi_fpflag_ec', 'koi_period', 'koi_period_err
1',
       'koi_period_err2', 'koi_time0bk', 'koi_time0bk_err1',
       'koi_time0bk_err2', 'koi_impact', 'koi_impact_err1', 'koi_impac
t_err2',
       'koi_duration', 'koi_duration_err1', 'koi_duration_err2', 'koi_
depth',
       'koi_depth_err1', 'koi_depth_err2', 'koi_prad', 'koi_prad_err
1',
       'koi_prad_err2', 'koi_teq', 'koi_teq_err1', 'koi_teq_err2', 'ko
i_insol',
       'koi_insol_err1', 'koi_insol_err2', 'koi_model_snr', 'koi_tce_p
lnt_num',
       'koi_tce_delivname', 'koi_steff', 'koi_steff_err1', 'koi_steff_
err2',
       'koi_slogg', 'koi_slogg_err1', 'koi_slogg_err2', 'koi_srad',
       'koi_srad_err1', 'koi_srad_err2', 'ra', 'dec', 'koi_kepmag'],
      dtype='object')
```

We use pandas.DataFrame.info() method to get more info on the dataset. As we see, so many columns have null values in them. Also so many columns which we do not need, like the kepler_name would not help in our classification at all.

In [8]:
```
dfc.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 9564 entries, 0 to 9563
Data columns (total 49 columns):
 #   Column             Non-Null Count  Dtype  
---  ------             --------------  -----  
 0   kepid              9564 non-null   int64  
 1   kepoi_name         9564 non-null   object 
 2   kepler_name        2308 non-null   object 
 3   koi_disposition    9564 non-null   object 
 4   koi_pdisposition   9564 non-null   object 
 5   koi_score          8054 non-null   float64
 6   koi_fpflag_nt      9564 non-null   int64  
 7   koi_fpflag_ss      9564 non-null   int64  
 8   koi_fpflag_co      9564 non-null   int64  
 9   koi_fpflag_ec      9564 non-null   int64  
 10  koi_period         9564 non-null   float64
 11  koi_period_err1    9110 non-null   float64
 12  koi_period_err2    9110 non-null   float64
 13  koi_time0bk        9564 non-null   float64
 14  koi_time0bk_err1   9110 non-null   float64
 15  koi_time0bk_err2   9110 non-null   float64
 16  koi_impact         9201 non-null   float64
 17  koi_impact_err1    9110 non-null   float64
 18  koi_impact_err2    9110 non-null   float64
 19  koi_duration       9564 non-null   float64
 20  koi_duration_err1  9110 non-null   float64
 21  koi_duration_err2  9110 non-null   float64
 22  koi_depth          9201 non-null   float64
 23  koi_depth_err1     9110 non-null   float64
 24  koi_depth_err2     9110 non-null   float64
 25  koi_prad           9201 non-null   float64
 26  koi_prad_err1      9201 non-null   float64
 27  koi_prad_err2      9201 non-null   float64
 28  koi_teq            9201 non-null   float64
 29  koi_teq_err1       0 non-null      float64
 30  koi_teq_err2       0 non-null      float64
 31  koi_insol          9243 non-null   float64
 32  koi_insol_err1     9243 non-null   float64
 33  koi_insol_err2     9243 non-null   float64
 34  koi_model_snr      9201 non-null   float64
```

```
 35  koi_tce_plnt_num   9218 non-null   float64
 36  koi_tce_delivname  9218 non-null   object
 37  koi_steff          9201 non-null   float64
 38  koi_steff_err1     9096 non-null   float64
 39  koi_steff_err2     9081 non-null   float64
 40  koi_slogg          9201 non-null   float64
 41  koi_slogg_err1     9096 non-null   float64
 42  koi_slogg_err2     9096 non-null   float64
 43  koi_srad           9201 non-null   float64
 44  koi_srad_err1      9096 non-null   float64
 45  koi_srad_err2      9096 non-null   float64
 46  ra                 9564 non-null   float64
 47  dec                9564 non-null   float64
 48  koi_kepmag         9563 non-null   float64
dtypes: float64(39), int64(5), object(5)
memory usage: 3.6+ MB
```

Basic preprocessing

We filter out the non numeric columns as they would serve us no purpose. Except `koi_disposition` since that is for labelling purpose.

Now, we need to encode the labels, ie, the koi_disposition values. We replace 'CONFIRMED' with 1 and 'FALSE POSITIVE' with 0 because we will use these label values for categorization. The other two values are arbitrary and serve only to filter out CANDIDATES and NOT DISPOSITIONED samples.

Also, we do the same with `koi-pdisposition` column as well.

In [9]:

```
# all the non-numeric columns

df_numeric = dfc.copy()

koi_disposition_labels = {
    "koi_disposition": {
        "CONFIRMED": 1,
        "FALSE POSITIVE": 0,
        "CANDIDATE": 2,
        "NOT DISPOSITIONED": 3
    },
    "koi_pdisposition": {
        "CONFIRMED": 1,
        "FALSE POSITIVE": 0,
        "CANDIDATE": 2,
        "NOT DISPOSITIONED": 3
    }
}

df_numeric.replace(koi_disposition_labels, inplace=True)
df_numeric
```

Out[9]:

	kepid	kepoi_name	kepler_name	koi_disposition	koi_pdisposition	koi_score	koi_f
0	10797460	K00752.01	Kepler-227 b	1	2	1.000	0
1	10797460	K00752.02	Kepler-227 c	1	2	0.969	0
2	10811496	K00753.01	NaN	2	2	0.000	0
3	10848459	K00754.01	NaN	0	0	0.000	0
4	10854555	K00755.01	Kepler-664 b	1	2	1.000	0
...	...	...	...	...	...	...	...
9559	10090151	K07985.01	NaN	0	0	0.000	0
9560	10128825	K07986.01	NaN	2	2	0.497	0
9561	10147276	K07987.01	NaN	0	0	0.021	0
9562	10155286	K07988.01	NaN	2	2	0.092	0
9563	10156110	K07989.01	NaN	0	0	0.000	0

9564 rows × 49 columns

We now want to filter out some more columns.

koi_score is not needed. It is the probability values for the categorization. However. these probability values will help us in the test phase on hunting new exoplanets among the candidates. So we will need it later.

SO, it is ideal to make a copy of the dataframe at its current state because we will need to come back to some columns again.

Finally, koi_time0bk and koi_time0bk_err1 and 2 are removed because they are the time of the first detected transit minus some offset which is not a useful feature.

for more info on columns check out [https://exoplanetarchive.ipac.caltech.edu/docs/API_kepcandidate_columns.html (https://exoplanetarchive.ipac.caltech.edu/docs/API_kepcandidate_columns.html)](https://exoplanetarchive.ipac.caltech.edu/docs/API_kepcandidate_columns.html)

Let us create two copies of the dataframe now, we will use one containing `koi-pdisposition` and `koi-score` for test phase, and the dataframe containing none of these in train phase.

In [10]:

```
# this is train data

# first we remove all string type columns from the dataframe

df_numeric = df_numeric.select_dtypes(exclude=['object']).copy()
df_test = df_numeric.copy()    # test data

# second, we manually remove some columns which are not needed as mentioned
above.
# additionally, 'koi_teq_err1' and 'koi_teq_err2' have all null values so t
hey too need to be removed

rem_cols = ['kepid', 'koi_pdisposition', 'koi_score', 'koi_time0bk', 'koi
_time0bk_err1', 'koi_time0bk_err2', 'koi_teq_err1', 'koi_teq_err2']
df_numeric.drop(rem_cols, axis=1, inplace=True)

# this is test data
rem_cols_test = [col for col in rem_cols if col not in ['koi_pdispositio
n', 'koi_score']]
df_test.drop(rem_cols_test, axis=1, inplace=True)



df_numeric.head()
```

Out[10]:

	koi_disposition	koi_fpflag_nt	koi_fpflag_ss	koi_fpflag_co	koi_fpflag_ec	koi_period	koi_pe
0	1	0	0	0	0	9.488036	2.780(
1	1	0	0	0	0	54.418383	2.480(
2	2	0	0	0	0	19.899140	1.490(
3	0	0	1	0	0	1.736952	2.630(
4	1	0	0	0	0	2.525592	3.760(

5 rows × 38 columns

In [11]:

```
df_test.head()
```

Out[11]:

	koi_disposition	koi_pdisposition	koi_score	koi_fpflag_nt	koi_fpflag_ss	koi_fpflag_co	koi_fp
0	1	2	1.000	0	0	0	0
1	1	2	0.969	0	0	0	0
2	2	2	0.000	0	0	0	0
3	0	0	0.000	0	1	0	0
4	1	2	1.000	0	0	0	0

5 rows × 40 columns

Now we have a somewhat decent dataset, however, this dataset still has a lot of missing values.

We will simply discard the rows that have atleast one null entry and only consider the non-null dataset for our training.

In [12]:

```
df_numeric = df_numeric[df_numeric.isnull().sum(axis=1) == 0]
df_numeric.describe()
```

Out[12]:

	koi_disposition	koi_fpflag_nt	koi_fpflag_ss	koi_fpflag_co	koi_fpflag_ec	koi_period
count	8744.000000	8744.000000	8744.000000	8744.000000	8744.000000	8744.0000
mean	0.774817	0.183211	0.242681	0.203454	0.125000	56.080618
std	0.829487	4.982739	0.428728	0.402590	0.330738	117.38528
min	0.000000	0.000000	0.000000	0.000000	0.000000	0.259820
25%	0.000000	0.000000	0.000000	0.000000	0.000000	2.667824
50%	1.000000	0.000000	0.000000	0.000000	0.000000	8.970985
75%	2.000000	0.000000	0.000000	0.000000	0.000000	34.190033
max	2.000000	465.000000	1.000000	1.000000	1.000000	1071.2326

8 rows × 38 columns

As we see, `koi_fpflag_nt` has an outlier max value of `465.0` which is improbable since it is a flag and all the other flags are 0 or 1 valued.

In [13]:

```
index = df_numeric[df_numeric.koi_fpflag_nt == df_numeric.koi_fpflag_nt.m
ax()].index
df_numeric.drop(index, inplace=True)
```

In [14]:
```
df_numeric.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 8743 entries, 0 to 9563
Data columns (total 38 columns):
 #   Column             Non-Null Count  Dtype  
---  ------             --------------  -----  
 0   koi_disposition    8743 non-null   int64  
 1   koi_fpflag_nt      8743 non-null   int64  
 2   koi_fpflag_ss      8743 non-null   int64  
 3   koi_fpflag_co      8743 non-null   int64  
 4   koi_fpflag_ec      8743 non-null   int64  
 5   koi_period         8743 non-null   float64
 6   koi_period_err1    8743 non-null   float64
 7   koi_period_err2    8743 non-null   float64
 8   koi_impact         8743 non-null   float64
 9   koi_impact_err1    8743 non-null   float64
 10  koi_impact_err2    8743 non-null   float64
 11  koi_duration       8743 non-null   float64
 12  koi_duration_err1  8743 non-null   float64
 13  koi_duration_err2  8743 non-null   float64
 14  koi_depth          8743 non-null   float64
 15  koi_depth_err1     8743 non-null   float64
 16  koi_depth_err2     8743 non-null   float64
 17  koi_prad           8743 non-null   float64
 18  koi_prad_err1      8743 non-null   float64
 19  koi_prad_err2      8743 non-null   float64
 20  koi_teq            8743 non-null   float64
 21  koi_insol          8743 non-null   float64
 22  koi_insol_err1     8743 non-null   float64
 23  koi_insol_err2     8743 non-null   float64
 24  koi_model_snr      8743 non-null   float64
 25  koi_tce_plnt_num   8743 non-null   float64
 26  koi_steff          8743 non-null   float64
 27  koi_steff_err1     8743 non-null   float64
 28  koi_steff_err2     8743 non-null   float64
 29  koi_slogg          8743 non-null   float64
 30  koi_slogg_err1     8743 non-null   float64
 31  koi_slogg_err2     8743 non-null   float64
 32  koi_srad           8743 non-null   float64
 33  koi_srad_err1      8743 non-null   float64
 34  koi_srad_err2      8743 non-null   float64
```

```
 35  ra                 8743 non-null   float64
 36  dec                8743 non-null   float64
 37  koi_kepmag         8743 non-null   float64
dtypes: float64(33), int64(5)
memory usage: 2.6 MB
```

In [15]:
```
df_test = df_test[df_test.isnull().sum(axis=1) == 0]
df_test.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 7803 entries, 0 to 9563
Data columns (total 40 columns):
 #   Column             Non-Null Count  Dtype  
---  ------             --------------  -----  
 0   koi_disposition    7803 non-null   int64  
 1   koi_pdisposition   7803 non-null   int64  
 2   koi_score          7803 non-null   float64
 3   koi_fpflag_nt      7803 non-null   int64  
 4   koi_fpflag_ss      7803 non-null   int64  
 5   koi_fpflag_co      7803 non-null   int64  
 6   koi_fpflag_ec      7803 non-null   int64  
 7   koi_period         7803 non-null   float64
 8   koi_period_err1    7803 non-null   float64
 9   koi_period_err2    7803 non-null   float64
 10  koi_impact         7803 non-null   float64
 11  koi_impact_err1    7803 non-null   float64
 12  koi_impact_err2    7803 non-null   float64
 13  koi_duration       7803 non-null   float64
 14  koi_duration_err1  7803 non-null   float64
 15  koi_duration_err2  7803 non-null   float64
 16  koi_depth          7803 non-null   float64
 17  koi_depth_err1     7803 non-null   float64
 18  koi_depth_err2     7803 non-null   float64
 19  koi_prad           7803 non-null   float64
 20  koi_prad_err1      7803 non-null   float64
 21  koi_prad_err2      7803 non-null   float64
 22  koi_teq            7803 non-null   float64
 23  koi_insol          7803 non-null   float64
 24  koi_insol_err1     7803 non-null   float64
 25  koi_insol_err2     7803 non-null   float64
 26  koi_model_snr      7803 non-null   float64
 27  koi_tce_plnt_num   7803 non-null   float64
 28  koi_steff          7803 non-null   float64
 29  koi_steff_err1     7803 non-null   float64
 30  koi_steff_err2     7803 non-null   float64
 31  koi_slogg          7803 non-null   float64
 32  koi_slogg_err1     7803 non-null   float64
 33  koi_slogg_err2     7803 non-null   float64
 34  koi_srad           7803 non-null   float64
```

```
 35  koi_srad_err1      7803 non-null   float64
 36  koi_srad_err2      7803 non-null   float64
 37  ra                 7803 non-null   float64
 38  dec                7803 non-null   float64
 39  koi_kepmag         7803 non-null   float64
dtypes: float64(34), int64(6)
memory usage: 2.4 MB
```

Test dataset should only contain `koi_disposition` value 2, since these are all candidate data. We will come back to this dataset later to predict

In [16]:
```
df_test = df_test[df_test.koi_disposition == 2]
df_test
```

Out[16]:

	koi_disposition	koi_pdisposition	koi_score	koi_fpflag_nt	koi_fpflag_ss	koi_fpflag_co	k
2	2	2	0.000	0	0	0	0
37	2	2	1.000	0	0	0	0
58	2	2	0.999	0	0	0	0
62	2	2	0.993	0	0	0	0
63	2	2	0.871	0	0	0	0
...	...	...	...	...	...	...	...
9538	2	2	0.843	0	0	0	0
9542	2	2	0.189	0	0	0	0
9552	2	2	0.519	0	0	0	0
9560	2	2	0.497	0	0	0	0
9562	2	2	0.092	0	0	0	0

1787 rows × 40 columns

Now is a good time to save the `df_test` dataframe to csv for future use.

In [17]:
```
df_test.to_csv('koi_test.csv')
```

We create a copy of this dataframe now to use it for our first neural network training, but we will come back to this dataframe again later. For now we are saving the `df_numeric` into a csv which can be later found in `input/koinumeric/koi_numeric.csv` file.

In [18]:
```
df_numeric.to_csv('koi_numeric.csv')
```

In [19]:
```
df_numeric1 = df_numeric.copy()
```

In [20]:

```
df_numeric1.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 8743 entries, 0 to 9563
Data columns (total 38 columns):
 #   Column             Non-Null Count  Dtype  
---  ------             --------------  -----  
 0   koi_disposition    8743 non-null   int64  
 1   koi_fpflag_nt      8743 non-null   int64  
 2   koi_fpflag_ss      8743 non-null   int64  
 3   koi_fpflag_co      8743 non-null   int64  
 4   koi_fpflag_ec      8743 non-null   int64  
 5   koi_period         8743 non-null   float64
 6   koi_period_err1    8743 non-null   float64
 7   koi_period_err2    8743 non-null   float64
 8   koi_impact         8743 non-null   float64
 9   koi_impact_err1    8743 non-null   float64
 10  koi_impact_err2    8743 non-null   float64
 11  koi_duration       8743 non-null   float64
 12  koi_duration_err1  8743 non-null   float64
 13  koi_duration_err2  8743 non-null   float64
 14  koi_depth          8743 non-null   float64
 15  koi_depth_err1     8743 non-null   float64
 16  koi_depth_err2     8743 non-null   float64
 17  koi_prad           8743 non-null   float64
 18  koi_prad_err1      8743 non-null   float64
 19  koi_prad_err2      8743 non-null   float64
 20  koi_teq            8743 non-null   float64
 21  koi_insol          8743 non-null   float64
 22  koi_insol_err1     8743 non-null   float64
 23  koi_insol_err2     8743 non-null   float64
 24  koi_model_snr      8743 non-null   float64
 25  koi_tce_plnt_num   8743 non-null   float64
 26  koi_steff          8743 non-null   float64
 27  koi_steff_err1     8743 non-null   float64
 28  koi_steff_err2     8743 non-null   float64
 29  koi_slogg          8743 non-null   float64
 30  koi_slogg_err1     8743 non-null   float64
 31  koi_slogg_err2     8743 non-null   float64
 32  koi_srad           8743 non-null   float64
 33  koi_srad_err1      8743 non-null   float64
 34  koi_srad_err2      8743 non-null   float64
```

```
 35  ra                 8743 non-null   float64
 36  dec                8743 non-null   float64
 37  koi_kepmag         8743 non-null   float64
dtypes: float64(33), int64(5)
memory usage: 2.6 MB
```

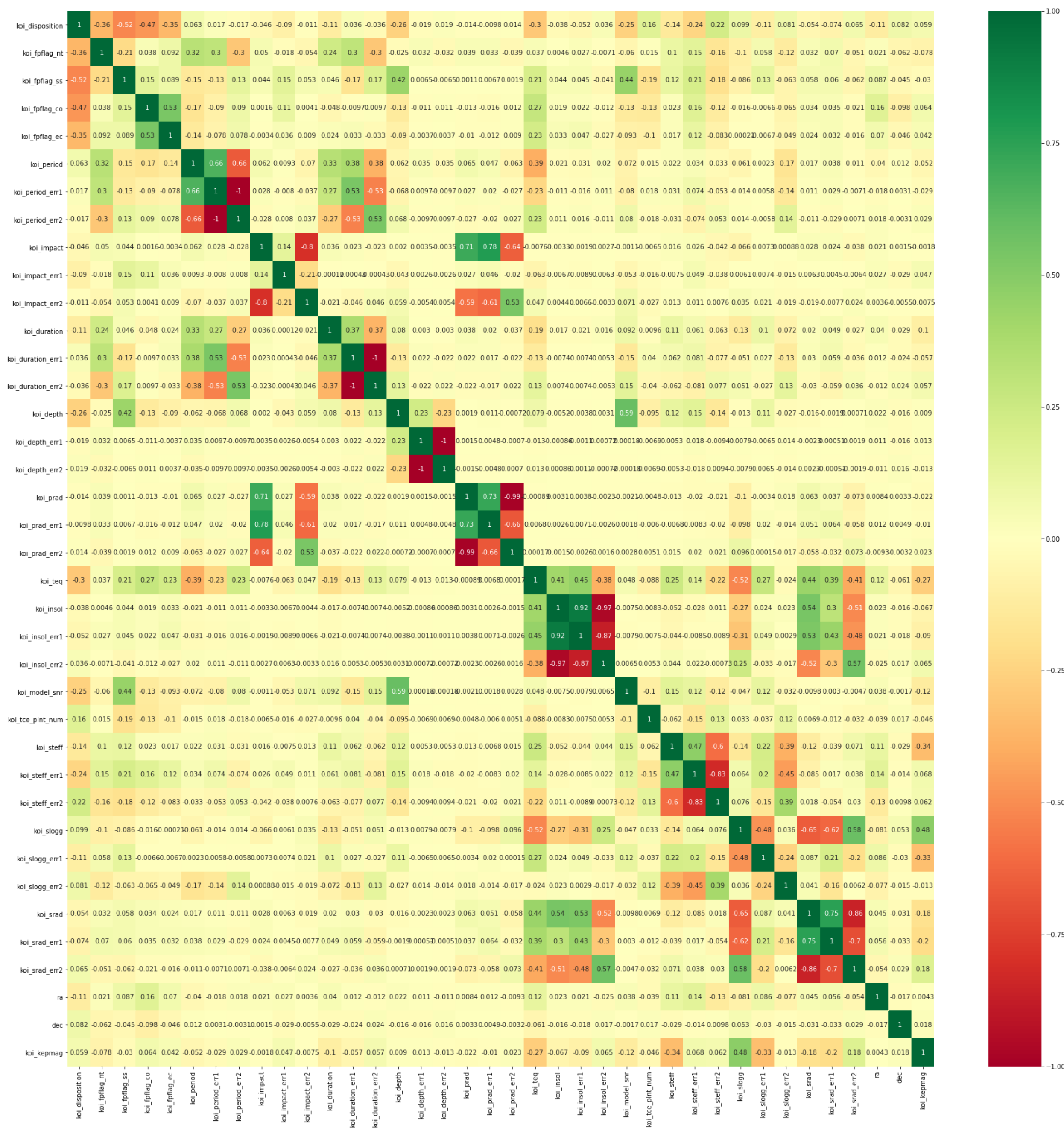
We plot a heatmap of the correlation matrix for our dataframe and we see that overall the data has a lot of uncertainties and very few columns are sufficiently correlated to the target `koi_disposition`

In [21]:

```
import seaborn as sns
import matplotlib.pyplot as plt

fig, ax = plt.subplots(figsize=(30, 30))
sns.heatmap(df_numeric1.corr(), annot=True, cmap="RdYlGn", ax=ax)
```

Out[21]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x7fcfccbfd290>
```



We try to standardize our dataframe because a lot of columns have huge values while others have very small values.

In [22]:

```
from sklearn.preprocessing import StandardScaler

std_scaler = StandardScaler()

# need to exclude the `koi_disposition` column from being standardized


df_numeric1.iloc[:, 5:] = std_scaler.fit_transform(df_numeric1.iloc[:, 5
:])


# df_numeric.iloc[:, 0].to_numpy().reshape(-1, 1).shape
# df_standardized_w_labels = np.c_[df_standardized, df_numeric.iloc[:, 0].t
o_numpy().reshape(-1, 1)]
# df_standardized_w_labels[:3]

df_numeric1.values
```

Out[22]:

```
array([[ 1.        ,  0.        ,  0.        , ..., -0.02937441,
         1.1984836 ,  0.79871776],
       [ 1.        ,  0.        ,  0.        , ..., -0.02937441,
         1.1984836 ,  0.79871776],
       [ 2.        ,  0.        ,  0.        , ...,  1.03307595,
         1.19639385,  0.86496258],
       ...,
       [ 0.        ,  0.        ,  0.        , ...,  0.43802002,
         0.93028566,  0.82700207],
       [ 2.        ,  0.        ,  0.        , ...,  0.9823818 ,
         0.92163466, -2.43834614],
       [ 0.        ,  0.        ,  0.        , ...,  1.03411313,
         0.91493339,  0.4109251 ]])
```

Congratulations. Now we have a complete dataframe with no null values, and also standardized for easier processing.

Now that the preprocessing part is over, we want to create a PyTorch dataset to handle this data and create DataLoader batches

In [23]:
```
import torch
import matplotlib.pyplot as plt
import torch.nn.functional as F
import torch.nn as nn
from torch.utils.data import Dataset, DataLoader
from torch.utils.data import random_split
```

We create a custom PyTorch dataset called `KeplerDataset` class by inheriting the `torch.utils.data.Dataset` class and overriding the `init` , `len` and `getitem` methods.

We have included a flag called `test` which, if set to True, will generate the dataset with `koi_disposition` value `2` or `CANDIDATE` , which we will use for testing

In [24]:
```
class KeplerDataset(Dataset):
    def __init__(self, test=False):
        self.dataframe_orig = pd.read_csv(koi_cumm_path)

        if (test == False):
            self.data = df_numeric1[( df_numeric1.koi_disposition == 1 )
| ( df_numeric1.koi_disposition == 0 )].values
        else:
            self.data = df_numeric1[~(( df_numeric1.koi_disposition == 1
) | ( df_numeric1.koi_disposition == 0 ))].values

        self.X_data = torch.FloatTensor(self.data[:, 1:])
        self.y_data = torch.FloatTensor(self.data[:, 0])

    def __len__(self):
        return len(self.data)

    def __getitem__(self, index):
        return self.X_data[index], self.y_data[index]

    def get_col_len(self):
        return self.X_data.shape[1]

kepler_df = KeplerDataset()
```

In [25]:

```
feature, target = kepler_df[1]
target, feature
```

Out[25]:

```
(tensor(1.),
 tensor([ 0.0000,  0.0000,  0.0000,  0.0000, -0.0142, -0.2187,  0.218
7, -0.0417,
         -0.2022, -0.0919, -0.1606, -0.3148,  0.3148, -0.2771, -0.020
1,  0.0201,
         -0.0321, -0.0436,  0.0277, -0.7682, -0.0454, -0.0661,  0.042
5, -0.2988,
          1.1600, -0.3049, -1.3345,  1.1010,  0.3633, -0.4272,  0.535
5, -0.1369,
         -0.2735,  0.1761, -0.0294,  1.1985,  0.7987]))
```

In [26]:

```
kepler_df.get_col_len()
```

Out[26]:

```
37
```

Now, we want to split our data into training and validation set and also transfer the data to a cuda-enabled device before performing computations

In [27]:

```
# splitting into training and validation set

torch.manual_seed(42)

split_ratio = .7 # 70 / 30 split

train_size = int(len(kepler_df) * split_ratio)
val_size = len(kepler_df) - train_size
train_ds, val_ds = random_split(kepler_df, [train_size, val_size])

len(train_ds), len(val_ds)
```

Out[27]:

```
(4548, 1950)
```

In [28]:

```
batch_size = 32

train_loader = DataLoader(train_ds, batch_size, shuffle=True, num_workers
=4, pin_memory=True)
val_loader = DataLoader(val_ds, batch_size, num_workers=4, pin_memory=Tru
e)
```

In [29]:

```
for features, target in train_loader:
    print(features.size(), target.size())
    break
```

```
torch.Size([32, 37]) torch.Size([32])
```

First feedforward network model

This is a rather simple feedforward architecture with just linear combinations and sigmoid activation. The model architecture is as followed:

1. Input-layer (fully connected) (37 x 32)
2. Sigmoid activation (32)
3. 1st Hidden-layer (fully connected) (32 x 16)
4. Sigmoid activation (16)
5. 2nd Hidden-layer (fully connected) (16 x 8)
6. Sigmoid activation (8)
7. Output-layer (fully connected) (8 x 1)
8. Sigmoid activation (output) (1)

Note that this model incorporates sigmoid at the output layer, so BCELoss() is used.

In [30]:

```
class KOIClassifier(nn.Module):
    def __init__(self, input_dim, out_dim):
        super(KOIClassifier, self).__init__()
        self.linear1 = nn.Linear(input_dim, 32)
        self.linear2 = nn.Linear(32, 32)
        self.linear3 = nn.Linear(32, 16)
        self.linear4 = nn.Linear(16, 8)
        self.linear5 = nn.Linear(8, out_dim)



    def forward(self, xb):
        out = self.linear1(xb)
        out = torch.sigmoid(out)
        out = self.linear2(out)
        out = torch.sigmoid(out)
        out = self.linear3(out)
        out = torch.sigmoid(out)
        out = self.linear4(out)
        out = torch.sigmoid(out)
        out = self.linear5(out)
        out = torch.sigmoid(out)


        return out


    def predict(self, x):
        pred = self.forward(x)
        return pred


    def print_params(self):
        for params in self.parameters():
            print(params)
```

In [31]:
```
input_dim = kepler_df.get_col_len()
out_dim = 1
model = KOIClassifier(input_dim, out_dim)
```

I have already trained this model using the same hyperparameters, the stats are located in `\input\first-nn-stats`

If you want to use the previous stats then uncomment the following cell and run

In [ ]:
```
"""

model_prev = KOIClassifier(input_dim, out_dim)
construct = torch.load('../input/first-nn-stats/checkpoint.pth')
model_prev.load_state_dict(construct['state_dict'])

import seaborn as sns
%matplotlib inline

cf_mat_train = pred_confusion_matrix(model_prev, train_loader)
cf_mat_val = pred_confusion_matrix(model_prev, val_loader)
fig, axes = plt.subplots(nrows=1, ncols=2, figsize=(8, 3))

ax1, ax2 = axes
sns.heatmap(cf_mat_train, fmt='g', annot=True, ax=ax1)
ax1.set_title('Training Data')

sns.heatmap(cf_mat_val, fmt='g', annot=True, ax=ax2)
ax2.set_title('Validation Data')

"""
```

This is where the training happens.

- optimiser = SGD
- number of epochs = 1000
- learning-rate = 0.01
- device of computation = CPU

In [ ]:

```
# training phase
criterion = nn.BCELoss()
optim = torch.optim.SGD(model.parameters(), lr=0.01)
n_epochs = 1000

def train_model():
    for X, y in train_loader:
        for epoch in range(n_epochs):
            optim.zero_grad()
            y_pred = model.forward(X).flatten()
            loss = criterion(y_pred, y)
            loss.backward()
            optim.step()

train_model()
```

In [ ]:

```
# testing the predictions
for X, y in train_loader:
    y_pred = model.forward(X)
    y_pred = y_pred > 0.5
    y_pred = torch.tensor(y_pred, dtype=torch.int32)
    print(y_pred)
    break
```

In [ ]:

```
from sklearn.metrics import confusion_matrix
def pred_confusion_matrix(model, loader):
    with torch.no_grad():
        all_preds = torch.tensor([])
        all_true = torch.tensor([])
        for X, y in loader:
            y_pred = model(X)
            y_pred = torch.tensor(y_pred > 0.5, dtype=torch.float32).flat
ten()
            all_preds = torch.cat([all_preds, y_pred])

            all_true = torch.cat([all_true, y])


    return confusion_matrix(all_true.numpy(), all_preds.numpy())
```

In [ ]:

```
import seaborn as sns
%matplotlib inline

cf_mat_train = pred_confusion_matrix(model, train_loader)
cf_mat_val = pred_confusion_matrix(model, val_loader)
fig, axes = plt.subplots(nrows=1, ncols=2, figsize=(8, 3))

ax1, ax2 = axes
sns.heatmap(cf_mat_train, fmt='g', annot=True, ax=ax1)
ax1.set_title('Training Data')

sns.heatmap(cf_mat_val, fmt='g', annot=True, ax=ax2)
ax2.set_title('Validation Data')
```

In [ ]:
```
checkpoint = {
    'state_dict': model.state_dict(),
    'optimizer': optim.state_dict()
}

torch.save(checkpoint, 'checkpoint.pth')
```

More preprocessing and feature selection

Even though the model seems to perform exceptionally well, we have made some fatal mistakes. Firstly, we standardized the whole dataset, as a result, the informations about the test data got mixed up with the train data. The test data and train data should be separate. Secondly, we added so many columns which are not much needed, for example the columns which have very high correlation coefficient with some others.

Finally, we need a more organized model with fewer parameters, otherwise we will risk overfitting.

Let us try to reduce dimensions first by removing columns which have correlation coeffs higher than 0.80

In [32]:

```
# this is where we return back to the point from where we branched, we take
the numeric dataframe again and apply some feature selection
df_new = pd.read_csv('koi_numeric.csv', index_col=0)
df_new.head()
```

Out[32]:

	koi_disposition	koi_fpflag_nt	koi_fpflag_ss	koi_fpflag_co	koi_fpflag_ec	koi_period	koi_pe
0	1	0	0	0	0	9.488036	2.780
1	1	0	0	0	0	54.418383	2.480
2	2	0	0	0	0	19.899140	1.490
3	0	0	1	0	0	1.736952	2.630
4	1	0	0	0	0	2.525592	3.760

5 rows × 38 columns

In [33]:

```
# a function to remove high correlation columns by selecting the upper tria
ngle of the correlation matrix
# and dropping all columns which have corr value > threshold at any row

def remove_high_corr(df, threshold):
    corr_mat = df.corr()
    trimask = corr_mat.abs().mask(~np.triu(np.ones(corr_mat.shape, dtype=
bool), k=1))
    blocklist = [col for col in trimask.columns if (trimask[col] > thresh
old).any()]
    df.drop(columns=blocklist, axis=1,inplace=True)
    return blocklist
```

In [34]:
```
remove_high_corr(df_new, 0.80)
```

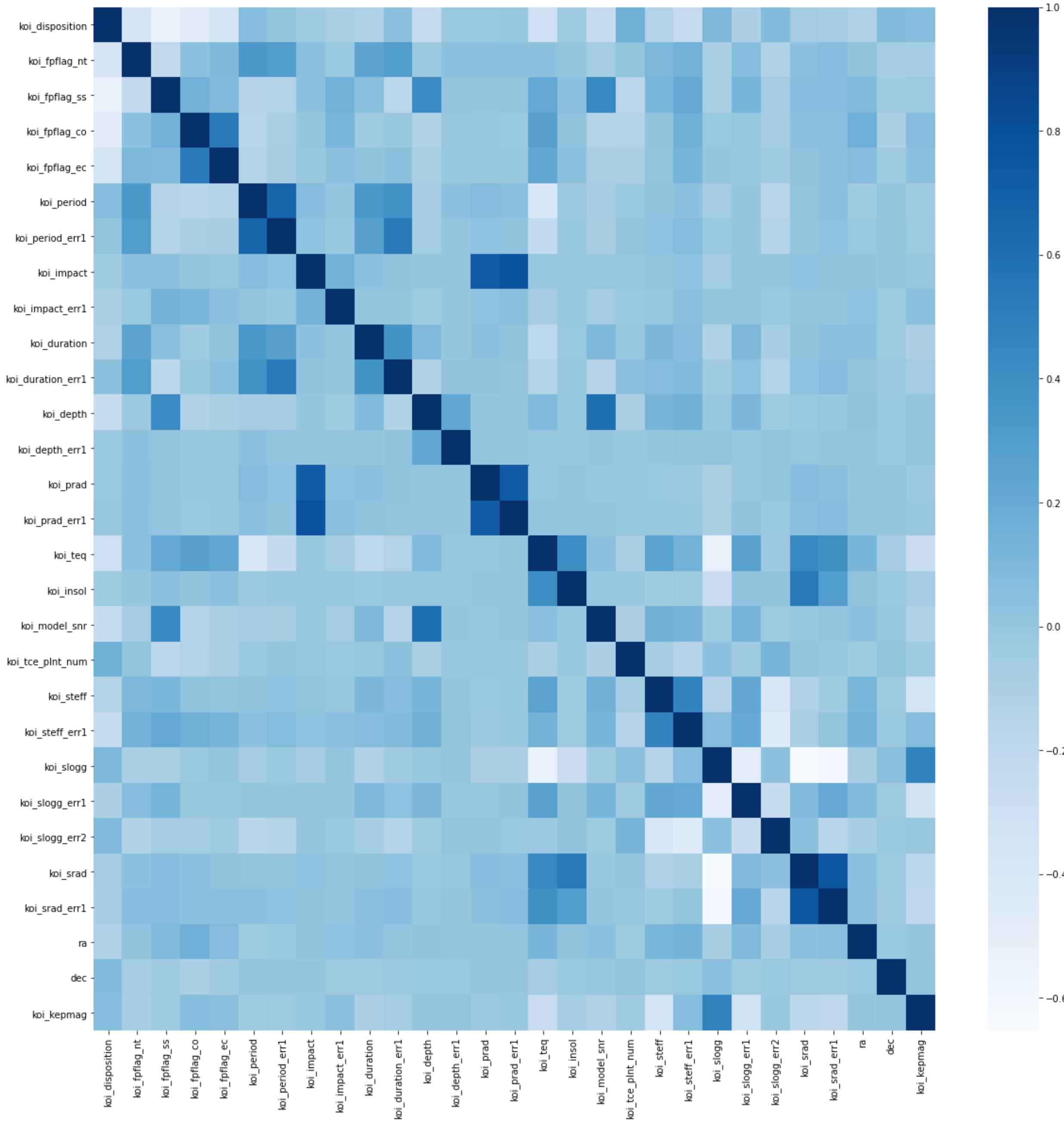
Out[34]:
```
['koi_period_err2',
 'koi_impact_err2',
 'koi_duration_err2',
 'koi_depth_err2',
 'koi_prad_err2',
 'koi_insol_err1',
 'koi_insol_err2',
 'koi_steff_err2',
 'koi_srad_err2']
```

In [35]:

```
fig, ax = plt.subplots(figsize=(20, 20))
sns.heatmap(df_new.corr(), cmap="Blues", ax=ax)
```

Out[35]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x7fcfc5a8a590>
```



Great! Now we can save this csv again and move on the the next parts

In [36]:

```
df_new.head()
```

Out[36]:

	koi_disposition	koi_fpflag_nt	koi_fpflag_ss	koi_fpflag_co	koi_fpflag_ec	koi_period	koi_pe
0	1	0	0	0	0	9.488036	2.780(
1	1	0	0	0	0	54.418383	2.480(
2	2	0	0	0	0	19.899140	1.490(
3	0	0	1	0	0	1.736952	2.630(
4	1	0	0	0	0	2.525592	3.760(

5 rows × 29 columns

So now we have a reduced dataset with 29 columns.

Let us save this reduced dataset also, so that we can use it in our pytorch dataset

In [37]:

```
df_new.to_csv('koi_numeric_reduced.csv')
```

Trying GPU accelaration, new Dataset and model architecture

We want to change the way our dataset class performs. Earlier we stiched together a bunch of modification but this time we want to maintain consistency. We previously standardized the entire dataset, including the test set (which included `koi_disposition` value 2) which was a bad practice. We will now only do standardization on the training data and validation data. When using test data we will do the standardization separately.

In [38]:

```
def get_default_device():
    if torch.cuda.is_available():
        return torch.device('cuda')
    else:
        return torch.device('cpu')

def to_device(data, device):
    """Move tensor(s) to chosen device"""
    if isinstance(data, (list,tuple)):
        return [to_device(x, device) for x in data]
    return data.to(device, non_blocking=True)

class DeviceDataLoader():
    """Wrap a dataloader to move data to a device"""
    def __init__(self, dl, device):
        self.dl = dl
        self.device = device

    def __iter__(self):
        """Yield a batch of data after moving it to device"""
        for b in self.dl:
            yield to_device(b, self.device)

    def __len__(self):
        """Number of batches"""
        return len(self.dl)


device = get_default_device()
device
```

Out[38]:

```
device(type='cuda')
```

The standardization process in the previous model was flawed because it standardized the entire dataset, introducing test data statistics into training and validation data. This is bad, because then our model will be influenced by test data and will never truly learn anything.

I therefore, used `sklearn.model_selection.train_test_split` to split the training data into training and validation data, and created a separate test data by filtering out based on `koi-disposition` values.

I then used `StandardScaler` separately on each dataset to produce independently standardized samples.

The rest of it are similar as before.

In [39]:
```
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split

std_scaler = StandardScaler()

dataframe = pd.read_csv('koi_numeric_reduced.csv', index_col=0)

train_data = dataframe.query('not koi_disposition == 2').values

X = train_data[:, 1:]
y = train_data[:, 0]

val_size = .3
train_X, val_X, train_y, val_y = train_test_split(X, y, test_size=val_siz
e, shuffle=True)

train_X[:, 4:] = std_scaler.fit_transform(train_X[:, 4:])
val_X[:, 4:] = std_scaler.fit_transform(val_X[:, 4:])


# print(f'train_X = {train_X.shape}\n\nval_X = {val_X.shape}\n')

class KOIDataset(Dataset):
    def __init__(self, X_data, y_data):
        self.X_data = torch.FloatTensor(X_data)
        self.y_data = torch.FloatTensor(y_data)


    def __len__(self):
        return len(self.X_data)

    def __getitem__(self, index):
        return self.X_data[index], self.y_data[index]




train_ds = KOIDataset(train_X, train_y)
val_ds = KOIDataset(val_X, val_y)
```

```
for feature, target in train_ds:
    print(feature, target)
    break
```

```
tensor([ 0.0000,  0.0000,  0.0000,  0.0000, -0.2395, -0.1865,  0.0187,
-0.2201,
        -0.1274, -0.1813, -0.3243, -0.0257, -0.0337, -0.0449, -0.6247,
-0.0503,
        -0.3528, -0.3512,  0.0104,  0.1830,  0.5401, -0.4996, -0.3552,
-0.1566,
        -0.1535, -1.4061,  1.1910, -0.0118]) tensor(1.)
```

This time, I used a batch size of 64

In [40]:

```
batch_size = 64
train_loader = DataLoader(train_ds, batch_size, shuffle=True, num_workers
=4, pin_memory=True)
val_loader = DataLoader(val_ds, batch_size, num_workers=4, pin_memory=Tru
e)
```

Ported all the dataloaders to GPU for faster processing.

In [41]:

```
train_loader = DeviceDataLoader(train_loader, device)
val_loader = DeviceDataLoader(val_loader, device)
```

In [42]:
```
for features, target in train_loader:
    print(target, features)
    break
```

```
tensor([0., 0., 1., 1., 0., 0., 0., 1., 1., 0., 0., 0., 1., 1., 0.,
1., 1., 0.,
        1., 1., 0., 1., 0., 0., 0., 0., 0., 0., 1., 1., 1., 1., 0.,
0., 0., 1.,
        1., 0., 1., 0., 0., 1., 0., 1., 0., 1., 1., 0., 0., 0., 1.,
0., 0., 1.,
        0., 0., 1., 0., 1., 0., 1., 0., 1., 0.], device='cuda:0') tens
or([[ 0.0000,  1.0000,  0.0000,  ..., -2.3706,  1.0594,  1.9648],
        [ 0.0000,  0.0000,  1.0000,  ...,  1.3903,  1.0862,  1.2464],
        [ 0.0000,  0.0000,  0.0000,  ..., -0.3760, -0.7383,  0.6718],
        ...,
        [ 0.0000,  0.0000,  1.0000,  ..., -0.5046, -1.2720,  0.4178],
        [ 0.0000,  0.0000,  0.0000,  ..., -0.8612, -0.4007, -0.5102],
        [ 1.0000,  0.0000,  1.0000,  ...,  1.1460, -0.9188,  0.9431]],
       device='cuda:0')
```

New feedforward network

The architecture is as followed.

1. Input Layer (fully connected) (28 x 24)
2. Sigmoid (Activation) (24)
3. Batch Normalization Layer (1D) (24)
4. Hidden Layer (1st) (24 x 16)
5. Sigmoid (Activation) (16)
6. Batch Normalization Layer (1D) (16)
7. Dropout Layer with probability 0.1 (16)
8. A. Output Layer (fully connected) (16 x 1)

In [43]:
```
# a function to measure prediction accuracy

def accuracy(outputs, labels):
    output_labels = torch.round(torch.sigmoid(outputs))    # manually have to activate sigmoid since the nn does not incorporate sigmoid at final layer

    return torch.tensor(torch.sum(output_labels == labels.unsqueeze(1)).item() / len(output_labels))
```

In [58]:

```
from collections import OrderedDict

input_dim = train_X.shape[1]

class KOIClassifierSeq(nn.Module):
    def __init__(self):
        super(KOIClassifierSeq, self).__init__()
        self.model = nn.Sequential(OrderedDict([
              ('fc1', nn.Linear(input_dim, 24)),
              ('sigmoid1', nn.Sigmoid()),
              ('batchnorm1', nn.BatchNorm1d(24)),
              ('fc2', nn.Linear(24, 16)),
              ('sigmoid2', nn.Sigmoid()),
              ('batchnorm2', nn.BatchNorm1d(16)),
              ('dropout', nn.Dropout(p=0.1)),
              ('fc3', nn.Linear(16, 1))
            ]))

    def forward(self, xb):
        return self.model(xb)

    def training_step(self, batch):
        features, label = batch
        out = self(features)
        loss = F.binary_cross_entropy_with_logits(out, label.unsqueeze(1
)) # Calculate loss
        return loss

    def validation_step(self, batch):
        features, label = batch
        out = self(features)
        loss = F.binary_cross_entropy_with_logits(out, label.unsqueeze(1
))   # Calculate loss
        acc = accuracy(out, label)           # Calculate accuracy
        return {'val_loss': loss.detach(), 'val_acc': acc}

    def validation_epoch_end(self, outputs):
        batch_losses = [x['val_loss'] for x in outputs]
        epoch_loss = torch.stack(batch_losses).mean()   # Combine losses
        batch_accs = [x['val_acc'] for x in outputs]
```

```
        epoch_acc = torch.stack(batch_accs).mean()      # Combine accuraci
es
        return {'val_loss': epoch_loss.item(), 'val_acc': epoch_acc.item
()}

    def epoch_end(self, epoch, result):
        print("Epoch [{}], train_loss: {:.4f}, val_loss: {:.4f}, val_acc:
{:.4f}".format(
            epoch, result['train_loss'], result['val_loss'], result['val_
acc']))
```

In [59]:

```
@torch.no_grad()
def evaluate(model, val_loader):
    outputs = [model.validation_step(batch) for batch in val_loader]
    return model.validation_epoch_end(outputs)

def fit(epochs, lr, model, train_loader, val_loader, opt_func=torch.optim
.SGD):
    history = []
    optimizer = opt_func(model.parameters(), lr)
    for epoch in range(epochs):
        # Training Phase
        model.train()
        train_losses = []
        for batch in train_loader:
            loss = model.training_step(batch)
            train_losses.append(loss)
            loss.backward()
            optimizer.step()
            optimizer.zero_grad()
        # Validation phase
        result = evaluate(model, val_loader)
        result['train_loss'] = torch.stack(train_losses).mean().item()
        model.epoch_end(epoch, result)
        history.append(result)
    return history
```

Finally, I have my model ready now. I ported the model to GPU again. The layers can be seen from the following output.

In [72]:
```
model1 = to_device(KOIClassifierSeq(), device)
model1
```

Out[72]:
```
KOIClassifierSeq(
  (model): Sequential(
    (fc1): Linear(in_features=28, out_features=24, bias=True)
    (sigmoid1): Sigmoid()
    (batchnorm1): BatchNorm1d(24, eps=1e-05, momentum=0.1, affine=Tru
e, track_running_stats=True)
    (fc2): Linear(in_features=24, out_features=16, bias=True)
    (sigmoid2): Sigmoid()
    (batchnorm2): BatchNorm1d(16, eps=1e-05, momentum=0.1, affine=Tru
e, track_running_stats=True)
    (dropout): Dropout(p=0.1, inplace=False)
    (fc3): Linear(in_features=16, out_features=1, bias=True)
  )
)
```

Let us fit our model using Adam optimiser and a small learning rate `1e-5`

In [73]:

```
num_epochs = 10
lr = 1e-4
history = fit(num_epochs, lr, model1, train_loader, val_loader, opt_func=
torch.optim.Adam)
```

```
Epoch [0], train_loss: 0.6957, val_loss: 0.6354, val_acc: 0.6627
Epoch [1], train_loss: 0.5986, val_loss: 0.5548, val_acc: 0.7708
Epoch [2], train_loss: 0.5233, val_loss: 0.4909, val_acc: 0.8283
Epoch [3], train_loss: 0.4732, val_loss: 0.4411, val_acc: 0.8511
Epoch [4], train_loss: 0.4228, val_loss: 0.4006, val_acc: 0.8809
Epoch [5], train_loss: 0.3846, val_loss: 0.3654, val_acc: 0.8889
Epoch [6], train_loss: 0.3492, val_loss: 0.3361, val_acc: 0.9036
Epoch [7], train_loss: 0.3169, val_loss: 0.3098, val_acc: 0.9057
Epoch [8], train_loss: 0.3052, val_loss: 0.2893, val_acc: 0.9117
Epoch [9], train_loss: 0.2816, val_loss: 0.2684, val_acc: 0.9274
```

this seems to perform really well. It got a steep jump in terms of accuracy. Let us keep training.

In [74]:

```
num_epochs = 5
lr = 1e-4
history = fit(num_epochs, lr, model1, train_loader, val_loader, opt_func=
torch.optim.Adam)
```

```
Epoch [0], train_loss: 0.2528, val_loss: 0.2501, val_acc: 0.9293
Epoch [1], train_loss: 0.2350, val_loss: 0.2292, val_acc: 0.9374
Epoch [2], train_loss: 0.2206, val_loss: 0.2160, val_acc: 0.9415
Epoch [3], train_loss: 0.2007, val_loss: 0.2010, val_acc: 0.9414
Epoch [4], train_loss: 0.1907, val_loss: 0.1908, val_acc: 0.9444
```

In [75]:

```
# a function to calculate training accuracy

def train_accuracy(model):
    train_acc = []
    for X, y in train_loader:
        out = model(X)
        train_acc.append(accuracy(out, y))

    return torch.stack(train_acc).mean().item()
```

In [76]:

```
train_accuracy(model1)
```

Out[76]:

```
0.9555121660232544
```

So, at the end of training which was relatively fast, We have 97.7% training accuracy and 97.2% validation accuracy. Let us calculate confusion matrix and visualize our predictions.

In [77]:

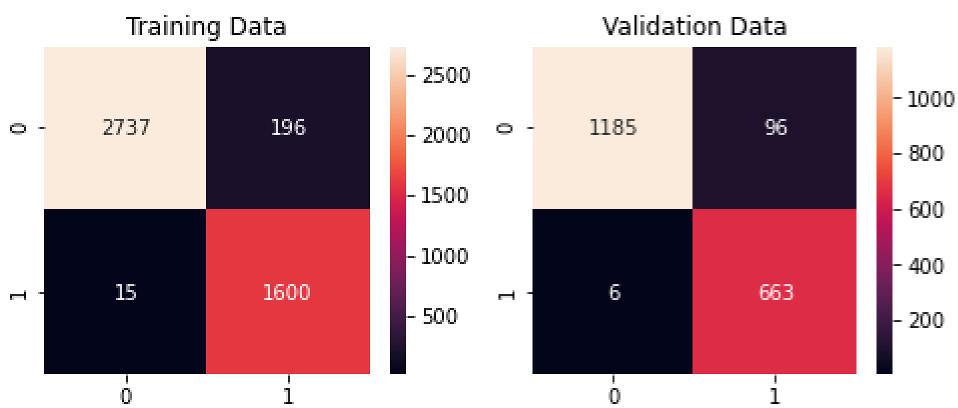
```
from sklearn.metrics import confusion_matrix
def pred_confusion_matrix(model, loader):
    with torch.no_grad():
        all_preds = to_device(torch.tensor([]), device)
        all_true = to_device(torch.tensor([]), device)
        for X, y in loader:
            y_pred = model(X)
            y_pred = torch.round(torch.sigmoid(y_pred))
            all_preds = torch.cat([all_preds, y_pred])

            all_true = torch.cat([all_true, y.unsqueeze(1)])



    return confusion_matrix(all_true.cpu().numpy(), all_preds.cpu().numpy
())
```

In [78]:

```
cf_mat_train = pred_confusion_matrix(model1, train_loader)
cf_mat_val = pred_confusion_matrix(model1, val_loader)
fig, axes = plt.subplots(nrows=1, ncols=2, figsize=(8, 3))

ax1, ax2 = axes
sns.heatmap(cf_mat_train, fmt='g', annot=True, ax=ax1)
ax1.set_title('Training Data')

sns.heatmap(cf_mat_val, fmt='g', annot=True, ax=ax2)
ax2.set_title('Validation Data')
```

Out[78]:

```
Text(0.5, 1.0, 'Validation Data')
```



As we can see, both training and validation data are predicted super accurately. We are not going to train any further. This is more than enough stats for a feedforward neural network classification.

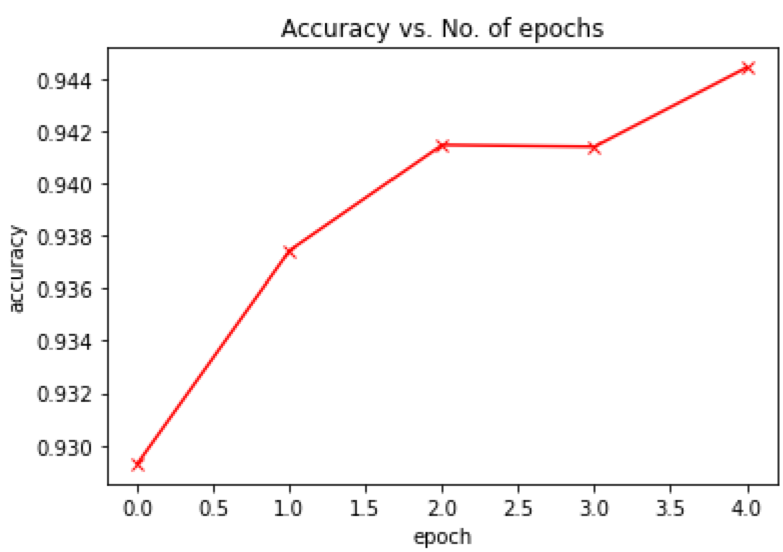
Let us plot the accuracies and predictions.

In [79]:

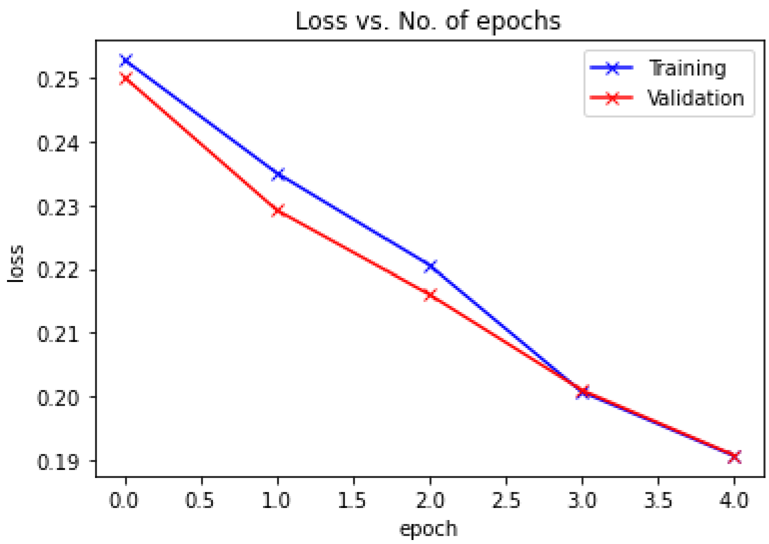
```
def plot_accuracies(history):
    accuracies = [x['val_acc'] for x in history]
    plt.plot(accuracies, '-rx')
    plt.xlabel('epoch')
    plt.ylabel('accuracy')
    plt.title('Accuracy vs. No. of epochs')

def plot_losses(history):
    train_losses = [x.get('train_loss') for x in history]
    val_losses = [x['val_loss'] for x in history]
    plt.plot(train_losses, '-bx')
    plt.plot(val_losses, '-rx')
    plt.xlabel('epoch')
    plt.ylabel('loss')
    plt.legend(['Training', 'Validation'])
    plt.title('Loss vs. No. of epochs')
```

In [80]:

```
plot_accuracies(history)
```



In [81]:
```
plot_losses(history)
```



Saving our model state similar as before.

In [82]:
```
second_model = {
    'state_dict': model1.state_dict()
}

torch.save(second_model, 'second_model.pth')

# I have uploaded the pth file to the /input directory. If needed, you can
 load it from there and load it into a model instance of KOIClassifierSeq
```

Testing on the Test data

In [83]:
```
test_df = pd.read_csv('koi_test.csv', index_col=0)
test_df
```

Out[83]:

	koi_disposition	koi_pdisposition	koi_score	koi_fpflag_nt	koi_fpflag_ss	koi_fpflag_co	k
2	2	2	0.000	0	0	0	0
37	2	2	1.000	0	0	0	0
58	2	2	0.999	0	0	0	0
62	2	2	0.993	0	0	0	0
63	2	2	0.871	0	0	0	0
...	...	...	...	...	...	...	...
9538	2	2	0.843	0	0	0	0
9542	2	2	0.189	0	0	0	0
9552	2	2	0.519	0	0	0	0
9560	2	2	0.497	0	0	0	0
9562	2	2	0.092	0	0	0	0

1787 rows × 40 columns

We need to apply the same preprocessing steps on the test dataset as well.

We can remove `koi_disposition` column as only has value 2 for `CANDIDATE`, We can remove `koi_pdisposition` aswell since it contains same data as `koi_disposition`. We will use the `koi_score` to see our prediction accuracy. We also have to remove the columns we had removed previously from the train and validation data, otherwise there will be a dimensionality mismatch.

In [84]:
```
cols = [
 'koi_disposition',
 'koi_pdisposition',
 'koi_period_err2',
 'koi_impact_err2',
 'koi_duration_err2',
 'koi_depth_err2',
 'koi_prad_err2',
 'koi_insol_err1',
 'koi_insol_err2',
 'koi_steff_err2',
 'koi_srad_err2']

test_df.drop(cols, axis=1, inplace=True)
```

In [85]:
```
test_df.head()
```

Out[85]:

	koi_score	koi_fpflag_nt	koi_fpflag_ss	koi_fpflag_co	koi_fpflag_ec	koi_period	koi_period
2	0.000	0	0	0	0	19.899140	1.490000
37	1.000	0	0	0	0	4.959319	5.150000
58	0.999	0	0	0	0	40.419504	1.140000
62	0.993	0	0	0	0	7.240661	1.620000
63	0.871	0	0	0	0	3.435916	4.730000

5 rows × 29 columns

We perform standardization same as before.

In [86]:
```
test_X = test_df.iloc[:, 1:].values
test_probs = test_df.iloc[:, 0].values

test_X[:, 4:] = std_scaler.fit_transform(test_X[:, 4:])



KOI_test = KOIDataset(test_X, test_probs)
```

In [87]:
```
batch_size = 64
test_loader = DataLoader(KOI_test, batch_size, num_workers=4, pin_memory=
True)
test_loader = DeviceDataLoader(test_loader, device)

for X, y in test_loader:
    print(X.size(), y.size())
    break
```

```
torch.Size([64, 28]) torch.Size([64])
```

In [88]:
```
def predict_probs(model, X):
    probs = torch.sigmoid(model(X))
    return probs
```

As we can see, the predictions are not as accurate.

In [89]:
```
torch.set_printoptions(precision=5, threshold=5000)
with torch.no_grad():
    for X, y in test_loader:
        #print(X, y)
        preds = torch.sigmoid(model1(X))
        for pred, true in zip(preds, y.unsqueeze(1)):
            print(f'model prediction: {pred.item()}\tKOI prediction: {tru
e.item()}')
        break
```

```
model prediction: 0.07717135548591614   KOI prediction: 0.0
model prediction: 0.1424451470375061    KOI prediction: 1.0
model prediction: 0.2067536562681198    KOI prediction: 0.999000012874
6033
model prediction: 0.2110399454832077    KOI prediction: 0.992999970912
9333
model prediction: 0.7160056829452515    KOI prediction: 0.870999991893
7683
model prediction: 0.11829042434692383   KOI prediction: 1.0
model prediction: 0.17240147292613983   KOI prediction: 1.0
model prediction: 0.7823303937911987    KOI prediction: 1.0
model prediction: 0.666327953338623     KOI prediction: 1.0
model prediction: 0.28639310598373413   KOI prediction: 1.0
model prediction: 0.01969689503312111   KOI prediction: 0.0
model prediction: 0.8551582098007202    KOI prediction: 1.0
model prediction: 0.11600252240896225   KOI prediction: 0.998000025749
2065
model prediction: 0.056901298463344574  KOI prediction: 0.994000017642
9749
model prediction: 0.09051447361707687   KOI prediction: 0.0
model prediction: 0.7346372008323669    KOI prediction: 1.0
model prediction: 0.050027504563331604  KOI prediction: 0.0
model prediction: 0.013841806910932064  KOI prediction: 1.0
model prediction: 0.8167394399642944    KOI prediction: 1.0
model prediction: 0.07456996291875839   KOI prediction: 0.996999979019
165
model prediction: 0.03901781886816025   KOI prediction: 1.0
model prediction: 0.34758153557777405   KOI prediction: 0.966000020503
9978
model prediction: 0.3728058338165283    KOI prediction: 1.0
model prediction: 0.530924916267395     KOI prediction: 1.0
model prediction: 0.7859179973602295    KOI prediction: 1.0
model prediction: 0.38144832849502563   KOI prediction: 1.0
model prediction: 0.9131045937538147    KOI prediction: 0.975000023841
8579
model prediction: 0.26961037516593933   KOI prediction: 1.0
model prediction: 0.4234529435634613    KOI prediction: 1.0
model prediction: 0.7373707294464111    KOI prediction: 0.964999973773
9563
model prediction: 0.0106026791036129    KOI prediction: 0.0
```

```
model prediction: 0.7614726424217224    KOI prediction: 0.996999979019
165
model prediction: 0.5415418148040771    KOI prediction: 0.632000029087
0667
model prediction: 0.6718788146972656    KOI prediction: 0.0
model prediction: 0.742770791053772     KOI prediction: 0.999000012874
6033
model prediction: 0.38225802779197693   KOI prediction: 1.0
model prediction: 0.1969139277935028    KOI prediction: 0.751999974250
7935
model prediction: 0.48439592123031616   KOI prediction: 0.0
model prediction: 0.7461593747138977    KOI prediction: 0.930999994277
9541
model prediction: 0.5529417991638184    KOI prediction: 0.711000025272
3694
model prediction: 0.3997315466403961    KOI prediction: 1.0
model prediction: 0.20839659869670868   KOI prediction: 1.0
model prediction: 0.9239524006843567    KOI prediction: 0.0
model prediction: 0.8166447281837463    KOI prediction: 1.0
model prediction: 0.5223369598388672    KOI prediction: 1.0
model prediction: 0.5239954590797424    KOI prediction: 0.990000009536
7432
model prediction: 0.06393381953239441   KOI prediction: 1.0
model prediction: 0.45012158155441284   KOI prediction: 1.0
model prediction: 0.19938087463378906   KOI prediction: 1.0
model prediction: 0.8943300247192383    KOI prediction: 0.995999991893
7683
model prediction: 0.8696718811988831    KOI prediction: 1.0
model prediction: 0.8192925453186035    KOI prediction: 0.999000012874
6033
model prediction: 0.0764036476612091    KOI prediction: 0.0
model prediction: 0.9203847050666809    KOI prediction: 1.0
model prediction: 0.9479935169219971    KOI prediction: 0.694999992847
4426
model prediction: 0.14783397316932678   KOI prediction: 0.992999970912
9333
model prediction: 0.9288195967674255    KOI prediction: 0.992999970912
9333
model prediction: 0.01872302033007145   KOI prediction: 0.870999991893
7683
model prediction: 0.6556954383850098    KOI prediction: 1.0
```

```
model prediction: 0.32978981733322144   KOI prediction: 0.893999993801
1169
model prediction: 0.6975426077842712    KOI prediction: 1.0
model prediction: 0.772310733795166     KOI prediction: 0.996999979019
165
model prediction: 0.48521891236305237   KOI prediction: 0.996999979019
165
model prediction: 0.8535788059234619    KOI prediction: 1.0
```

In [90]:

```
def accuracy_test(outputs, label_prob):
    output_labels = torch.round(torch.sigmoid(outputs))
    labels = torch.round(label_prob)
    return torch.tensor(torch.sum(output_labels == labels.unsqueeze(1)).i
tem() / len(output_labels))


def test_accuracy(model):
    test_acc = []
    with torch.no_grad():
        for X, y in test_loader:
            out = model(X)
            test_acc.append(accuracy_test(out, y))

    return torch.stack(test_acc).mean().item()
```

In [91]:

```
test_accuracy(model1)
```

Out[91]:

```
0.5341158509254456
```

We got a test accuracy which is, unfortunately, not as good. But its a start. We will save the model at its current state.

In [92]:

```
torch.save(model1.state_dict(), 'final_model_53_percent.pth')
```

Using a simpler model

We will now try a simpler model with only one hidden layer with no batchnorm or dropout layer and only sigmoid as activation.

In [100]:

```
class KOIClassifierSimple(nn.Module):
    def __init__(self):
        super(KOIClassifierSimple, self).__init__()
        self.model = nn.Sequential(OrderedDict([
              ('fc1', nn.Linear(input_dim, 24)),
              ('sigmoid1', nn.Sigmoid()),
              ('fc2', nn.Linear(24, 16)),
              ('sigmoid2', nn.Sigmoid()),
              ('fc3', nn.Linear(16, 1))
            ]))

    def forward(self, xb):
        return self.model(xb)

    def training_step(self, batch):
        features, label = batch
        out = self(features)
        loss = F.binary_cross_entropy_with_logits(out, label.unsqueeze(1
)) # Calculate loss
        return loss

    def validation_step(self, batch):
        features, label = batch
        out = self(features)
        loss = F.binary_cross_entropy_with_logits(out, label.unsqueeze(1
))   # Calculate loss
        acc = accuracy(out, label)           # Calculate accuracy
        return {'val_loss': loss.detach(), 'val_acc': acc}

    def validation_epoch_end(self, outputs):
        batch_losses = [x['val_loss'] for x in outputs]
        epoch_loss = torch.stack(batch_losses).mean()   # Combine losses
        batch_accs = [x['val_acc'] for x in outputs]
        epoch_acc = torch.stack(batch_accs).mean()      # Combine accuraci
es
        return {'val_loss': epoch_loss.item(), 'val_acc': epoch_acc.item
()}

    def epoch_end(self, epoch, result):
        print("Epoch [{}], train_loss: {:.4f}, val_loss: {:.4f}, val_acc:
```

```
{:.4f}".format(
            epoch, result['train_loss'], result['val_loss'], result['val_
acc']))
```

In [113]:

```
model2 = to_device(KOIClassifierSimple(), device)
model2
```

Out[113]:

```
KOIClassifierSimple(
  (model): Sequential(
    (fc1): Linear(in_features=28, out_features=24, bias=True)
    (sigmoid1): Sigmoid()
    (fc2): Linear(in_features=24, out_features=16, bias=True)
    (sigmoid2): Sigmoid()
    (fc3): Linear(in_features=16, out_features=1, bias=True)
  )
)
```

In [114]:

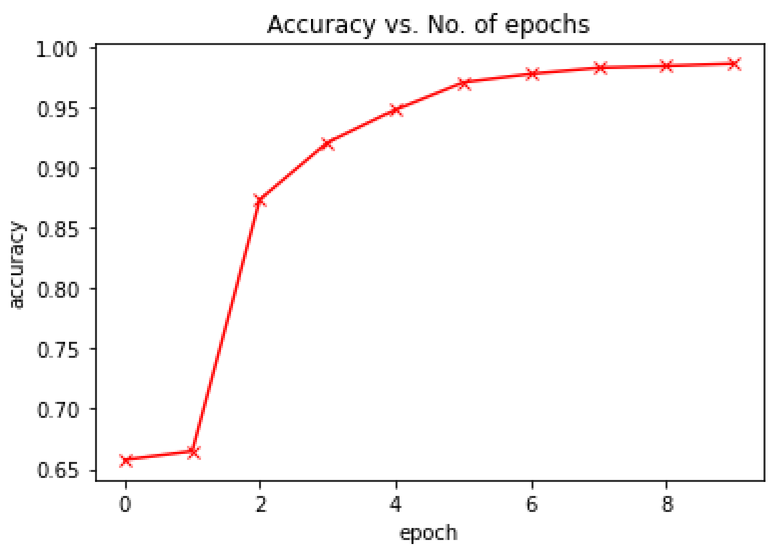
```
num_epochs = 10
lr = 1e-3
history2 = fit(num_epochs, lr, model2, train_loader, val_loader, opt_func
=torch.optim.Adam)
```

```
Epoch [0], train_loss: 0.6439, val_loss: 0.6193, val_acc: 0.6577
Epoch [1], train_loss: 0.5932, val_loss: 0.5346, val_acc: 0.6648
Epoch [2], train_loss: 0.4771, val_loss: 0.4006, val_acc: 0.8739
Epoch [3], train_loss: 0.3529, val_loss: 0.2930, val_acc: 0.9213
Epoch [4], train_loss: 0.2683, val_loss: 0.2162, val_acc: 0.9485
Epoch [5], train_loss: 0.1965, val_loss: 0.1653, val_acc: 0.9713
Epoch [6], train_loss: 0.1548, val_loss: 0.1320, val_acc: 0.9783
Epoch [7], train_loss: 0.1223, val_loss: 0.1087, val_acc: 0.9834
Epoch [8], train_loss: 0.0984, val_loss: 0.0904, val_acc: 0.9849
Epoch [9], train_loss: 0.0817, val_loss: 0.0789, val_acc: 0.9869
```

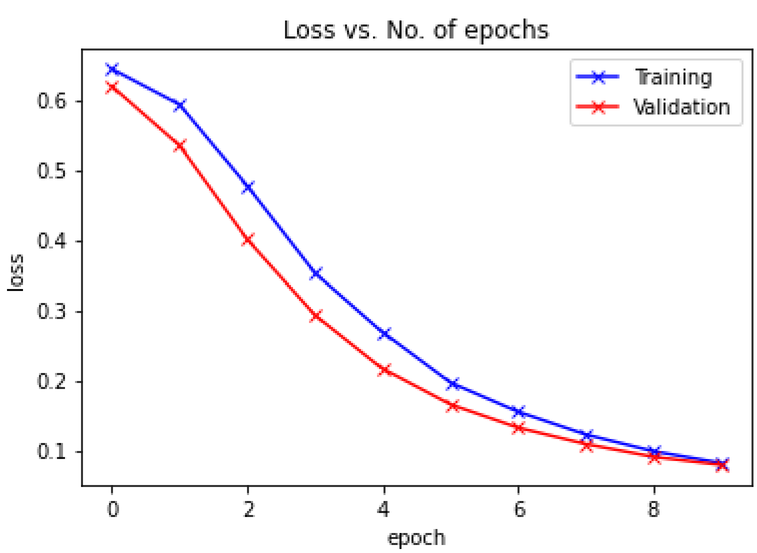
In [115]:
```
train_accuracy(model2)
```

Out[115]:
```
0.9893662929534912
```

In [116]:
```
plot_accuracies(history2)
```



In [117]:
```
plot_losses(history2)
```



In [112]:

```
test_accuracy(model2)
```

Out[112]:

```
0.6792524456977844
```

As we see, a simpler model was able to give us a test accuracy of 67.9% which is a lot better than our previous model. This goes to show that a more complex model might not always be the go-to solution for every task. We could even use other machine learning algorithms like SVM or Decision Trees to come into agreeable accuracy.

Conclusion

There might be a number of reasons why our model failed to perform accurately in the test set. The test set is predominated by positive probabilities, with an uneven distribution of positive and negative candidates. For this reason our model might underperform. Another reason might be the case of overfitting. Simpler model is always better. Maybe by changing and tinkering with the network architecture a bit, we can come up to a decent enough prediction accuracy.

And as evidently shown, a simpler model might be able to generalize better and give better estimations.

Although, this is nowhere close to being an actual prediction modelling for exoplanet search. A much better analysis would be on time-series data or transit curve images using CNN architectures.

Having basically no idea about the deeper intricacies of Astronomy, and only relying on the column descriptions from the official website, it was pretty much a wild guess but the fact that a seemingly random prediction model was able to perform with 53% accuracy and then being able to get a 68% accuracy on the test data with an even simpler model was pretty nice!

If anyone is interested in tinkering with this notebook even more and has domain-specific knowledge as to which columns are more imoprtant in planetary predictions, feel free to fork this and modify it. Thanks!