CoRoT Contributions

Release 1.0

Jul 15, 2020

Reading and Plotting

Here the summary results and developments of the project called A Data Analysis Pipeline for the CoRoT Light Curve Data, supported by NSEE and FAPESP. The pipeline major three specific topics that will allow any user with basic knowledge in Python to reproduce the analysis. The work is called here also by its repository name: *CoRoT Contributions*.

This work fundamentally rely on multidiscipline knowledge, passing by computation, mathematics, signal analysis and phisycs. The main idea is to show how to implement this fields knowledge to solve a practical problem, therefore no introduction level discussion is expected. The focus relies on high level discution on most of those subjects.

In this documentation first an **Introduction** on the CoRoT mission is presented to bring the reader more close to the problem faced by the astronomical community and clearify the importance of the proposed problem. Thence, the **Problem** that will be faced is presented and discussed to introduce the necessary knowledge to interpret the data and understand some decisions taken during the project developments. Finally the data processing **Pipeline** is presented with all the algorithms and analysis necessary to get to the exoplanet classifier presented as the project results.

Finally, all pipeline steps are presented in a detailed manner, by going through all the low-level computations, with code examples. Leading the way to the final results obtained, and reports.

See also:

This documentation is just a guide trough how one can use the provided python libraries do go from the raw data collected from the CoRoT satellite, to a simple architecture to be used on machine learning algorithms. Then, some examples of how to use theses features on machine learning classifiers are presented. If one also wants to check other possible applications please check the GitHub repository.

CHAPTER 1

Introduction to CoRoT

The CoRoT (Convection, Rotation and planetary Transits) was the first mission dedicated and designed for the exoplanetary research. Operated in a lower Earth orbit, has the objectives of using its CCD widefield cameras to obtain and gather light information for the study of supposed exoplanetary behaviors. The project, launched in 2006 had nominal lifetime of 2.5 years, but it actually ended in 2014 when it was de-orbited. The project was led by CNES, with contributions from ESA, Austria, Belgium, Germany, Spain and Brazil.

One of the key information collected by the CCD cameras was the light intensity from the planets, which there is a collection of thousands of light curves observations of different candidates. Interesting enough, this information are actually from exoplanets candidates, and the astronomical community is now working on classifying those observations to map which ones are actually exoplanets from those who are not. There are several methods to detect exoplanets and get information from them, as an example, there are:

- Radial velocity
- Transit photometry
- Relativistic beaming
- Gravitation microlensing

and several others. Here, teh transit photometry approach will be used, and a further explanation on this technique will be presented on the next section. No more will be discussed from the other thecniques, but the reader is free to check it out how those work. The only interesting subject that most of those has in common is that their collected information are time series (dynamical information), and most of the techniques presented in this study could be used similarly for all of them.

For more reference on works regarding the classification of the light curves observations one might check the following M. Deleuil1, S. Aigrain2, et al.

For more details on the CoRoT mission please check out the ESA CoRoT site.

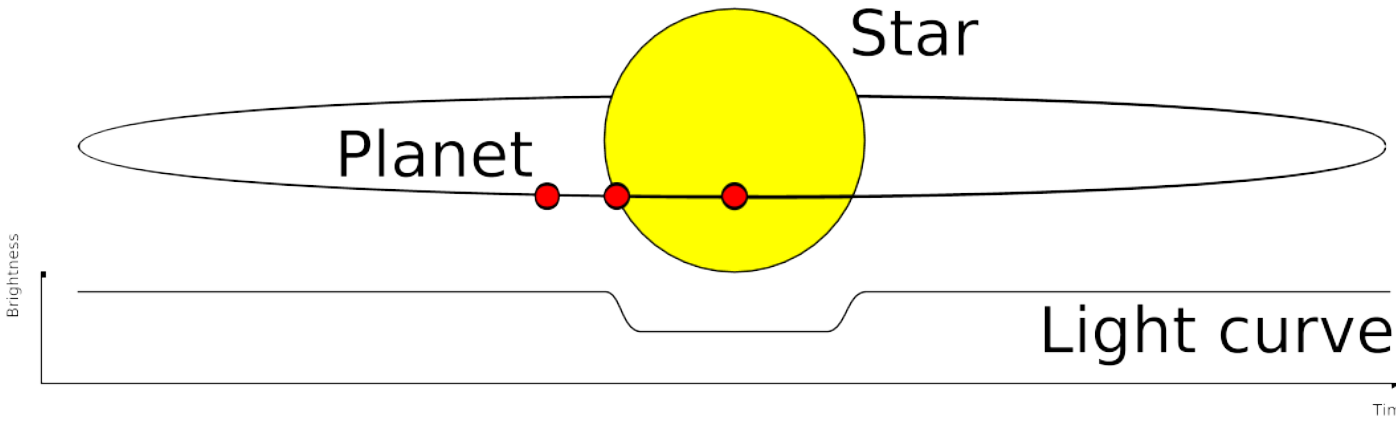
CHAPTER 2

Problem proposed

Being as concise as possible, the main goal is to use machine learning to automate the classification of CoRoT collected data, as a way to cluster data that has meaningful information to be analyzed from data that has none. This means that we want to classify data that present possible exo-planets, from data that has information about something other then exo-planets.

To provide a brief introduction to the knowledge needed to achieve this goal, we will describe the data set a little bit. The data set is provided by the repository provided at the web site , where it contains files in a *.fits* format. Those are pretty common in the astronomical whereabouts, but not for the machine learning public. In each *.fits* file, we will have a light curve time series for a particular CoRoT observation. All observations will have the white light intensity, some will have the RGB.

Therefore, we are talking about using the transit photometry time series to classify if in this particular observation there is a chance for the observation be one of an exo-planet, or something else. As an illustration, one will have time-series data, that will contain the planetary transit information such in



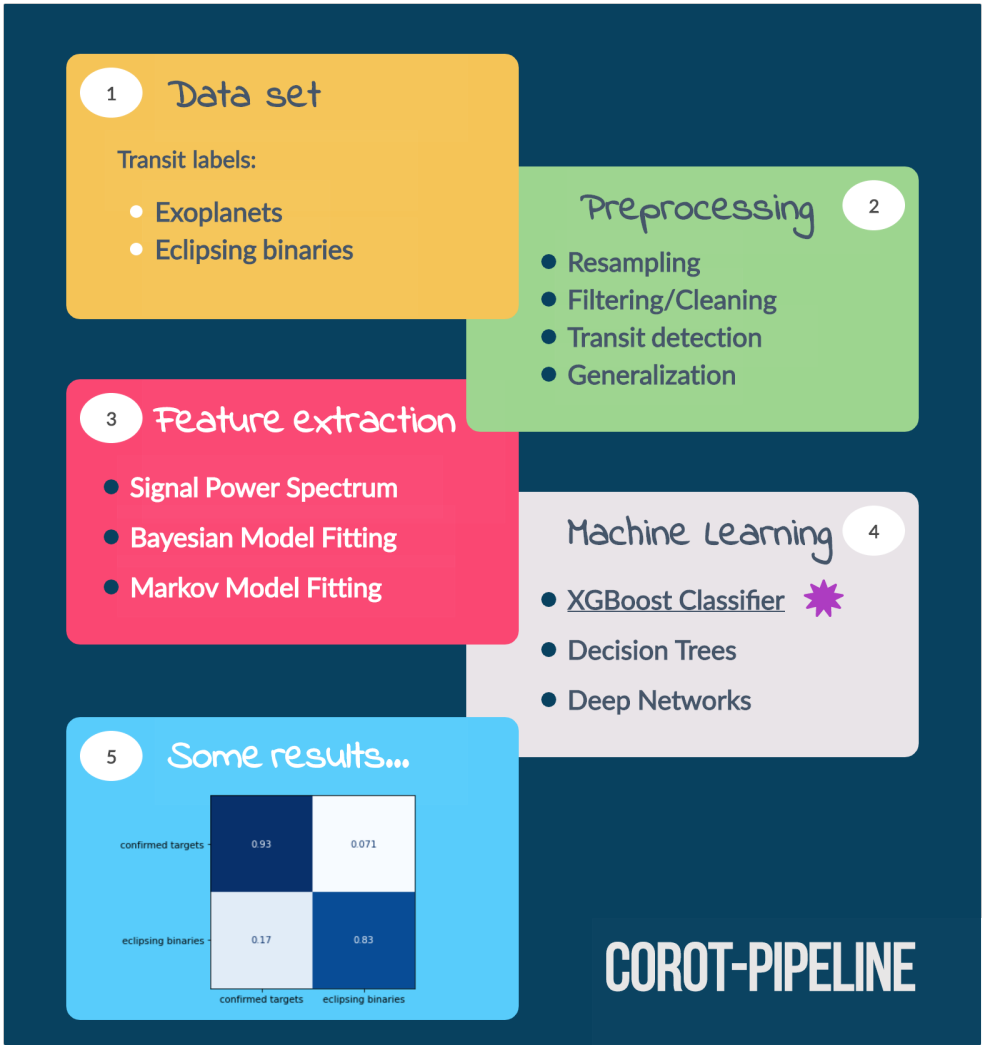
So the features for the machine learning algorithm must be gathered from the time series, and the algorithm must reach for the probability of this observation be the observation of an exo-planet.

CHAPTER 3

Pipeline

The preprocessing pipeline consists on four major steps:

- Getting the data - (10%)
- Preprocessing in Python - (20%)
- Feature Engineering - (50%)
- Machine Leaning - (20%)



Notice

In these items, we introduced the usual percentage of the work amount usually taken for each particular part of a machine learning project. This is usually the amount when you have features that do not involve time, dynamic features... therefore static information, which is mostly used to happen, such as in problems of customer clustering, when one has features such

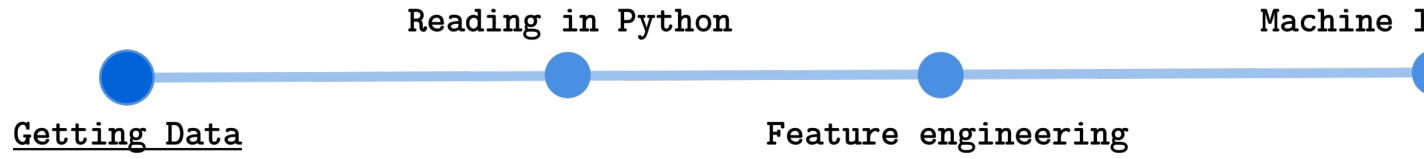
- Age
- Salary
- Address
- ...

Note that these features do not change over time. Here we have a time series, so we have to somehow extract from those time series, some static information. So be prepared for the feature engineering part of the pipeline!

Warning: Note that the first three pipeline steps will generate a data file, with the processed data for the next step. The user can choose to either follow each process step and by it self generate and preprocess the data. Or one can go the google drive link bellow

- Google drive access

and download the already preprocessed data. For example, if the user wants to jump all the three first steps and just have access to the features for the machine learning algorithm, one can download then from the folder */features* in the google drive.



To get the data on your local machine, first one mush address to the data repository web site where we will have the soo called level N2 processed data from the CoRoT observations. Those are the most processed versions of the CoRoT collected data. At this repository, one will find five classes of observations,

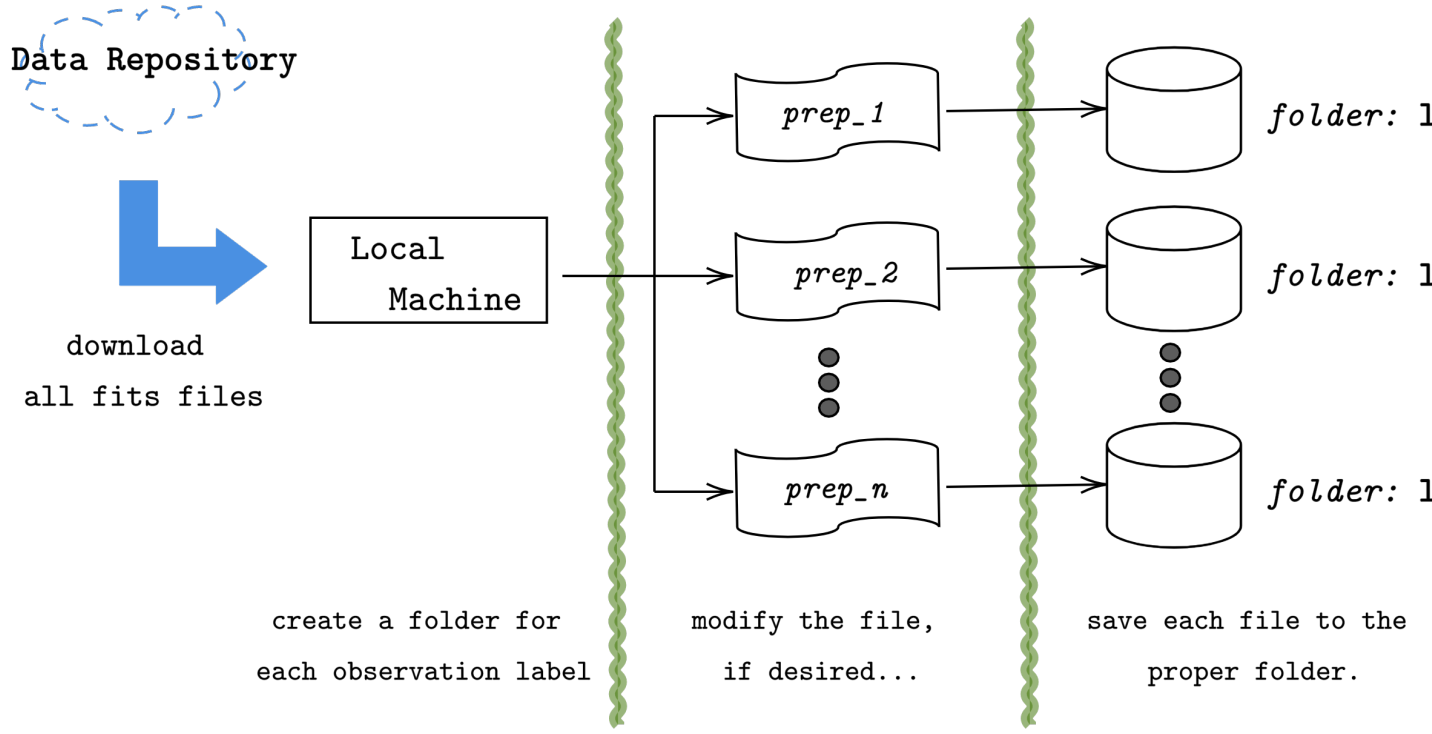
- CoRoT bright: the called bright stars (examples of not exo-planets)
- CoRoT targets: the called exo-planets (examples of exo-planets)
- Red giants: the observations classified as red giants (examples of not exo-planets)
- Stars with rotation periods from Affer (not used)
- Stars with rotation periods from Medeiros (not used)

The last two classes will not interest us. We actually will only use the first three collection of data. Those collections have information of transit photometry or only light curve intensity, furthermore their labels allow us to segregate the information as not exo-planet and exo-planet. While the last two, provide labels that cannot ensure if the collected data is from an exo-planet observation or not.

If the user select one of the three categories, it will show a table with several items. Each item is a particular light curve observation, and the user can select and download any of the curves that he desires.

After that one might also download another class of light curve observations called eclipsing binaries from the data repository in the Query Forms tab in the FAINT STARS option. There the user will be able to query each curve from their specific CoRoT Id. Then by searching the stars CoRoT Id from the tables at the CoRoT transit catalog, it is possible to reach a group of close to 1500 eclipsing binaries.

Note: One interesting aspect of the eclipsing binaries is that their light curves are pretty close to the exo-planets ones. Therefore it is pretty difficult to cluster eclipsing binaries from exo-planets using only the light curve. This is the most interesting case study of this project.



After downloading for the three classes (bright stars, confirmed targets and red giants) the user must keep each class in a particular folder, with the name that the user wants to label each class. As schematized in the figure above.

It is advised to keep a database folder following the structure:

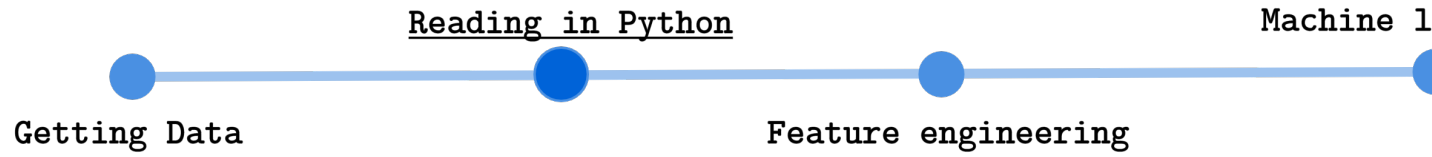
```
./database
    ├── bright_stars
    │   ├── ..._1.fits
    │   ├── ..._2.fits
    │   └── ..._k.fits
    ├── confirmed_targets
    │   ├── ..._1.fits
    │   ├── ..._2.fits
    │   └── ..._l.fits
    ├── eclipsing_binaries
    │   ├── ..._1.fits
    │   ├── ..._2.fits
    │   └── ..._m.fits
    └── red_giants
        ├── ..._1.fits
        ├── ..._2.fits
        └── ..._n.fits
```

Drive files access

If the user does not want to go through the process presented above and just want to download the data, already in the format presented above, it is possible to get it from my google drive, or in the direct download link:

- Google drive access
- Direct download

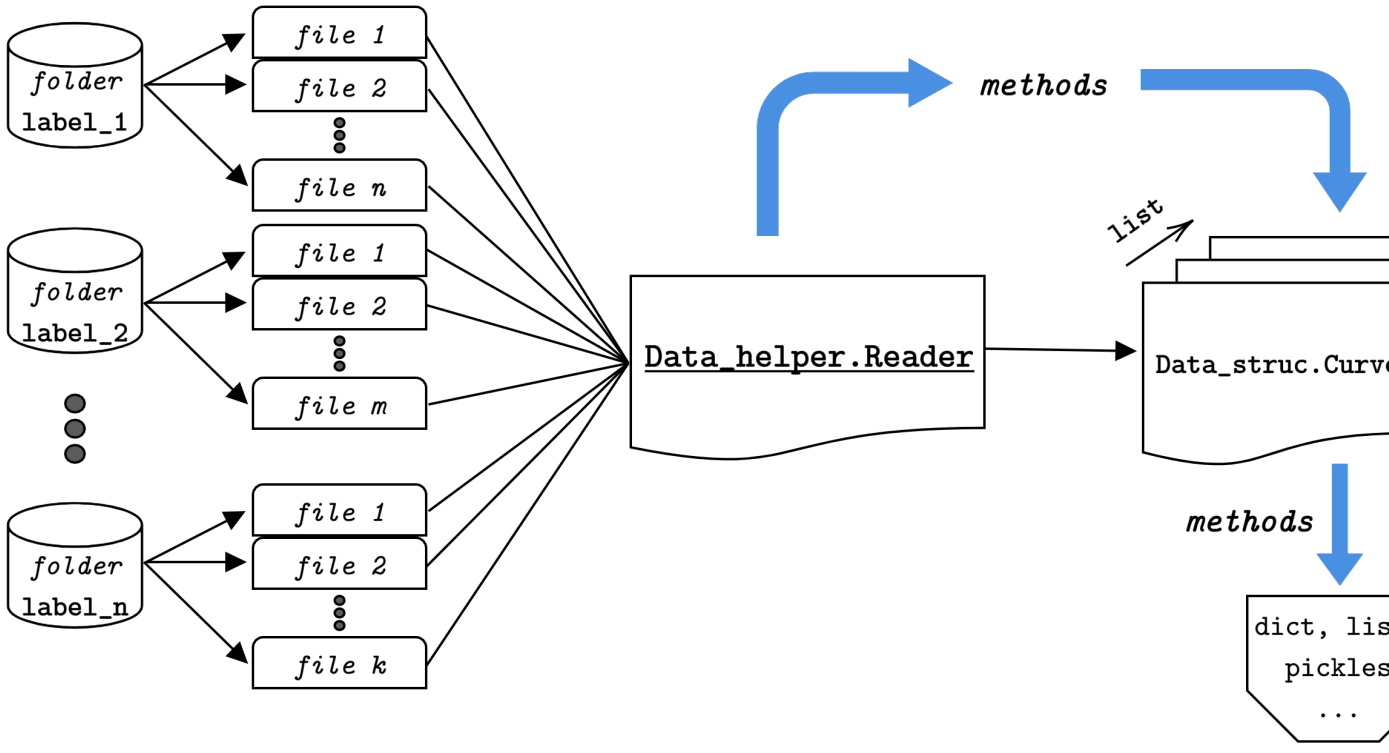
The files are something close to tens of GB. Since it contains all the raw fits files for the three classes (bright stars, red giants, confirmed targets and eclipsing binaries).



After getting the data from the online repositories, we need to transform this data into data structures more python friendly, such as dictionaries, lists, arrays, or into files with more suitable formats then *fits* files, *e.g.* pickle files, or MATLAB files (if one wants to algorithms on MATLAB).

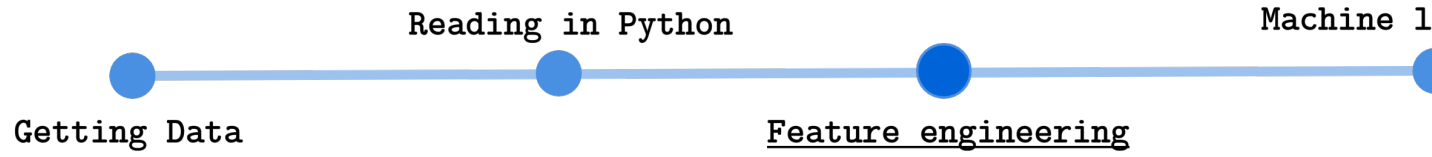
For that, we create here a preprocessing library called `Reader` that transform the data from the fits files into simple objects of the class `Curve` that simplifies the information access, and make it faster, provided that it works with the so-called memory maps and only save HUD tables that will be necessary for the future. This process reduces the dataset up to 80% (the data that was something close to 40 GB, is now something close to 7 GB).

The next scheme illustrates the process applied to the data to create this interface from the *.fits* format into the python friendly form:



From the above scheme, we can see that the process starts from the *fits* files and after passing through the `Reader`, we achieve a structure that with simple calls to the information variables, it is possible to get that variable data in *dict* and/or *list* formats.

After reading the data, and before saving in a format such as the pickle file, we also apply a preprocessing technique to filter out the noise of the time series. This technique is applied here instead of in the feature engineering step, just to show the user the importance to understand the feature processing part with an information extraction practical example. Showing that a simple filtering technique can enable access to several other pieces of information. And by that, engaging the user to see the next step of the pipeline, which will have several other feature engineering techniques.



The feature engineering process is the part that takes most work in a machine learning project. It is the part where one must extract meaningful information from the data to provide for the machine learning algorithms. But interesting enough, one cannot know beforehand what information is actually the best one for the machine learning algorithm. Most machine learning projects, the data is actually static (does not depend on time) and have a simpler path to be taken, where the only usual procedure is done in the feature engineering is the common normalization and feature reduction (when the dimension of the features are very high and the learning algorithm is pretty heavy). But in the case of time series classification, the user has dynamic data... which needs to be transformed to static data, to then go through the common feature engineering processes.

There are several paths that one might take to get static information from the data when one has a time series in the hands. Some of the techniques are presented:

- Power spectrum
- Periodograms
- Number of peaks
- Dynamic insights

The first two, are not much than the Fourier representation of the data, which is actually the most characteristic information of the time series. The second is just commonly used info in the internet tutorials (which is a very noisy scenario, which will only generate more noisy data). The last one usually is a very powerful approach, in which, we first use the data to estimate a dynamical model of the time series phenomenon, and then use that model parameter to cluster each time series. The most interesting part of this approach is that it relies on the leaning algorithm focus to enhance certain behaviors of the dynamic model and mitigate the effects of others.

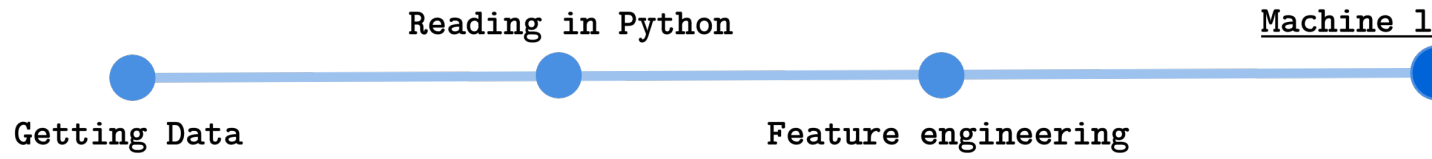
Usually, people abuse simple least-squares or deterministic techniques to acquire the parameters of a particular structure model. Here we will present a stochastic approach of this technique, where we will estimate stochastic and probabilistic models for the time series, and their parameters will then be used as features for the machine learning process.

Here the bellow features will be generated for the machine learning algorithm:

- Periodograms
- Bayes marginal likelihood
- Hidden Markov Models
- Latent Dirichlet Allocation

The first one is the classical technique (which present several concerns and computation problems, due to the Fourier algorithms). The second one uses probabilistic Bayes Models parameters as features. The third uses the estimated parameters for Hidden Markov Models as features. And the fourth one actually is a very complex technique usually used to cluster text into speech topics, and here is used outside its scope, to check if the topic's classification also can be applied to time series dependencies.

Note: Provided that each approach is more unique than the other, here it will not be discussed further more about any one of them. A more detailed description will be presented in the Feature Engineering step of the documentation.



The machine learning step of the pipeline, does nothing much more than the basics machine learning procedures once the time series static information is provided. From the last step, simple features where build from the time series. using the following techniques:

- Periodograms: produces the evenly reshaped power spectrums
- Bayes marginal likelihood: produces the Ridge Bayesian model parameters
- Hidden Markov Models: produces the Markov trasition matrices

And now it is desired to build a machine learning algorithm that will take each one of those features and try to classify the observation classes. Therefore, we just want to distinguish between the four labels:

- Red giants
- Confirmed targets or exo-planets
- Bright stars
- Eclipsing binaries

The astronomical community show a particular interest in distinguish specifically two of those classes from each other: the confirmed targets from the eclipsing binaries. This challenge show itself to be appealing because there is several difficulties in distinguishing the eclipsing binaries and exo planets light curve information using classical theory. Therefore, having an automate algorithm capable of solving this enigma would be most attractive.

Thence here we present three major algorithms to classify the data from exo-planets and eclipsing binaries. The first is using the XGBoost library, since it is proven to be a very powerful algorithm in machine learning forums and competitions, such as Kaggle. Also, some classic algorithms will be used, such as Decision Trees from the scikit learn packages to compare the boosting feature capability from the XGBoost algorithms, and some powerful mathematical algorithms such as Support Vector Machines classifiers, which is supposed to be analytically optimal.

All the algorithm will follow the common machine learning path, such as:

- Read the features and labels
- Encode, normalize, reduce features and reshape
- Make train and test data-set
- Build the machine learning model
- Search hyper parameters

- Train machine learning model
- Validate the model
- Analyse the results

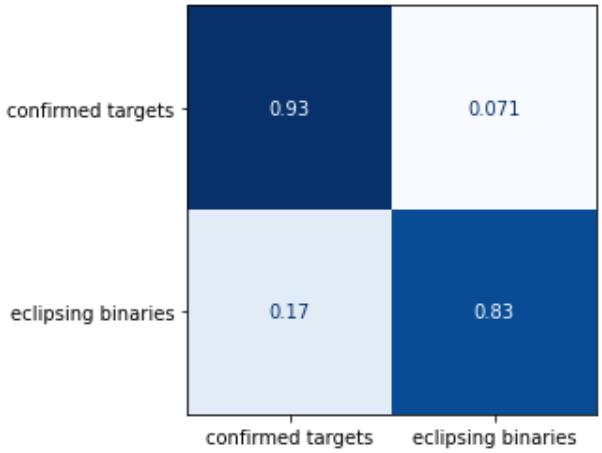
So the reader can understand that for any of the models and any of the features, after the Feature Engineering pipeline, the algorithm is pretty much the same. Therefore, the price of dealing with time series data, is that one must have the Feature Engineering pipeline to extract static information from the data, before going through the common machine learning pipeline.

Note: As in the previous pipeline step, the algorithms will not be further discussed here. If the reader wants, a more detailed description of each machine learning approach can be found on its respective chapter of the documentation, with application example.

CHAPTER 4

Some results. . .

To show the capability of the approach presented, checkout the some of the best results obtained with the XGBoost Classifier algorithm:



In this figure, the confusion matrix of the classifier predictions for the testing dataset is presented. Notice that the XGBoost algorithm, was able to provide a **93%** accuracy on **exoplanet labels**, and **83%** on the **eclipsing binaries**, as shown by the above confusion matrix.

4.1 Read *.fits* raw data

For reading the *.fits* data-set one may use the utils library. From the `utils` library, there is a `data_helper.Reader()` object, optimized to read the *.fits* files. Inside a *.fits* file, usually is found a sctructure with three tables of type HUD. In this particular case, the three tables teels a *history* of the data, where the first is the most raw data possible, then the second is the treated/filtered version of the first table, and the third table is the compressed version of the information from the second table.

> *We are particularly interested in the most informative data-set in a minimal (compressed) manner. Also, here we do not have the luxury of dealing with big-data problems... for that we might need **Spark** or* **Hadoop** support. Thence, the third table is usually the most interesting one.*

All files from a given folder will be readed and labeled as disered. As an example an advised folder structure is the following:

```
./database
    |
    ├── bright_stars
    |       ├── ..._1.fits
    |       ├── ..._2.fits
    |       |
    |       └── ..._k.fits
    |
    ├── confirmed_targets
    |       ├── ..._1.fits
    |       ├── ..._2.fits
    |       |
    |       └── ..._l.fits
    |
    ├── eclipsing_binaries
    |       ├── ..._1.fits
    |       ├── ..._2.fits
    |       |
    |       └── ..._m.fits
    |
    └── red_giants
            ├── ..._1.fits
            ├── ..._2.fits
            |
            └── ..._n.fits
```

Inside the `database/bright_stars` it has all *.fits* files of class *bright stars*, in `database/confirmed_targets` it has all *.fits* files of class *confirmed exoplanets*, in `database/eclipsing binaries` it has all *.fits* files of class *confirmed multi transit eclipsing binaries* and in `database/red_giants` it has all *.fits* files of class *red giants*.

All the provided folders are then readed and the data is concatenated into one big list of `data_struc.Curve` objects, wich is a pretty helpful interface structure from the `astropy.table` objects to simple `list` and `dict` variables. Thence, in the variable `curves` we actually have a `list` of `data_struc.Curve`.

[1]:
```
from utils import *

folder_list = [ './database/raw_fits/confirmed_targets',
                './database/raw_fits/eclipsing_binaries',
                './database/raw_fits/red_giants',
                './database/raw_fits/bright_stars' ]

dread = data_helper.Reader()
curves = dread.from_folder(folder=folder_list[0], label='confirmed targets', index=2)
curves += dread.from_folder(folder=folder_list[1], label='eclipsing binaries',␣
↪index=2, items=40)
#curves += dread.from_folder(folder=folder_list[2], label='red giants', index=2)
#curves += dread.from_folder(folder=folder_list[3], label='bright stars', index=2)
```

(continued from previous page)

Data type cannot be displayed: application/javascript, application/vnd.bokehjs_load.v0+json

```
INFO:reader_log:Reader module created...
INFO:reader_log:Reading 37 curve packages...
INFO:reader_log:Done reading!
INFO:reader_log:Reading 40 curve packages...
INFO:reader_log:Done reading!
```

Importing the `utils` as shown, one is actually import all the following packages:

- `data_helper` with Reader Object
- `data_struc` with Curve Object
- `visual` with several plot functions

And as an example, the visual package just compress the bokeh code library, since it is a pretty expansive code, for example, to make a line plot without visual, one should do something like

```
from bokeh.palettes import Magma
from bokeh.layouts import column
from bokeh.plotting import figure, show
from bokeh.models import ColumnDataSource
from bokeh.io import output_notebook, push_notebook

p = figure(
    title="Some title",
    plot_width=400,
    plot_height=600)

# Style the figure image
p.grid.grid_line_alpha = 0.1
p.xgrid.band_fill_alpha = 0.1
p.xgrid.band_fill_color = Magma[10][1]
p.yaxis.axis_label = "Some label for y axis"
p.xaxis.axis_label = "Some label for x axis"

# Place the information on plot
p.line(x_data, y_data,
        legend_label="My legend label",
        line_width=2,
        color=Magma[10][2],
        muted_alpha=0.1,
        line_cap='rounded')
p.legend.location = "right_top"
p.legend.click_policy = "disable"

show(p)
```

and the same can be achieved using `visual`, just as follows

[2]:
```
p = visual.line_plot(curves[25]["DATE"],
                     curves[25]['WHITEFLUXSYS'],
                     legend_label='Raw Light Curve',
                     title='Example curve plot',
                     y_axis={'label': 'Intensity'},
                     x_axis={'type': 'datetime',
```

(continues on next page)

(continued from previous page)

```
                                'label': 'Date'})

visual.show_plot(p)
```

Data type cannot be displayed: application/javascript, application/vnd.bokehjs_exec.v0+json

4.2 Preprocessing data

The preprocessing part of this analysis will include the preparation of the light curve data for each observation and saving both the original data, such as the preprocessed data in a more suitable data format for Python users. The main goal is to read the data from the *.fits* file (already done with `data_helper.Reader()`), filter the data to remove dynamical noise, and than save in a suitable format such as *.pickle*, or *.mat* for those in favor...

Here is the step where we will filter the data. To do that, first we must choose between the two possible paths

- Continuous time filters
- Discrete time filters

> *A practice advice, it is always preferable discrete time filtering, because the routines are simpler and more efficient... but this is not always possible when dealing with several time series. Fortunately, here it is possible! To go with the discrete approach, we must check if all time series has a close mean sample time, with low variance. If not, we must first resample the time series, and then use discrete filtering techniques on the data. Do not try, to apply discrete filtering techniques on analog data (without a consistent sample time), otherwise you will apply a technique that is not a linear transfomation on the data, and therefore, you will be, by hand, introducing noise to the data.*

So first, lets check the sample time of the series, and after filter the high frequency noise from the data. The usefulness of the filtering of the time series, will be shown in the final of the document, where a feature extraction technique will be applied just as an example on how the denoised time series is so necessary.

4.3 Resampling series

4.3.1 Sample time distribution

To analyse the time sample of the series, one can do a box plot for each time-series curve. At each box plot it is represented the distribution of the diference `t[k] - t[k-1]` for `k` representing each sample of the time-series and `t` the sampled time.

[3]:
```
import numpy as np
from datetime import datetime

item = 0
values, labels = [], []
for curve in curves:
    diff_time = np.diff(curve["DATE"])
    values += [x.total_seconds() / 60 for x in diff_time]
    labels += [str(item) for x in diff_time]
```

(continues on next page)

(continued from previous page)

```
    item += 1
                                #    Include comment here
                                #          to enhance the outliers
p = visual.box_plot(values, labels,   y_range=(8.515, 8.555),
                    title='Average sample time within curves',
                    y_axis={'label': 'Sample time (min)'},
                    x_axis={'label': 'Curves'})
visual.show_plot(p)
```

Data type cannot be displayed: application/javascript, application/vnd.bokehjs_exec.v0+json

4.3.2 Resample time series data

Since the time sample time do not vary that much from one light curve to another, **not considering the outliers**... One can create the sample time as the mean of the median and then resample each time series using this found value.

> *One also might say that those time series, does not need resampling, since all the series present a close sample time mean. But notice that the above figure has the ''y'' axis clipped from ''[8.515, 8.555]''. If the user comment the line described on the above cell, one will see that there are some worrying outliers. **It is because of those outliers that a resampling technique must be applied!***

[4]:
```
import statistics as stt

time_sample = round(stt.mean(values), 2)

print("The time sample is now {} minutes.".format(time_sample))
```

```
The time sample is now 8.56 minutes.
```

Now that we have a feasible approximation of the time sample, it is possible to use this reference to reseample each time series to this single sample rate.

[5]:
```
import scipy.signal as scs

data = {'y':[], 't':[], 'ti':[], 'lab':[]}
for curve in curves:
    # Get the time information
    init_time = curve['DATE'][0]
    data_vector = curve['WHITEFLUXSYS']
    time_vector = [(t - init_time).total_seconds() / 60 for t in curve['DATE']]
    # Compute the amount of points
    n_points = int(time_vector[-1] // time_sample)
    # Compute the resampled time series
    res_data, res_time = scs.resample(data_vector, n_points, time_vector)
    data['y'].append( res_data )
    data['t'].append( res_time )
    data['ti'].append(init_time)
    data['lab'].append(curve.labeler)
```

To check if the new time sample was correctly placed and there is no more samples with outlier time samples. One can use the histogram of the time sample variation of all light curves to ensure the consistency of the sample time.

[6]:
```
t_std = [stt.stdev(np.diff(t)) for t in data['t']]
hist, edges = np.histogram(t_std, density=True, bins=3)

p = visual.hist_plot(hist, edges,
                     title='Sample time consistency',
                     y_axis={'label': 'Density'},
                     x_axis={'label': 'Sample time difference (min)'})

visual.show_plot(p)
```

Data type cannot be displayed: application/javascript, application/vnd.bokehjs_exec.v0+json

4.4 Filtering series

The main goal here is to remove the random signals that are contributing to the time series information, the objective is actully to clean the time series from highly spread random variables. This is pretty interesting, because the applied filtering technique will not disturb the meaningful information of the series, since it will only mitigate the random information effects on the series. So lets see the time series:

[7]:
```
index = 5          # Curve index

p = visual.line_plot(data['t'][index],
                     data['y'][index],
                     legend_label='Raw Light Curve',
                     title='Light Curve',
                     y_axis={'label': 'Intensity'},
                     x_axis={'type': 'datetime',
                             'label': 'Date'})
visual.show_plot(p)
```

Data type cannot be displayed: application/javascript, application/vnd.bokehjs_exec.v0+json

One can see that the data is too noisy, from the low variant part of the signal... To reduce the amount of noise on data we might use the `PyAstronomy` library that has some interesting smoothing algorithms, *e.g.* the *hamming* filter that will be used following. We can also change the window size considered for filtering the light curve, here as an example we are using `window = 33` samples.

> *Note that we are correcting the grammar and talking about smothing and not just filtering the data. In smothing techniques, it is spected that the provided data is in a discrete format. Because, actually, a mathematical filtering is applied... not a frequency filter. We are trying to reduce the influence of random variables with highly spread frequency behavior from the data.*

[8]:
```
from PyAstronomy import pyasl

window = 33
algorithm = 'hamming'
sm1 = pyasl.smooth(data['y'][index], window, algorithm)

p = visual.line_plot(data['t'][index],
                     sm1,
```

(continues on next page)

(continued from previous page)

```
                      legend_label='Raw Light Curve',
                      title='Light Curve',
                      y_axis={'label': 'Intensity'},
                      x_axis={'label': 'Time index'})
visual.show_plot(p)
```

Data type cannot be displayed: application/javascript, application/vnd.bokehjs_exec.v0+json

Now we are talking!! Without the noise is possible to observe the actual variation of the time series, when the eclipses appear. To see the importance of the filtering process applied above, let's try to do some feature engineering... and use the data to extract some information of the process.

4.5 Application example

We know, that the series from `database/confirmed_targets` are series that highly represents the transit photometry, since it has pronunced eclipses. So let's try, to use the time series data to create an algorithm to determine the moments of eclipse on the time series.

The main idea is to create a binary label for the eclipse pattern from the light curve. For that, we could check the light curve derivative to analyse the time series variation along time. Usually when you have a high variation, the chances of having a eclipse is bigger. As an example, let's plot the derivative of one particular light curve, `index = 2`. So for that, lets plot both the derivative of the non filtered time series, and the filtered one, to see if we can take anything out of these informations.

[9]:
```
p = visual.line_plot(data['t'][index][1:],
                     np.diff(data['y'][index])/time_sample,
                     legend_label='Derivative of Light Curve',
                     title='Light Curve Derivative',
                     color_index=4,
                     y_axis={'label': 'Intensity'},
                     x_axis={'type': 'datetime',
                             'label': 'Date'})
p1 = visual.line_plot(data['t'][index][1:],
                      np.diff(sm1)/time_sample,
                      legend_label='Derivative of Light Curve',
                      title='Light Curve Derivative',
                      color_index=4,
                      y_axis={'label': 'Intensity'},
                      x_axis={'label': 'Time index'})
visual.show_plot(p, p1)
```

Data type cannot be displayed: application/javascript, application/vnd.bokehjs_exec.v0+json

From these graphs it is possible to see, that with a simple threshold selection it is possible to infer the regions where we have the eclipses, in the filtered derivative version.

> *More practice guide... When dealing with time series, the most simple linear transformation (e.g. a derivative) could enhance the data noise in a very powerfull way. Interesting enough, the noise didn't seem to be that big on the time series plot before right?! So after appling a simple linear trasnformation, the noise increased to be bigger to the biggest actual value of the data. Note the amplidutes of the highest*

> *value on the filtered version, and the not filtered one... in the not filtered version, one can only see noise!!*
> *> So imagine if you pass this not filtered data trough a complex neural network, that will apply several non linear transformation to your data... we are talking about CAOS! Even if you think you have interesting results, your algorithm is actually working in a very sensible place, that eventually he will go unstable.*
>
> *Another topic to realise... the most treatening noise that one can have in a time series is this called white noise. This noise exists in all frequencies and only statistical and mathematical techniques that obey the law of large numbers can deal with it. And for that, you must have discrete time data!*

Then, to introduce some statistics perspective to the result, and make it more automatic, the mean and standard deviation of the light curve variation is used to determine possible decision thresholds, that could infer the moments of initialization of the eclipse, and the finalization...

For that, lets compute the mean of the derivative and the standard deviation

[10]:
```
variation_mean = np.average(np.diff(sm1)/time_sample)
variation_stan = np.std( np.diff(sm1)/time_sample )

print("The varaition signal has a mean around {}".format(round(variation_mean,3)))
print("And a standard deviation around {}".format(round(variation_stan,2)))
```

```
The varaition signal has a mean around 0.035
And a standard deviation around 8.4
```

Therefore, one can create a threshold close to $\pm\sigma$, $\pm 2\sigma$ and $\pm 3\sigma$, as one can see in the next bellow figure. This thresholds will inform if there was a big varaiation of the time series.

[11]:
```
size = data['t'][index][1:].shape[0]
one_std = variation_stan * np.ones((size,))

x_data = data['t'][index][1:]
y_data = [np.diff(sm1) / time_sample,
          1*one_std, 2*one_std, 3*one_std, -1*one_std, -2*one_std, -3*one_std]
legends= ['Derivative', '68.0%', '95.0%', '99.7%', '68.0%', '95.0%', '99.7%']
colors = [8, 1, 3, 5, 1, 3, 5]

p = visual.multline_plot(x_data, y_data,
                         legend_label=legends,
                         title='Light Curve Derivative',
                         color_index=colors,
                         y_axis={'label': 'Intensity'},
                         x_axis={'label': 'Time index'})
visual.show_plot(p)
```

Data type cannot be displayed: application/javascript, application/vnd.bokehjs_exec.v0+json

Now one can create a `listener` to check each peak and create the respective eclipse label. First it is necessary to check two states: the first is the state of `in_eclipse` and the second the `out_eclipse`. Which will map when the series goes into one eclipse, then out of the eclipse.

[12]:
```
trigger, out_eclipse = False, True # because it starts out of the eclipse
light_variation = np.diff(sm1) / time_sample
light_variation = light_variation.tolist()
threshold = [-1*variation_stan, 1*variation_stan]

light_state = [False]
```

(continues on next page)

(continued from previous page)

```
size = len(light_variation)
for k in range(size):
    if out_eclipse:
        if light_variation[k] < threshold[0]:
            out_eclipse = False
    else:
        if (light_variation[k-1] > threshold[1]) \
            and (light_variation[k] < threshold[1]):
            out_eclipse = True
    light_state.append(out_eclipse)
```

With those results, lets see if we can plot the light curve and highlight the moments where we have a suposed eclipse.

[13]:

```
in_eclipse = np.ma.masked_where(np.array(light_state), sm1)
out_eclipse = np.ma.masked_where(~np.array(light_state), sm1)

x_data = data['t'][index]
y_data = [in_eclipse, out_eclipse]
legends= ['In eclipse', 'Out eclipse']
colors = [3, 7]

p1 = visual.multline_plot(x_data, y_data,
                          legend_label=legends,
                          title='Light Curve Derivative',
                          color_index=colors,
                          y_axis={'label': 'Intensity'},
                          x_axis={'label': 'Time index'})
visual.show_plot(p1, p)
```

Data type cannot be displayed: application/javascript, application/vnd.bokehjs_exec.v0+json

Checkpoint!! One thing that we also need to do, is change the `index` variable and check each exoplanet curve and see if we could ensure that this algorithm works for most of then. And also with the bright stars and red giants...

After that we are ready to engage on more complex analysis, such as statistical approaches and machine learning techniques!

> ***Notice one interesting thing: using the derivative approach, the steady state information of the light curve is automatically discarded! This is a hell of deal, since it is already a filter that mitigates the influence of low frequency siganls and highlight the effect of high frequency ones!!!! O.o Crazy!! Since we only want to analyse the bahavior of the high descent variation, when there are eclipses, the signal derivation is the approach that most highlight this feature. :)***

4.6 Generation algorithms

Here, the algorithm presented above is implemented for each time serie curve and then we generate a file with the pre-processed data. So for that we need to run the following the next procedure. Since all time series are already resampled, only the procedure of filtering and labelling are necessary to be made for all time series. This is the function, that provided the resampled time series, it returns a the filtered and labeled data:

[14]:
```
def filter_series(data=None, window=33, algorithm="hamming"):
    '''
        This is the function used to filter all time series readed
        in a batch process. And it could take some time to run...
    '''
    filtered_curves = {'r':[],'y':[],'t':[],'i':[], 'lab':[]}
    for curve, time, init in zip(data['y'], data['t'], data['ti']):
        filt_curve = pyasl.smooth(curve, window, algorithm)
        filtered_curves['r'].append( curve )
        filtered_curves['y'].append( filt_curve )
        filtered_curves['t'].append( time )
        filtered_curves['i'].append( init )
    filtered_curves['lab'] = data['lab']
    return filtered_curves
```

[15]:
```
filtered_data = filter_series(data=data)
```

Now that we have the filtered data, we could use the derivative algorithm to label each time series...

[ ]:
```
def label_series(data=None, time_sample=None, std_num=3):
    '''
        This is the function used to label the eclipses part of
        the time series, using the filtered lgiht curve data in
        a batch process. And it could take some time to run...
    '''
    data['eclipse_labels'] = []
    for curve in data['y']:
        derivative = np.diff( curve ) / time_sample
        mean, stan = np.mean( derivative ), np.std( derivative )
        derivative, threshold = derivative - mean, std_num * stan
        light_state, out_eclipse = [False], True
        for k in range(len(derivative)):
            if out_eclipse:
                if derivative[k] < - threshold:
                    out_eclipse = False
            else:
                if (derivative[k-1] > threshold) \
                    and (derivative[k] < threshold):
                    out_eclipse = True
            light_state.append( out_eclipse )
        data['eclipse_labels'].append( light_state )
    return data
```

[ ]:
```
filtered_data = label_series(data=filtered_data, time_sample=time_sample)
```

Know we have some structure data in the `filtered_data` variable that is actually pretty suitable for Python users, which is composed only by `dict`, `list`, `array` and `datetime`. And follows this particular structure:

```
{
    'r': [
        array(...),
        array(...),
```

(continues on next page)

(continued from previous page)

```
        array(...)
    ],
    'y': [
        array(...),
        array(...),

        array(...)
    ],
    't': [
        array(...),
        array(...),

        array(...)
    ],
    'i': [
        datetime,
        datetime,

        datetime
    ],
    'lab': [
        str,
        str,

        str
    ],
}
```

where each key is:

- `r` : The raw intensity of each curve
- `y` : The filtered intensity of each curve
- `t` : The time index initialized in 0 of each curve
- `i` : The time of the first sample of each curve
- `lab`: The string with the label of this curve

As an example, to fetch the filtered intesity samples of the third curve, one just:

```
data = filtered_data['y'][2]
```

4.7 Save pre-processed data

4.7.1 Save as *.mat* file

This is just a function that will save the filtered data as a MATLAB data file:

```
[ ]: import scipy.io as sio

file_path = './filtered.mat'
```

(continues on next page)

(continued from previous page)

```
sio.savemat(file_name=file_path, mdict=filtered_data)
```

4.7.2 Save as *.pickle* file

This is a fucntion to save the data as a *pickle* file to save the dictionary:

[ ]:
```
import pickle

file_path = './filtered.pkl'

output = open(file_path, 'wb')
pickle.dump(filtered_data, output)
output.close()
```

4.8 Reading the data

To go trough with this example, one might have or a

- *filtered.pkl* - pickle format file
- *filtered.mat* - matlab format file

as generated in the previous step of the pipeline. Note that, we are now ready to generate features for the machine learning algorithms. The biggest challenge is to get static features out of the time series (wich is a dynamic data). Here we will approach three main static feature generation paths:

- Frequency analysis with periodograms
- Naive Bayes likelihood parameters
- Markov transition probability matrix

All of these three features will be further used as static data for the machine learning algorithms to learn how to classify the light curves as each class. So first, it is necessary to read the preprocessed/filtered data from the last pipeline step:

> *Note that we must have the data preprocessed and labeled as exo-planets and not exo-planets. If the user does not have this file already, it just need to run follow trough the last pipeline step procedures, or one can download one version of the preprocessed data from:* - Google drive access

[1]:
```
import pickle

file_path = './filtered.pkl'
with open(file_path, 'rb') as f:
    curves = pickle.load(f)
```

After, lets import the utils package with the support algorithms...

[2]:
```
from utils import *
```

Data type cannot be displayed: application/javascript, application/vnd.bokehjs_load.v0+json

Just plot an example time series to get a notion on the data obtained from the pickle (*.pkl*) file generated from the previous study script, and for that the `utils` library will be imported to take advantage of its `visual` functionality:

[3]:
```
from utils import *
from datetime import datetime
from datetime import timedelta

index = 8

time_in_days = [curves['i'][index]
                + timedelta(minutes=time) for time in curves['t'][index]]

x_data = [time_in_days, time_in_days]
y_data = [curves['r'][index], curves['y'][index]]
legends= ['Raw Light Curve', 'Filtered Light Curve']
colors = [2, 5]

p = visual.multline_plot(x_data, y_data,
                         legend_label=legends,
                         title='Light Curve Example',
                         color_index=colors,
                         y_axis={'label': 'Intensity'},
                         x_axis={'label': 'Date (m/dd)',
                                 'type': 'datetime'})
visual.show_plot(p)
```

Data type cannot be displayed: application/javascript, application/vnd.bokehjs_exec.v0+json

Just for ilustration... lets see how much samples we have in each curve. This gives the notion on how much data there is in each curve. From that it is possible to realise how much time the next algorithms will take to generate the necessary information.

[4]:
```
sizes = [len(curve) for curve in curves['y']]
x_values = [k+1 for k in range(len(sizes))]
sizes.sort(reverse=True)

p = visual.line_plot(x_values, sizes,
                     legend_label='Dimensions',
                     title='Curves dimensions',
                     color_index=2,
                     y_axis={'label': 'Sizes'},
                     x_axis={'label': 'Curve index'})
visual.show_plot(p)
```

Data type cannot be displayed: application/javascript, application/vnd.bokehjs_exec.v0+json

Compute some important constants that will be used during the analysis, such as mean sample time and sample frequency.

[5]:
```
sample_time = curves['t'][0][1] - curves['t'][0][0]    # minutes
sample_freq = 1 / (60 * sample_time)                   # hertz

print("The series have a time sample of {} minutes, consequently a sample frequency␣
↪of {} Hz".format(round(sample_time,2), round(sample_freq,6)))
```

(continues on next page)

(continued from previous page)

```
The series have a time sample of 8.56 minutes, consequently a sample frequency of 0.
↪001946 Hz
```

4.9 Feature: Frequency response

4.9.1 Introduction

One must know that theorically the most characteristic representation of a particular signal, would be its Fourier spectrum. If the real life signals were actually simple, it would be pretty simple to characterize those signals by their frequency spectrum. But unfortunately the real world signals are actully filled with noise from several natures.

So, if you want to get the most informative static information of a dynamic signal, it would be the power spectrum of this signal in the frequency domain without the undesired noise. In this chapter, it is developed an algorithm that attempt to find this representation for each light curve and adapt this information to be further used as features for machine learning algorithms.

Select one curve to create the frequency analysis, let's say the curve indexed as `index = 5`. We use that curve as an example to create the routine to create time-series power spectrum feature for the machine learning algorithms. Later, a generation algorithm will be presented, which will properly reproduce this process for all handle light curves. For that matter, we first present the curve:

[6]:
```
index = 5

time_in_days = [curves['i'][index]
                + timedelta(minutes=time) for time in curves['t'][index]]

x_data = [time_in_days, time_in_days]
y_data = [curves['r'][index], curves['y'][index]]
legends= ['Raw Light Curve', 'Filtered Light Curve']
colors = [2, 5]

p = visual.multline_plot(x_data, y_data,
                         legend_label=legends,
                         title='Light Curve Example',
                         color_index=colors,
                         y_axis={'label': 'Intensity'},
                         x_axis={'label': 'Date (m/dd)',
                                 'type': 'datetime'})
visual.show_plot(p)
```

Data type cannot be displayed: application/javascript, application/vnd.bokehjs_exec.v0+json

4.9.2 Spectrum generation

Then we can use the `signal` library from `scipy` to create the Periodogram or also called as the Spectrogram of this time series. We will create both the frequency information for the filtered signal, and the original data saved from the

last analysis, just to highlight the information removed with the filtering technique.

[7]:
```
import scipy.signal as ssg

freq, spectra = ssg.periodogram(curves['r'][index],
                                fs=sample_freq, scaling='density')
ffreq, fspectra = ssg.periodogram(curves['y'][index],
                                  fs=sample_freq, scaling='density')
efreq, espectra = ssg.periodogram(curves['y'][index]-curves['r'][index],
                                  fs=sample_freq, scaling='density')

x_data = [freq, ffreq]
y_data = [spectra, fspectra]
legends= ['Raw LC Spectrum', 'Filtered LC Spectrum']
colors = [2, 5]

p = visual.multline_plot(x_data, y_data,
                         legend_label=legends,
                         title='Light Curve Frequency Spectrum',
                         color_index=colors,
                         y_axis={'label': 'Magnitude'},
                         x_axis={'label': 'Frequency (Hz)',
                                 'type': 'log'})
p1 = visual.line_plot(efreq, espectra,
                      legend_label='Difference spectra',
                      title='Spectrum of the filtered out noise',
                      color_index=3,
                      y_axis={'label': 'Magnitude'},
                      x_axis={'label': 'Frequency (Hz)',
                              'type': 'log'})

visual.show_plot(p, p1)
```

Data type cannot be displayed: application/javascript, application/vnd.bokehjs_exec.v0+json

From here, one might see that the filtering technique applied on the last analysis, only remove high frequency compenents from the data. We don't know yep if the removed information is important or not for future analysis with machine learning, therefore, both frequency responses will be saved as feature for further analysis.

4.9.3 Detrended spectrum

But before generating the proposed feature for the machine learning, we must first present another possible process of the signal that might be relevant for further analysis. The DC level of the signal, usually represents the 0 Hz component of the frequency spectrum, and does not provide any dynamic meaningful information of the data. Therefore, it is common sence to first remove the so called trend that composes the DC level of the signal. This process is also called detrending of the signal. And it is pretty simple to be applied using the `signal` library from `scipy`.

[8]:
```
import scipy.signal as ssg

detrended_data = ssg.detrend(curves['y'][index], type='linear')
```

(continues on next page)

(continued from previous page)

```
time_in_days = [curves['i'][index]
                + timedelta(minutes=time) for time in curves['t'][index]]

p = visual.line_plot(time_in_days, curves['y'][index],
                     legend_label='Raw Light Curve',
                     title='Light Curve Raw',
                     color_index=2,
                     y_axis={'label': 'Intensity'},
                     x_axis={'label': 'Time (dd/mm/yy)'})
p1 = visual.line_plot(time_in_days, detrended_data,
                     legend_label='Detrended Light Curve',
                     title='Light Curve Detrended',
                     color_index=4,
                     y_axis={'label': 'Intensity'},
                     x_axis={'label': 'Time (dd/mm/yy)'})
visual.show_plot(p, p1)
```

Data type cannot be displayed: application/javascript, application/vnd.bokehjs_exec.v0+json

The influence on the periodogram can be shown by just generating once more the periodogram of the detrended signal and the previous one, with DC level influence.

[9]:
```
freq, spectra = ssg.periodogram(curves['y'][index], fs=sample_freq, scaling='spectrum
↪')
ffreq, fspectra = ssg.periodogram(detrended_data, fs=sample_freq, scaling='spectrum')
efreq, espectra = ssg.periodogram(detrended_data-curves['y'][index], fs=sample_freq,␣
↪scaling='spectrum')

x_data = [freq, ffreq]
y_data = [spectra, fspectra]
legends= ['Raw LC Spectrum', 'Detrended LC Spectrum']
colors = [2, 5]

p = visual.multline_plot(x_data, y_data,
                         legend_label=legends,
                         title='Light Curve Spectrum',
                         color_index=[2, 5],
                         y_axis={'label': 'Magnitude'},
                         x_axis={'label': 'Frequency (Hz)',
                                 'type': 'log'})
p1 = visual.line_plot(efreq, espectra,
                     legend_label='LC Difference Spectrum',
                     title='Difference Spectrum',
                     color_index=5,
                     y_axis={'label': 'Magnitude'},
                     x_axis={'label': 'Frequency (Hz)',
                             'type': 'log'})
visual.show_plot(p, p1)
```

Data type cannot be displayed: application/javascript, application/vnd.bokehjs_exec.v0+json

Interesting enough, this is the spectrum of the time series analysed (`index=5`). Of course we will detect several

highly evidenced frequency components, because the time series is clearly periodic. That is the most informative data that we could get from this time series using signal and dynamic systems theory.

> *One might see that the information removed by using the detrend technique only removes low frequency information in a smooth way. It is necessary to be carreful when using filtering tehcniques, so that no nonlinear and aggressive techniques are applied to the data. Those aggressive techniques are usually not practical and might result on unreal phenomenon on the data.*

4.9.4 Resample spectrum

One might notice that the spectrogram will not have the same resolution for each feature... This is the biggest problem on computing the time series power spectrum: the dimension inconsitency produced by the computation techniques to craft the power spectrum. Usually those algorithms rely on time series windows (remember the convolution process) to estimate the parameters of the Fourier representation of the series. Each time series has a particular number of samples and windows sizes with their sweep space, and those controls the build power spectrum resolution. Because of this, each light curve will have a particular power spectrum resolution.

It is therefore, necessary to create an algorithm able to reshape or resample those power spectrum to an unique resolution. This is necessary because most machine learning algorithms only deal with constant feature dimensions. To achieve this goal, it is introduced the `resample_freq_data` function.

[10]:
```
import scipy.signal as ssg
from PyAstronomy import pyasl

def resample_freq_data(data=None,
                       upper_sample=True,
                       window=13,
                       algorithm='hamming'):
    output_data = list()
    # Get the maximun and minimun frequencies, and
    # the resolution of each feature
    fmax, fmin, size = [], [], []
    for feat in data['freq']:
        fmax.append(feat.max())
        fmin.append(feat.min())
        size.append(len(feat))
    # Compute the high and low resolution step for
    # the resampling
    high_step = (max(fmax) - min(fmin)) / max(size) # high resolution
    low_step = (max(fmax) - min(fmin)) / min(size)  # low resolution
    # Define the cutoff frequency to limit the
    # spectrum data set, adn compute the new
    # signal resolution based on the steps
    up_cut_freq = min(fmax)
    down_cut_frea = max(fmin)
    high_resolution = up_cut_freq / high_step
    low_resolution = up_cut_freq / low_step
    # Compute the resample of each feature
    compact = zip(data['spec'], data['freq'])
    for spec, freq in compact:
        # Find the closest index to the cut
        # off freq, to remove the information
        uspec, ufreq = spec, freq
        if max(freq) > up_cut_freq:
            freq_diff = [abs(f - up_cut_freq) for f in freq]
```

(continues on next page)

(continued from previous page)

```
            cut_index = freq_diff.index(min(freq_diff))
            uspec = spec[:cut_index+1]
        # Compute the resampled signal for
        # high or low resolution
        if upper_sample:
            sig_size = round(high_resolution)
        else:
            sig_size = round(low_resolution)
        # Resample the spectrum
        uspec = ssg.resample(uspec, int(sig_size))
        # Smooth the resampled spectrum
        uspec = pyasl.smooth(uspec, window, algorithm)
        # Save the info into the output signal
        output_data.append( uspec[:-10] )
    return output_data
```

4.9.5 Generation algorithm

Lets run all the preprocessing for all the light curve time series, and build the feature variable to be saved for the machine learning step of the pipeline:

[11]:
```
detrend = True
filtered = True

# Detrend and filter all light curve
# time series data
size = len(curves['r'])
aux_data = {
    'freq': [],
    'spec': []
}
for item in range(size):
    # If we want a filtered data
    if filtered:
        data = curves['y'][item]
    else:
        data = curves['r'][item]
    # If we want a detrended data
    if detrend:
        data = ssg.detrend(data, type='linear')
    # Create the periodogram
    freq, spec = ssg.periodogram(data, fs=sample_freq, scaling='spectrum')
    # Save on the current variable
    aux_data['freq'].append( freq )
    aux_data['spec'].append( spec )

# Build the machine learning data
# structure to be saved on pickle file
ml_data = {
    'features': {
        'spec': resample_freq_data(aux_data,window=23)
    },
    'labels': curves['lab']
}
```

Just to guarantee the quality of the resampled variables, lets just plot the resampled one and the original spectrums

[12]:
```
import numpy as np

index_plot = 15

# Resample the frequency data
resolution = len(ml_data['features']['spec'][index_plot])
init = aux_data['freq'][index_plot][0]
final = aux_data['freq'][index_plot][-1]
rfreq = np.linspace(init, final, resolution)

# Create the plot data
x_data = [aux_data['freq'][index_plot], rfreq]
y_data = [aux_data['spec'][index_plot], ml_data['features']['spec'][index_plot]]
legends, colors, lw = ['Original', 'Resampled'], [2, 8], [3, 2]

p = visual.multline_plot(x_data, y_data,
                         legend_label=legends,
                         title='Spectrum validation',
                         color_index=colors,
                         line_width=lw,
                         y_axis={'label': 'Magnitude'},
                         x_axis={'label': 'Frequency (Hz)',
                                 'type': 'log'})

visual.show_plot(p)
```

Data type cannot be displayed: application/javascript, application/vnd.bokehjs_exec.v0+json

4.9.6 Save feature

Here we will modify the feature in a way to be ready for machine learning algorithm take this information as features and labels.

[ ]:
```
file_name = './features/freq_data/freq_data.pkl'

output = open(file_name, 'wb')
pickle.dump(ml_data, output)
output.close()
```

4.10 Feature: Naive Bayes likelihood

To create the Naive Bayes marginal likelihood approach, we just will try to use the bayesian theory for optimal filtering. The main structure here used will be the Bayes Ridge Regression, from the `sklearn` library. The main idea is to create a regression based model for each curve, and use the estimated parameters of each model as feature for machine learning classification. The regression model can be represented as

$$p(y|X, \omega, \alpha) = \mathcal{N}(y|X\omega, \alpha)$$

In the ridge regression, is assumed the prior value for the cofficient ω to be given by a spherical Gaussian, leading the regression problem to be mapped as

$$p(\omega|\lambda) = \mathcal{N}(\omega|0, \lambda^{-1}I_p)$$

The model estimation is just a matter of finding the set of ω, that minimizes the $\updownarrow_2$ norm for a provided user defined parameter λ. The ω values are the most characteristic information of each light curve. Thence, here we use those values as features to train the classifier to cluster the light curve classes.

There are some discussions that one might include for this approach. One might ask:

- What question should the bayes alorithm answer?
- It will be just a regression model pro predict the next step?
- It will be a regression model to predict model tendencies?

The best answer for that will appear when one uses the features provided from each path taken to try to classify the model. We cannot say before hand what feature will be best or not... Therefore, we must create some fetaure and then use their information to try to cluster the curve data.

4.10.1 Regression model

In both approaches is necessary to create a regression model, with chosen order n_x. To do that, one can use a function such as this:

[13]:
```
def build_regressor(data=None, order=6, norm=True):
    nx = order
    outputs = []
    regressors = []
    for curve in data:
        phi, y = [], []
        size = len(curve)
        # Normalize the curve data
        serie = curve
        if norm:
            serie = (curve - min(curve)) / (max(curve) - min(curve))
        # Build the regressor model
        for k in range(size-nx):
            phi.append(serie[k:k+nx])
            y.append(serie[k+nx])
        # Save the build regressors
        regressors.append(phi)
        outputs.append(y)
    return regressors, outputs
```

4.10.2 Next step parameters

Here we will create the regression problem for each light curve, and estimate the respective rigde bayes parameters. For that the `linear_model` library from `sklearn` will be used, specific the `BayesianRidge` object. To both create the regressor, for each light curve and then train the model, one must do the following

Usually it is interesting to normalize the data before fitting a regression model.

[14]:
```
from sklearn import linear_model
```

(continues on next page)

(continued from previous page)

```
bayes_data = {
    'features': {
        'params': []
    },
    'labels': curves['lab']
}

# Build the regression model
regr, out = build_regressor(curves['y'], order=20, norm=False)

# Estimate the bayes regression model
# for each set of regressor and outputs
for phi, y in zip(regr, out):
    # Create the model
    clf = linear_model.BayesianRidge()
    # Estimate the model
    clf.fit( phi, y )
    # Save the parameters
    bayes_data['features']['params'].append( clf.coef_ )
```

4.10.3 Save feature

Here we just save the feature variable in a particular pickle file

```
[ ]: file_name = './features/bayes_data/nx_6/bayes_data.pkl'

output = open(file_name, 'wb')
pickle.dump(bayes_data, output)
output.close()
```

4.11 Feature: Markov Hidden Models

Here we will develop a time series prediction algorithm using the so called Hidden Markov Models. They are not much more than a state space model without the input signal... a model that freely vary provided an initial condition. The model can be simply mapped as

$$x(k+1) = Ax(k)$$

where our job is to determine the parameter matrix A. Note that $x(k)$ has dimension n_x wich is the model complexity, and user provided.

> *Here we actully will use a the library called* `hmmlearn <https://hmmlearn.readthedocs.io/en/latest/tutorial.html>`__ *to estimate the model.*

One might wonder the reason to estimate this prediction model. The idea is simple, the matrix A will caracterize the main behavior of the dynamic system, and then it is possible this summarized information of the time series (the A matrix) as feature for the classification machine learning algorithm further used.

4.11.1 Preprocessing data

But for that to work, we need to maintain the parameters of each estimated A matrix as close as possible to each other. This means that we first need to preprocess the data in a manner to maintain the same signal power within each curve. This can be done by detrending the time series, and then normalizing their values.

To detrend the data we will use the same library previouly used above from `scipy.signal`.

[15]:
```
import scipy.signal as ssg

# Create the feature data to be
# further saved for machine learning
hmm_data = {
    'y': [],
    't': curves['t'],
    'labels': curves['lab'],
    'features': {
        'prob_matrix': []
    }
}

# Flags for the pipeline
norm = False

# Pre processing pipe line
for curve in curves['y']:
    # Detrend time series
    series = ssg.detrend(curve, type='linear')
    # Normalize (0, 1) time series
    if norm:
        mins, maxs = min(series), max(series)
        series = (series - mins) / (maxs - mins)
    # Add time series to processing data
    hmm_data['y'].append( series )
```

4.11.2 Estimate HMM

Now we can just use the algorithm created to determine the Hidden Markov Model for each curve time series. For each curve, we can fetch the parameter called `transmat_` which is actually the transition probability matrix of the state space model, here known as A. From that we will have a A matrix for each curve... with model complexity equal to $n_x =$ `n_components`, therefore making $A \in \Re_{(n_x, n_x)}$.

[16]:
```
import numpy as np
from hmmlearn import hmm

# Model parameters
cfgs = {
    'n_components': 8,
    'covariance_type': 'full',
    'n_iter': 100
}

# Compute each probability matrix
for curve_data in hmm_data['y']:
    # Create the hmm model
    remodel = hmm.GaussianHMM(**cfgs)
```

(continues on next page)

(continued from previous page)

```
    # Fit the hmm model to data
    remodel.fit(np.array(
        curve_data).reshape(len(curve_data),1))
    # Recover the probability matrix
    hmm_data['features']['prob_matrix'].append( remodel.transmat_ )
```

4.11.3 Save feature

Here we just save the feature variable in a particular pickle file

[ ]:
```
file_name = './features/hmm_data/nx_8/hmm_data.pkl'

output = open(file_name, 'wb')
pickle.dump(hmm_data, output)
output.close()
```

4.11.4 Download features

If the user does not want to run this pipeline step algorithm (since it takes time) one can just try to download the features, in several different setups from the Google Drive:

- Google drive access

In this drive the user will find a repository called `features`, that will have this structure:

```
./features
    |
    |-- bayes_data
    |       |-- nx_4
    |       |     |-- bayes_data.pkl
    |       |     |-- norm_bayes_data.pkl
    |       |-- nx_6
    |       |     |-- bayes_data.pkl
    |       |     |-- norm_bayes_data.pkl
    |       |
    |       |-- nx_20
    |             |-- bayes_data.pkl
    |             |-- norm_bayes_data.pkl
    |
    |-- freq_data
    |       |
    |       |-- freq_data.pkl
    |
    |-- hmm_data
            |-- nx_4
            |     |-- hmm_data.pkl
            |     |-- norm_hmm_data.pkl
            |-- nx_6
            |     |-- hmm_data.pkl
            |     |-- norm_hmm_data.pkl
            |
            |-- nx_20
                  |-- hmm_data.pkl
                  |-- norm_hmm_data.pkl
```

Here the frequency spectrum approach only has one feature type to be used. But the Hidden Markov Model and the Naive Bayes likelihood has several one, provided the combination of the parameters n_x and the possibility to normalize or not the light curve time series. Thence, if the user want the parameters for the model with $n_x = 8$, for the case considering the normalized data, it will be found at `/features/hmm_data/nx_8/norm_hmm_data.pkl` for the hidden markov model approach, and `/features/bayes_data/nx_8/norm_bayes_data.pkl` for the naive bayes.

4.12 XGBoost Classifier

The first algorithm that we will use is the XGBoost with its classic classifier. This is the classic simple algorithm from XGBoost library, further a more complex one will be used. This algorithm will be used for each generated feature, namelly:

- Periodograms
- Bayes Similarity
- Hidden Markov Models

All approaches will pass trough the common machine learning pipeline, where we must:

- Normalize the data (if necessary)
- Divide the data between trainning and testing
- Search the hyper parameters
- Cross validate the models
- Analyse the results

4.12.1 Periodograms

The application using the periodograms is actually pretty simple, now that the data is prepared and all of those preprocessing from last pipeline step is already done. The algorithm became straigh forward. First it is necessary to read the features generated.

[1]:
```
import pickle

file_name = './features/freq_data/freq_data.pkl'
with open(file_name, 'rb') as file:
    freq_data = pickle.load(file)
freq_data.keys()
```

[1]:
```
dict_keys(['features', 'labels'])
```

Manipulate features

After reading the data, it is necessary to create the classical regression structure model in the format $Y = f(\Theta, X)$, normalize the feature data and encode any possible label data into numerical classes. This is just the preparation for the machine leaning algorithm to guarantee that the provided info is properlly designed for any machine learning classical form.

[2]:
```
import numpy as np
from sklearn import preprocessing

# Create the label encoder
le_freq = preprocessing.LabelEncoder()
le_freq.fit(freq_data['labels'])

# Define the regression model
regressors = preprocessing.normalize(freq_data['features']['spec'])
outputs = le_freq.transform(freq_data['labels'])
```

[3]:
```
regressors.shape[1]
```

[3]:
```
12758
```

Also it is interesting to reduce the features dimension to build a simpler model. It is not necessary to create a classifier with such amount (12758 features...) of features. There are several techniques that can be used to reduce the features dimensions. The Principal Component Analisys, is very effective when dealing with high dimensional data. Here the `PCA` algorithm from the `sklearn` library is used.

[4]:
```
from sklearn.decomposition import PCA

# Create the PCA decomposer
pca_dec = PCA(n_components=70, svd_solver='arpack')

# Train the PCA object
pca_dec.fit(regressors)

# Transform the data using
# the PCA model
pca_regressor = pca_dec.transform(regressors)
```

Train-test data split

Next it is necessary to segregate the data into a set for validation and one for trainning the model.

[5]:
```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(
    pca_regressor, outputs, test_size=0.33, random_state=42)
```

Hyper tunning

We could consider tunning the model hyper parameters to answer questions such as:

- Wich value of `n_estimators` is the best for this model and data?
- Wich cost function is the best to be selected as `objective` for this model?

We could do a hyper search, to find the best hyper parameters for this model, automating the hyper parameter selection. There are several already builded algorithms to optimize this parameter search, and build find with high performance the best parameters, provided a set of possible values. But, to understand what those algorithms actually does, we could once build our own search algorithm...

As an example, lets run a first handly defined hyper parameter tunning using the confusion matrix of the model:

[6]:
```
import xgboost as xgb
from sklearn.metrics import confusion_matrix

# Define the model parameters
param_dist = {
    'objective':'binary:logistic',
    'n_estimators' : 11
}

# Create the range parameters to
# search
n_estimators = [ k+1 for k in range(100)]

# Create the plotting variable
plot_vals = {
    'true': {
        'confirmed targets': [],
        'eclipsing binaries': [],
    },
    'false': {
        'confirmed targets': [],
        'eclipsing binaries': [],
    }
}

# Estimate and validate each candidate
for opt in n_estimators:
    # Update the model parameters
    param_dist['n_estimators'] = opt
    # Create the xgBoost classifier
    clfs = xgb.XGBClassifier(**param_dist)
    # Fit the model to the data
    clfs.fit(X_train, y_train,
            eval_metric='logloss',
            verbose=True)
    # Estimate the test output
    y_pred = clfs.predict(X_test)
    # Compute the confusion matrix
    conf_mat = confusion_matrix(
        y_test, y_pred,
        normalize='true')
    # Save the confusion matrix
    plot_vals['true']['confirmed targets'].append(conf_mat[0,0])
    plot_vals['true']['eclipsing binaries'].append(conf_mat[1,1])
    plot_vals['false']['confirmed targets'].append(conf_mat[0,1])
    plot_vals['false']['eclipsing binaries'].append(conf_mat[1,0])
```

[7]:
```
from utils import *

# Line plot each confidence matrix parameter
x_data = [n_estimators, n_estimators, n_estimators, n_estimators]
y_data = [plot_vals['true']['confirmed targets'],
          plot_vals['true']['eclipsing binaries'],
          plot_vals['false']['confirmed targets'],
          plot_vals['false']['eclipsing binaries']]
legends= ['True - C.T.', 'True - E.B.', 'False - C.T.', 'False - E.B.']
colors = [6, 7, 2, 3]
```

(continues on next page)

(continued from previous page)

```
p = visual.multiline_plot(x_data, y_data,
                          legend_label=legends,
                          title='Hyper parameter search - Confusion parameters plot',
                          color_index=colors,
                          y_axis={'label': 'Proportion'},
                          x_axis={'label': 'n_estimators'})
visual.show_plot(p)
```

Data type cannot be displayed: application/javascript, application/vnd.bokehjs_load.v0+json

Data type cannot be displayed: application/javascript, application/vnd.bokehjs_exec.v0+json

Train model

After running the hyper parameter search we can create a model with the best defined hyper parameters, or setup parameters, and consolidate the model in to the best version for further performance analysis. The model is saved on a particular variable, such as `freq_clf` to be further used in some vote chain model, if further necessary.

> *One interesting result from the above result, is that the best compromise result happens at the :math:'n_x = 5' not on :math:'n_x=4' as it seems. Even though for the :math:'n_x=4' the algorithm is able to get all the exo planets, the compromise on having a confidence of only 66% for the eclipsing binaries classification (classifier close to a coin flipper to classify eclipsing binaries), doesn't allow us to select :math:'n_x=4'. Therefore the best trade-off on both classes happens at :math:'n_x=5'.*

[8]:
```
# XGBoost Classifier model parameters
param_dist = {
    'verbosity': 0,
    'objective':'binary:logistic',
    'n_estimators' : 5
}

# Create the model classifier
freq_clf = xgb.XGBClassifier(**param_dist)

# Train the model
freq_clf.fit(X_train, y_train,
             eval_set=[
               (X_train, y_train),
               (X_test, y_test)
             ],
             eval_metric='logloss',
             verbose=False)
```

[8]:
```
XGBClassifier(base_score=0.5, booster=None, colsample_bylevel=1,
              colsample_bynode=1, colsample_bytree=1, gamma=0, gpu_id=-1,
              importance_type='gain', interaction_constraints=None,
              learning_rate=0.300000012, max_delta_step=0, max_depth=6,
              min_child_weight=1, missing=nan, monotone_constraints=None,
              n_estimators=5, n_jobs=0, num_parallel_tree=1,
              objective='binary:logistic', random_state=0, reg_alpha=0,
```

(continues on next page)

(continued from previous page)

```
                reg_lambda=1, scale_pos_weight=1, subsample=1, tree_method=None,
                validate_parameters=False, verbosity=0)
```

Results

In this part it is presented the results from the classification algorithm. Both regarding the data visualization and the model classification quality.

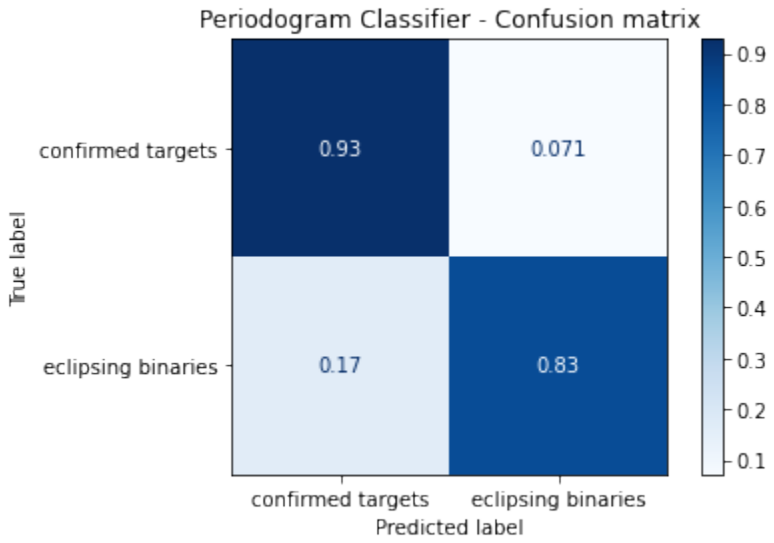
[9]:
```
import pprint
pp = pprint.PrettyPrinter(indent=2)

pp.pprint(freq_clf.evals_result())
```

```
{ 'validation_0': { 'logloss': [ 0.537111,
                                 0.449156,
                                 0.384568,
                                 0.327504,
                                 0.280901]},
  'validation_1': { 'logloss': [ 0.609796,
                                 0.574501,
                                 0.546393,
                                 0.504637,
                                 0.509942]}}
```

[10]:
```
import matplotlib.pyplot as plt
from sklearn.metrics import plot_confusion_matrix

disp = plot_confusion_matrix(freq_clf, X_test, y_test,
                             display_labels=le_freq.classes_,
                             cmap=plt.cm.Blues,
                             normalize='true')
disp.ax_.set_title('Periodogram Classifier - Confusion matrix')
plt.show()
```



Comments

From this results it is possible to see that the classifier using periodogram amplitudes can get some interesting knowledge on the eclipsing binaries classification, and have an even better results for the exo planets (confirmed target labels). Also the results are not just good considering the classification capability, but also considering the robustness of the algorithm. The robustness quality can be checked from the printed loss, which shows a continous descending loss for both the test and train data.

4.12.2 Naive Bayes likelihood

Here we will read the Naive Bayes model parameters estimated for each light curve and use this information as feature for the xgBoost classifier. To start this approach, we must first read the Bayes features saved from last step:

[11]:
```
import pickle

file_name = './features/bayes_data/nx_6/bayes_data.pkl'
with open(file_name, 'rb') as file:
    bayes_data = pickle.load(file)
bayes_data.keys()
```

[11]:
```
dict_keys(['features', 'labels'])
```

Manipulate features

After reading the data, it is necessary to create the classical regression structure model in the format $Y = f(\Theta, X)$, normalize the feature data and encode any possible label data into numerical classes. This is just the preparation for the machine leaning algorithm to guarantee that the provided info is properlly designed for any machine learning classical form.

[12]:
```
import numpy as np
from sklearn import preprocessing

# Create the label encoder
le_bayes = preprocessing.LabelEncoder()
le_bayes.fit(bayes_data['labels'])

# Define the regression model
regressors = preprocessing.normalize(bayes_data['features']['params'])
outputs = le_bayes.transform(bayes_data['labels'])
```

Train-test data split

Next it is necessary to segregate the data into a set for validation and one for trainning the model.

[13]:
```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(
    regressors, outputs, test_size=0.33, random_state=42)
```

Hyper tunning

We could consider tunning the model hyper parameters to answer questions such as:

- Wich value of `n_estimators` is the best for this model and data?
- Wich cost function is the best to be selected as `objective` for this model?

We could do a hyper search, to find the best hyper parameters for this model, automating the hyper parameter selection. There are several already builded algorithms to optimize this parameter search, and build find with high performance the best parameters, provided a set of possible values. But, to understand what those algorithms actually does, we could once build our own search algorithm...

As an example, lets run a first handly defined hyper parameter tunning using the confusion matrix of the model:

[14]:
```
import xgboost as xgb
from sklearn.metrics import confusion_matrix

# Define the model parameters
param_dist = {
    'objective':'binary:logistic',
    'n_estimators' : 11
}

# Create the range parameters to
# search
n_estimators = [ k+1 for k in range(100)]

# Create the plotting variable
plot_vals = {
    'true': {
        'confirmed targets': [],
        'eclipsing binaries': [],
    },
    'false': {
        'confirmed targets': [],
        'eclipsing binaries': [],
    }
}

# Estimate and validate each candidate
for opt in n_estimators:
    # Update the model parameters
    param_dist['n_estimators'] = opt
    # Create the xgBoost classifier
    clfs = xgb.XGBClassifier(**param_dist)
    # Fit the model to the data
    clfs.fit(X_train, y_train,
            eval_metric='logloss',
            verbose=True)
    # Estimate the test output
    y_pred = clfs.predict(X_test)
    # Compute the confusion matrix
    conf_mat = confusion_matrix(
        y_test, y_pred,
        normalize='true')
    # Save the confusion matrix
    plot_vals['true']['confirmed targets'].append(conf_mat[0,0])
    plot_vals['true']['eclipsing binaries'].append(conf_mat[1,1])
    plot_vals['false']['confirmed targets'].append(conf_mat[0,1])
    plot_vals['false']['eclipsing binaries'].append(conf_mat[1,0])
```

[15]:
```
from utils import *

# Line plot each confidence matrix parameter
x_data = [n_estimators, n_estimators, n_estimators, n_estimators]
y_data = [plot_vals['true']['confirmed targets'],
          plot_vals['true']['eclipsing binaries'],
          plot_vals['false']['confirmed targets'],
          plot_vals['false']['eclipsing binaries']]
legends= ['True - C.T.', 'True - E.B.', 'False - C.T.', 'False - E.B.']
colors = [6, 7, 2, 3]

p = visual.multline_plot(x_data, y_data,
                         legend_label=legends,
                         title='Hyper parameter search - Confusion parameters plot',
                         color_index=colors,
                         y_axis={'label': 'Proportion'},
                         x_axis={'label': 'n_estimators'})
visual.show_plot(p)
```

Data type cannot be displayed: application/javascript, application/vnd.bokehjs_exec.v0+json

Train model

After running the hyper parameter search we can create a model with the best defined hyper parameters, or setup parameters, and consolidate the model in to the best version for further performance analysis. The model is saved on a particular variable, such as `bayes_clf` to be further used in some vote chain model, if further necessary.

[16]:
```
# XGBoost Classifier model parameters
param_dist = {
    'verbosity': 0,
    'objective':'binary:logistic',
    'n_estimators' : 11
}

# Create the model classifier
bayes_clf = xgb.XGBClassifier(**param_dist)

# Train the model
bayes_clf.fit(X_train, y_train,
              eval_set=[
                (X_train, y_train),
                (X_test, y_test)
              ],
              eval_metric='logloss',
              verbose=False)
```

[16]:
```
XGBClassifier(base_score=0.5, booster=None, colsample_bylevel=1,
              colsample_bynode=1, colsample_bytree=1, gamma=0, gpu_id=-1,
              importance_type='gain', interaction_constraints=None,
              learning_rate=0.300000012, max_delta_step=0, max_depth=6,
              min_child_weight=1, missing=nan, monotone_constraints=None,
              n_estimators=11, n_jobs=0, num_parallel_tree=1,
              objective='binary:logistic', random_state=0, reg_alpha=0,
```

(continues on next page)

(continued from previous page)

```
              reg_lambda=1, scale_pos_weight=1, subsample=1, tree_method=None,
              validate_parameters=False, verbosity=0)
```

Results

In this part it is presented the results from the classification algorithm. Both regarding the data visualization and the model classification quality.

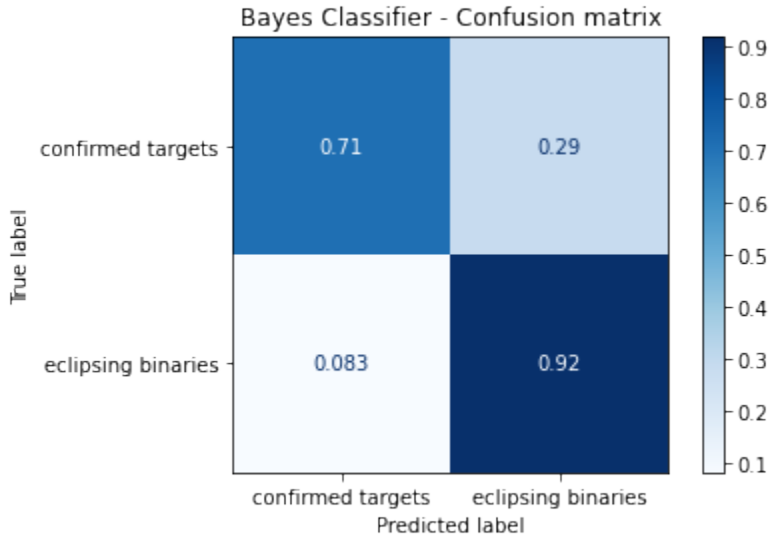
[17]:
```
import pprint
pp = pprint.PrettyPrinter(indent=2)

pp.pprint(bayes_clf.evals_result())
```

```
{ 'validation_0': { 'logloss': [ 0.567346,
                                 0.456689,
                                 0.383979,
                                 0.341458,
                                 0.312702,
                                 0.277829,
                                 0.251812,
                                 0.232627,
                                 0.216223,
                                 0.202809,
                                 0.194951]},
  'validation_1': { 'logloss': [ 0.657235,
                                 0.599724,
                                 0.567681,
                                 0.561523,
                                 0.556633,
                                 0.547125,
                                 0.553211,
                                 0.528883,
                                 0.531398,
                                 0.528597,
                                 0.536761]}}
```

[18]:
```
import matplotlib.pyplot as plt
from sklearn.metrics import plot_confusion_matrix

disp = plot_confusion_matrix(bayes_clf, X_test, y_test,
                             display_labels=le_bayes.classes_,
                             cmap=plt.cm.Blues,
                             normalize='true')
disp.ax_.set_title('Bayes Classifier - Confusion matrix')
plt.show()
```



Comments

From this results it is possible to see that the classifier using Naive Bayes estimated model parameters is able to highly characterize the eclipsing binaries, and has acceptable classification performance for the confirmed targets. Also, the algorithm, as in the approach using the Periodograms, is highly consistent, due to the continuous descending loss during learning process for both the trainning and testing data. But comparing with the Periodogram approach, we can ensure that this one is much more simpler, considering the data preprocessing, than the one using the Periodograms. To generate the periodograms, several artesanal filtering techniques are necessary to generate the data in a suitable format for the machine learning model. Even considering that the classification performance is better then the Naive Bayes, more tests will be necessary to check if this better performace worth the trouble and stability risk of the preprocessing techniques.

4.12.3 Hidden Markov Models

Here we use the model estimated from the Hidden Markov Models library, wich is the estimated A matrix, or the so called transition probability matrices as feature for the learning classifier. For that we must read the pickle file with the desired features:

[19]:
```
import pickle

file_name = './features/hmm_data/nx_8/hmm_data.pkl'
with open(file_name, 'rb') as file:
    hmm_data = pickle.load(file)
hmm_data.keys()
```

[19]:
```
dict_keys(['y', 't', 'labels', 'features'])
```

Manipulate features

After reading the data, it is necessary to create the classical regression structure model in the format $Y = f(\Theta, X)$, normalize the feature data and encode any possible label data into numerical classes. This is just the preparation for the

machine leaning algorithm to guarantee that the provided info is properlly designed for any machine learning classical form.

[20]:
```
import numpy as np
from sklearn import preprocessing

# Encode the label
le_hmm = preprocessing.LabelEncoder()
le_hmm.fit( hmm_data['labels'] )

# Define the model order
feat = hmm_data['features']
nx = feat['prob_matrix'][0].shape[0]

regressors = []
for phi in feat['prob_matrix']:
    # Reshape the regressor
    reg = phi.reshape(nx*nx)
    # Add to the regressors
    regressors.append(reg)
# Normalize the regressors
regressors = preprocessing.normalize(regressors)
# Define outputs as encoded variables
outputs = le_hmm.transform(hmm_data['labels'])
```

Train-test data split

Next it is necessary to segregate the data into a set for validation and one for trainning the model.

[21]:
```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(
    regressors, outputs, test_size=0.33, random_state=42)
```

Hyper tunning

We could consider tunning the model hyper parameters to answer questions such as:

- Wich value of `n_estimators` is the best for this model and data?
- Wich cost function is the best to be selected as `objective` for this model?

We could do a hyper search, to find the best hyper parameters for this model, automating the hyper parameter selection. There are several already builded algorithms to optimize this parameter search, and build find with high performance the best parameters, provided a set of possible values. But, to understand what those algorithms actually does, we could once build our own search algorithm...

As an example, lets run a first handly defined hyper parameter tunning using the confusion matrix of the model:

[22]:
```
from sklearn.metrics import confusion_matrix

n_estimators = [ k+1 for k in range(100)]

conf_matrices = []
for opt in n_estimators:
    # Update the model parameters
```

(continues on next page)

(continued from previous page)

```
    param_dist['n_estimators'] = opt
    # Create the xgBoost classifier
    clfs = xgb.XGBClassifier(**param_dist)
    # Fit the model to the data
    clfs.fit(X_train, y_train,
            eval_metric='logloss',
            verbose=True)
    # Estimate the test output
    y_pred = clfs.predict(X_test)
    # Compute the confusion matrix
    conf_mat = confusion_matrix(
        y_test, y_pred,
        normalize='true')
    # Save the confusion matrix
    conf_matrices.append(conf_mat)
```

[23]:
```
from utils import *

# Create and organize the plot values
plot_vals = {
    'true': {
        'confirmed targets': [],
        'eclipsing binaries': [],
    },
    'false': {
        'confirmed targets': [],
        'eclipsing binaries': [],
    }
}
for result in conf_matrices:
    plot_vals['true']['confirmed targets'].append(result[0,0])
    plot_vals['true']['eclipsing binaries'].append(result[1,1])
    plot_vals['false']['confirmed targets'].append(result[0,1])
    plot_vals['false']['eclipsing binaries'].append(result[1,0])

x_values = range(len(conf_matrices))
x_data = [x_values, x_values, x_values, x_values]
y_data = [plot_vals['true']['confirmed targets'],
          plot_vals['true']['eclipsing binaries'],
          plot_vals['false']['confirmed targets'],
          plot_vals['false']['eclipsing binaries']]
legends= ['True - C.T.', 'True - E.B.', 'False - C.T.', 'False - E.B.']
colors = [6, 7, 2, 3]

p = visual.multline_plot(x_data, y_data,
                         legend_label=legends,
                         title='Hyper parameter search - Confusion plot',
                         color_index=colors,
                         y_axis={'label': 'Intensity'},
                         x_axis={'label': 'n_estimators'})
visual.show_plot(p)
```

Data type cannot be displayed: application/javascript, application/vnd.bokehjs_exec.v0+json

Train model

After running the hyper parameter search we can create a model with the best defined hyper parameters, or setup parameters, and consolidate the model in to the best version for further performance analysis. The model is saved on a particular variable, such as `hmm_clf` to be further used in some vote chain model, if further necessary.

[24]:
```
import xgboost as xgb

param_dist = {
    'verbosity': 0,
    'objective':'binary:logistic',
    'n_estimators' : 34
}

hmm_clf = xgb.XGBClassifier(**param_dist)

hmm_clf.fit(X_train, y_train,
            eval_set=[
                (X_train, y_train),
                (X_test, y_test)
            ],
            eval_metric='logloss',
            verbose=False)
```

[24]:
```
XGBClassifier(base_score=0.5, booster=None, colsample_bylevel=1,
              colsample_bynode=1, colsample_bytree=1, gamma=0, gpu_id=-1,
              importance_type='gain', interaction_constraints=None,
              learning_rate=0.300000012, max_delta_step=0, max_depth=6,
              min_child_weight=1, missing=nan, monotone_constraints=None,
              n_estimators=34, n_jobs=0, num_parallel_tree=1,
              objective='binary:logistic', random_state=0, reg_alpha=0,
              reg_lambda=1, scale_pos_weight=1, subsample=1, tree_method=None,
              validate_parameters=False, verbosity=0)
```

Results

Here we include some visualization results for the xgBoost algorithm classification. As the first result, we just print the model eval metrics, here the *log loss* of the model, for both the trainning and testing data.

[25]:
```
import pprint
pp = pprint.PrettyPrinter(indent=2)

evals_result = hmm_clf.evals_result()
pp.pprint(evals_result)
```

```
{ 'validation_0': { 'logloss': [ 0.572305,
                                 0.474664,
                                 0.396965,
                                 0.336296,
                                 0.304903,
                                 0.262527,
                                 0.235207,
                                 0.212948,
                                 0.192356,
                                 0.177346,
                                 0.163549,
                                 0.15337,
```

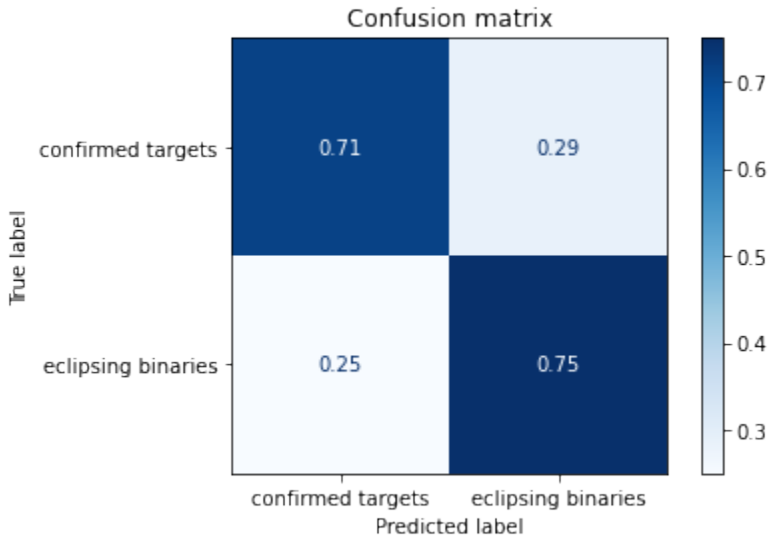
(continues on next page)

(continued from previous page)

```
                                    0.140518,
                                    0.130899,
                                    0.123485,
                                    0.117721,
                                    0.113933,
                                    0.11041,
                                    0.106786,
                                    0.101635,
                                    0.098872,
                                    0.096439,
                                    0.093749,
                                    0.091741,
                                    0.089836,
                                    0.088826,
                                    0.087047,
                                    0.086091,
                                    0.083466,
                                    0.081756,
                                    0.079948,
                                    0.079189,
                                    0.077326,
                                    0.076633]},
  'validation_1': { 'logloss': [ 0.673667,
                                    0.635211,
                                    0.618026,
                                    0.607399,
                                    0.628232,
                                    0.650925,
                                    0.654683,
                                    0.651218,
                                    0.653813,
                                    0.663263,
                                    0.660528,
                                    0.677213,
                                    0.667749,
                                    0.663841,
                                    0.676913,
                                    0.67355,
                                    0.675848,
                                    0.674091,
                                    0.687084,
                                    0.693771,
                                    0.69817,
                                    0.700085,
                                    0.69875,
                                    0.69817,
                                    0.688993,
                                    0.692456,
                                    0.697232,
                                    0.699571,
                                    0.694171,
                                    0.6861,
                                    0.6878,
                                    0.690059,
                                    0.682183,
                                    0.685172]}}
```

[26]:
```
import matplotlib.pyplot as plt
from sklearn.metrics import plot_confusion_matrix

disp = plot_confusion_matrix(hmm_clf, X_test, y_test,
                             display_labels=le_hmm.classes_,
                             cmap=plt.cm.Blues,
                             normalize='true')
disp.ax_.set_title('Confusion matrix')
plt.show()
```



Comments

This approch is far the worst one from the ones presented here. This is because, not only the preprocessing algorithms are to heavy and has several stability issues, the algorithm does not have a consistent and desired learning behavior for the test data, when considering the loss. Even though, the algorithm has a consistent classification performance on both classes, this other issues make these results also unstable, therefore it cannot be trusted.

4.13 Decision trees

The approach presented in this path, will be one that uses the classical Decision Trees to manage the classification problem. Most of the steps presented here will repeat itself from the XGBoost path, but the algorithm analysis will be a little bit different. This shows that by following the procedure proposed in this study, in general, leads to an interesting pipeline that enables the user to run several machine learning algorithms from the same rundown. Also, this algorithm will be used for each generated feature, namelly:

- Periodograms
- Bayes Similarity
- Hidden Markov Models

All approaches will pass trough the common machine learning pipeline, where we must:

- Normalize the data (if necessary)
- Divide the data between trainning and testing

- Search the hyper parameters
- Cross validate the models
- Analyse the results

4.13.1 Periodograms

The application using the periodograms is actually pretty simple, now that the data is prepared and all of those preprocessing from last pipeline step is already done. The algorithm became straigh forward. First it is necessary to read the features generated.

[1]:
```
import pickle

file_name = './features/freq_data/freq_data.pkl'
with open(file_name, 'rb') as file:
    freq_data = pickle.load(file)
freq_data.keys()
```

[1]:
```
dict_keys(['features', 'labels'])
```

Manipulate features

After reading the data, it is necessary to create the classical regression structure model in the format $Y = f(\Theta, X)$, normalize the feature data and encode any possible label data into numerical classes. This is just the preparation for the machine leaning algorithm to guarantee that the provided info is properlly designed for any machine learning classical form.

[2]:
```
import numpy as np
from sklearn import preprocessing

# Create the label encoder
le_freq = preprocessing.LabelEncoder()
le_freq.fit(freq_data['labels'])

# Define the regression model
regressors = preprocessing.normalize(freq_data['features']['spec'])
outputs = le_freq.transform(freq_data['labels'])
```

[3]:
```
regressors.shape[1]
```

[3]:
```
12758
```

Also it is interesting to reduce the features dimension to build a simpler model. It is not necessary to create a classifier with such amount (12758 features...) of features. There are several techniques that can be used to reduce the features dimensions. The Principal Component Analisys, is very effective when dealing with high dimensional data. Here the `PCA` algorithm from the `sklearn` library is used.

[4]:
```
from sklearn.decomposition import PCA

# Create the PCA decomposer
pca_dec = PCA(n_components=70)#, svd_solver='arpack')

# Train the PCA object
pca_dec.fit(regressors)
```

(continues on next page)

(continued from previous page)

```
# Transform the data using
# the PCA model
pca_regressor = pca_dec.transform(regressors)
```

Train-test data split

Next it is necessary to segregate the data into a set for validation and one for trainning the model.

[5]:
```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(
    pca_regressor, outputs, test_size=0.33, random_state=42)
```

Hyper tunning

We could consider tunning the model hyper parameters to answer questions such as:

- Wich value of `max_depth` is the best for this model and data?
- Wich cost function is the best to be selected as `criterion` for this model?

We could do a hyper search, to find the best hyper parameters for this model, automating the hyper parameter selection. There are several already builded algorithms to optimize this parameter search, and build find with high performance the best parameters, provided a set of possible values. But, to understand what those algorithms actually does, we could once build our own search algorithm...

As an example, lets run a first handly defined hyper parameter tunning using the confusion matrix of the model:

[6]:
```
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import confusion_matrix

# Define the model parameters
param_dist = {
    'random_state' : 0,
    'criterion' : 'entropy', # or 'entropy' for information gain
    'max_depth' : 0
}

# Create the range parameters to
# search
max_features_dim = X_train.shape[1]
max_depth = [k + 1 for k in range(max_features_dim)]

# Create the plotting variable
plot_vals = {
    'true': {
        'confirmed targets': [],
        'eclipsing binaries': [],
    },
    'false': {
        'confirmed targets': [],
        'eclipsing binaries': [],
    }
}
```

(continues on next page)

(continued from previous page)

```
# Estimate and validate each candidate
for opt in max_depth:
    # Update the model parameters
    param_dist['max_depth'] = opt
    # Create the xgBoost classifier
    clfs = DecisionTreeClassifier(**param_dist)
    # Fit the model to the data
    clfs.fit(X_train, y_train)
    # Estimate the test output
    y_pred = clfs.predict(X_test)
    # Compute the confusion matrix
    conf_mat = confusion_matrix(
        y_test, y_pred,
        normalize='true')
    # Save the confusion matrix
    plot_vals['true']['confirmed targets'].append(conf_mat[0,0])
    plot_vals['true']['eclipsing binaries'].append(conf_mat[1,1])
    plot_vals['false']['confirmed targets'].append(conf_mat[0,1])
    plot_vals['false']['eclipsing binaries'].append(conf_mat[1,0])
```

[7]:
```
from utils import *

# Line plot each confidence matrix parameter
x_data = [max_depth, max_depth, max_depth, max_depth]
y_data = [plot_vals['true']['confirmed targets'],
          plot_vals['true']['eclipsing binaries'],
          plot_vals['false']['confirmed targets'],
          plot_vals['false']['eclipsing binaries']]
legends= ['True - C.T.', 'True - E.B.', 'False - C.T.', 'False - E.B.']
colors = [6, 7, 2, 3]

p = visual.multline_plot(x_data, y_data,
                         legend_label=legends,
                         title='Hyper parameter search - Confusion parameters plot',
                         color_index=colors,
                         y_axis={'label': 'Proportion'},
                         x_axis={'label': 'n_estimators'})
visual.show_plot(p)
```

Data type cannot be displayed: application/javascript, application/vnd.bokehjs_load.v0+json

Data type cannot be displayed: application/javascript, application/vnd.bokehjs_exec.v0+json

Train model

After running the hyper parameter search we can create a model with the best defined hyper parameters, or setup parameters, and consolidate the model in to the best version for further performance analysis.

[8]:
```
param_dist = {
```

(continues on next page)

(continued from previous page)

```
    'random_state': 0,
    'criterion' : 'entropy', # or 'entropy' for information gain
    'max_depth' : max_depth[2]
}

# Create the model classifier
freq_clf = DecisionTreeClassifier(**param_dist)

# Train the model
freq_clf.fit(X_train, y_train)
```

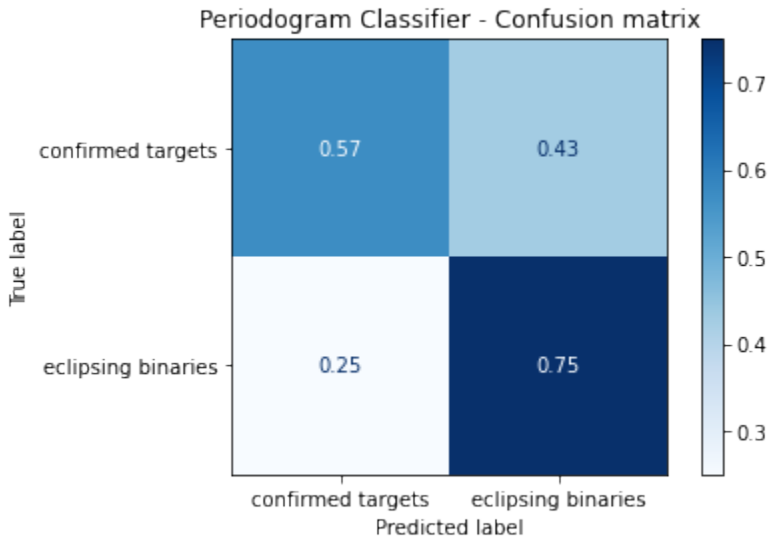
[8]:
```
DecisionTreeClassifier(ccp_alpha=0.0, class_weight=None, criterion='entropy',
                       max_depth=3, max_features=None, max_leaf_nodes=None,
                       min_impurity_decrease=0.0, min_impurity_split=None,
                       min_samples_leaf=1, min_samples_split=2,
                       min_weight_fraction_leaf=0.0, presort='deprecated',
                       random_state=0, splitter='best')
```

Results

In this part it is presented the results from the classification algorithm. Both regarding the data visualization and the model classification quality.

[9]:
```
import matplotlib.pyplot as plt
from sklearn.metrics import plot_confusion_matrix

disp = plot_confusion_matrix(freq_clf, X_test, y_test,
                             display_labels=le_freq.classes_,
                             cmap=plt.cm.Blues,
                             normalize='true')
disp.ax_.set_title('Periodogram Classifier - Confusion matrix')
plt.show()
```



Comments

From this result, it possible to see that the decision tree model is able to learn some information from eclipsing binaries, as it just get 25% of the testing data wrong. But when analysing the confirmed targets results, it is possible to see that we have a coin toss algorithm, since it cannot learn anything from the exo planets data. Comparing with the XGBoost algorithm it is possible to see a big downgrade on the classification results, ensuring the outstanding capability of the boosting feature of the algorithm.

4.13.2 Naive Bayes likelihood

Here we will read the Naive Bayes model parameters estimated for each light curve and use this information as feature for the xgBoost classifier. To start this approach, we must first read the Bayes features saved from last step:

[10]:
```
import pickle

file_name = './features/bayes_data/nx_12/bayes_data.pkl'
with open(file_name, 'rb') as file:
    bayes_data = pickle.load(file)
bayes_data.keys()
```

[10]:
```
dict_keys(['features', 'labels'])
```

Manipulate features

After reading the data, it is necessary to create the classical regression structure model in the format $Y = f(\Theta, X)$, normalize the feature data and encode any possible label data into numerical classes. This is just the preparation for the machine leaning algorithm to guarantee that the provided info is properlly designed for any machine learning classical form.

[11]:
```
import numpy as np
from sklearn import preprocessing

# Create the label encoder
le_bayes = preprocessing.LabelEncoder()
le_bayes.fit(bayes_data['labels'])

# Define the regression model
regressors = preprocessing.normalize(bayes_data['features']['params'])
outputs = le_bayes.transform(bayes_data['labels'])
```

Train-test data split

Next it is necessary to segregate the data into a set for validation and one for trainning the model.

[12]:
```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(
    regressors, outputs, test_size=0.33, random_state=42)
```

Hyper tunning

We could consider tunning the model hyper parameters to answer questions such as:

- Wich value of `n_estimators` is the best for this model and data?
- Wich cost function is the best to be selected as `criterion` for this model?

We could do a hyper search, to find the best hyper parameters for this model, automating the hyper parameter selection. There are several already builded algorithms to optimize this parameter search, and build find with high performance the best parameters, provided a set of possible values. But, to understand what those algorithms actually does, we could once build our own search algorithm...

As an example, lets run a first handly defined hyper parameter tunning using the confusion matrix of the model:

[13]:
```
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import confusion_matrix

# Define the model parameters
param_dist = {
    'random_state' : 0,
    'criterion' : 'gini', # or 'entropy' for information gain
    'max_depth' : 0
}

# Create the range parameters to
# search
max_features_dim = X_train.shape[1]
max_depth = [k + 1 for k in range(max_features_dim)]

# Create the plotting variable
plot_vals = {
    'true': {
        'confirmed targets': [],
        'eclipsing binaries': [],
    },
    'false': {
        'confirmed targets': [],
        'eclipsing binaries': [],
    }
}

# Estimate and validate each candidate
for opt in max_depth:
    # Update the model parameters
    param_dist['max_depth'] = opt
    # Create the xgBoost classifier
    clfs = DecisionTreeClassifier(**param_dist)
    # Fit the model to the data
    clfs.fit(X_train, y_train)
    # Estimate the test output
    y_pred = clfs.predict(X_test)
    # Compute the confusion matrix
    conf_mat = confusion_matrix(
        y_test, y_pred,
        normalize='true')
    # Save the confusion matrix
    plot_vals['true']['confirmed targets'].append(conf_mat[0,0])
    plot_vals['true']['eclipsing binaries'].append(conf_mat[1,1])
    plot_vals['false']['confirmed targets'].append(conf_mat[0,1])
    plot_vals['false']['eclipsing binaries'].append(conf_mat[1,0])
```

[14]:
```
from utils import *

# Line plot each confidence matrix parameter
x_data = [max_depth, max_depth, max_depth, max_depth]
y_data = [plot_vals['true']['confirmed targets'],
          plot_vals['true']['eclipsing binaries'],
          plot_vals['false']['confirmed targets'],
          plot_vals['false']['eclipsing binaries']]
legends= ['True - C.T.', 'True - E.B.', 'False - C.T.', 'False - E.B.']
colors = [6, 7, 2, 3]

p = visual.multline_plot(x_data, y_data,
                         legend_label=legends,
                         title='Hyper parameter search - Confusion parameters plot',
                         color_index=colors,
                         y_axis={'label': 'Proportion'},
                         x_axis={'label': 'n_estimators'})
visual.show_plot(p)
```

Data type cannot be displayed: application/javascript, application/vnd.bokehjs_exec.v0+json

Train model

After running the hyper parameter search we can create a model with the best defined hyper parameters, or setup parameters, and consolidate the model in to the best version for further performance analysis. The model is saved on a particular variable, such as `bayes_clf` to be further used in some vote chain model, if further necessary.

[15]:
```
param_dist = {
    'random_state': 0,
    'criterion' : 'gini', # or 'entropy' for information gain
    'max_depth' : max_depth[3]
}

# Create the model classifier
bayes_clf = DecisionTreeClassifier(**param_dist)

# Train the model
bayes_clf.fit(X_train, y_train)
```

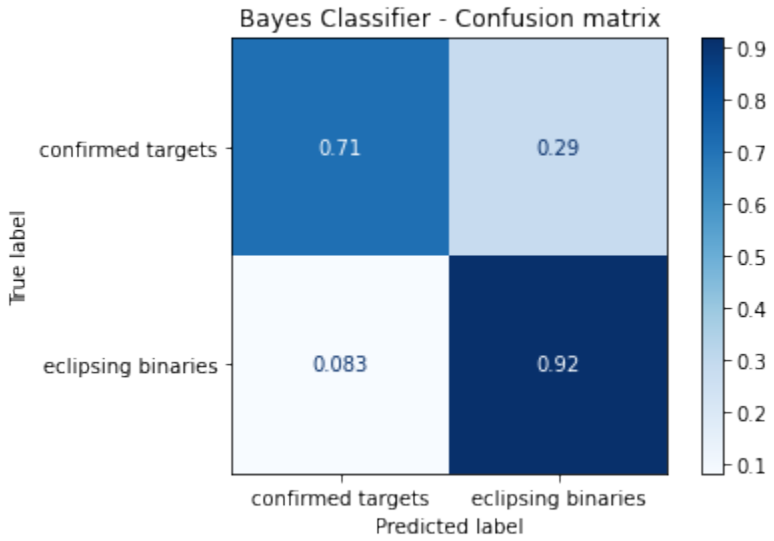
[15]:
```
DecisionTreeClassifier(ccp_alpha=0.0, class_weight=None, criterion='gini',
                       max_depth=4, max_features=None, max_leaf_nodes=None,
                       min_impurity_decrease=0.0, min_impurity_split=None,
                       min_samples_leaf=1, min_samples_split=2,
                       min_weight_fraction_leaf=0.0, presort='deprecated',
                       random_state=0, splitter='best')
```

Results

In this part it is presented the results from the classification algorithm. Both regarding the data visualization and the model classification quality.

[16]:
```
import matplotlib.pyplot as plt
from sklearn.metrics import plot_confusion_matrix

disp = plot_confusion_matrix(bayes_clf, X_test, y_test,
                             display_labels=le_bayes.classes_,
                             cmap=plt.cm.Blues,
                             normalize='true')
disp.ax_.set_title('Bayes Classifier - Confusion matrix')
plt.show()
```

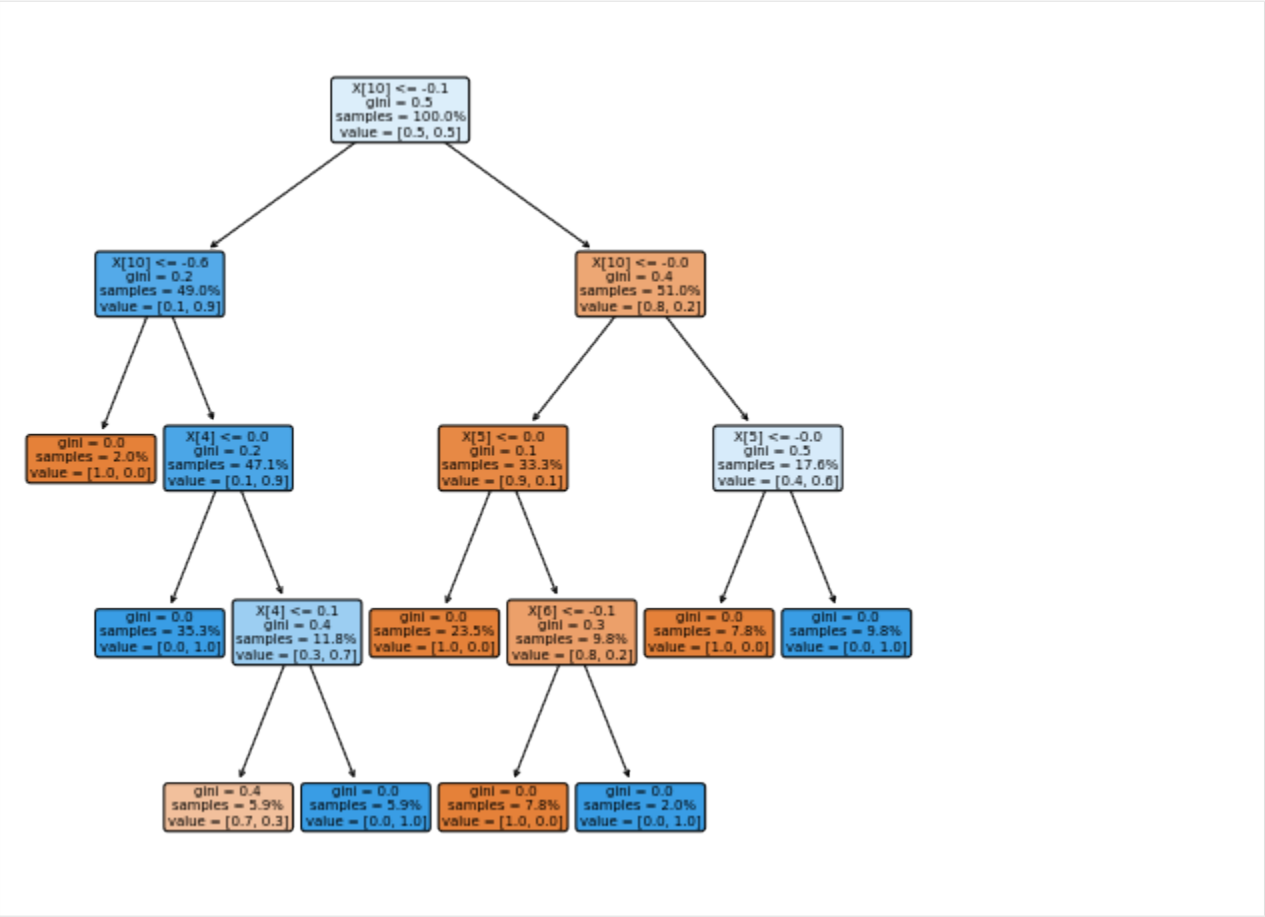


Comments

This one of the most interesting results... note that this same result appered on the XGBoost classifier algorithm. This enables the speculation that the Naive Bayes parameters provide an information so clean about the classes that any algorithm, as complex as possible, or as simple as possible, will be able to find a classification pattern. To examplify this assumption we can plot the tree such as:

[17]:
```
from sklearn import tree

plt.figure(figsize=(8,8))
p_tree = tree.plot_tree(bayes_clf,
                        precision=1,
                        filled=True,
                        proportion=True,
                        rotate=True,
                        rounded=True)
```



4.13.3 Hidden Markov Models

Here we use the model estimated from the Hidden Markov Models library, wich is the estimated A matrix, or the so called transition probability matrices as feature for the learning classifier. For that we must read the pickle file with the desired features:

[18]:
```
import pickle

file_name = './features/hmm_data/nx_20/hmm_data.pkl'
with open(file_name, 'rb') as file:
    hmm_data = pickle.load(file)
hmm_data.keys()
```

[18]:
```
dict_keys(['y', 't', 'labels', 'features'])
```

Manipulate features

After reading the data, it is necessary to create the classical regression structure model in the format $Y = f(\Theta, X)$, normalize the feature data and encode any possible label data into numerical classes. This is just the preparation for the machine leaning algorithm to guarantee that the provided info is properlly designed for any machine learning classical form.

[19]:
```
import numpy as np
from sklearn import preprocessing

# Encode the label
le_hmm = preprocessing.LabelEncoder()
le_hmm.fit( hmm_data['labels'] )

# Define the model order
feat = hmm_data['features']
nx = feat['prob_matrix'][0].shape[0]

regressors = []
for phi in feat['prob_matrix']:
    # Reshape the regressor
    reg = phi.reshape(nx*nx)
    # Add to the regressors
    regressors.append(reg)
# Normalize the regressors
regressors = preprocessing.normalize(regressors)
# Define outputs as encoded variables
outputs = le_hmm.transform(hmm_data['labels'])
```

Train-test data split

Next it is necessary to segregate the data into a set for validation and one for trainning the model.

[20]:
```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(
    regressors, outputs, test_size=0.33, random_state=42)
```

Hyper tunning

We could consider tunning the model hyper parameters to answer questions such as:

- Wich value of `n_estimators` is the best for this model and data?
- Wich cost function is the best to be selected as `objective` for this model?

We could do a hyper search, to find the best hyper parameters for this model, automating the hyper parameter selection. There are several already builded algorithms to optimize this parameter search, and build find with high performance the best parameters, provided a set of possible values. But, to understand what those algorithms actually does, we could once build our own search algorithm...

As an example, lets run a first handly defined hyper parameter tunning using the confusion matrix of the model:

[21]:
```
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import confusion_matrix

# Define the model parameters
param_dist = {
    'random_state' : 0,
    'criterion' : 'gini', # or 'entropy' for information gain
    'max_depth' : 0
}
```

(continues on next page)

(continued from previous page)

```
# Create the range parameters to
# search
max_features_dim = X_train.shape[1]
max_depth = [k + 1 for k in range(max_features_dim)]

# Create the plotting variable
plot_vals = {
    'true': {
        'confirmed targets': [],
        'eclipsing binaries': [],
    },
    'false': {
        'confirmed targets': [],
        'eclipsing binaries': [],
    }
}

# Estimate and validate each candidate
for opt in max_depth:
    # Update the model parameters
    param_dist['max_depth'] = opt
    # Create the xgBoost classifier
    clfs = DecisionTreeClassifier(**param_dist)
    # Fit the model to the data
    clfs.fit(X_train, y_train)
    # Estimate the test output
    y_pred = clfs.predict(X_test)
    # Compute the confusion matrix
    conf_mat = confusion_matrix(
        y_test, y_pred,
        normalize='true')
    # Save the confusion matrix
    plot_vals['true']['confirmed targets'].append(conf_mat[0,0])
    plot_vals['true']['eclipsing binaries'].append(conf_mat[1,1])
    plot_vals['false']['confirmed targets'].append(conf_mat[0,1])
    plot_vals['false']['eclipsing binaries'].append(conf_mat[1,0])
```

[22]:
```
from utils import *

# Line plot each confidence matrix parameter
x_data = [max_depth, max_depth, max_depth, max_depth]
y_data = [plot_vals['true']['confirmed targets'],
          plot_vals['true']['eclipsing binaries'],
          plot_vals['false']['confirmed targets'],
          plot_vals['false']['eclipsing binaries']]
legends= ['True - C.T.', 'True - E.B.', 'False - C.T.', 'False - E.B.']
colors = [6, 7, 2, 3]

p = visual.multline_plot(x_data, y_data,
                         legend_label=legends,
                         title='Hyper parameter search - Confusion parameters plot',
                         color_index=colors,
                         y_axis={'label': 'Proportion'},
                         x_axis={'label': 'n_estimators'})
visual.show_plot(p)
```

Data type cannot be displayed: application/javascript, application/vnd.bokehjs_exec.v0+json

Train model

After running the hyper parameter search we can create a model with the best defined hyper parameters, or setup parameters, and consolidate the model in to the best version for further performance analysis. The model is saved on a particular variable, such as `hmm_clf` to be further used in some vote chain model, if further necessary.

[23]:
```
param_dist = {
    'random_state': 0,
    'criterion' : 'gini', # or 'entropy' for information gain
    'max_depth' : max_depth[0]
}

# Create the model classifier
hmm_clf = DecisionTreeClassifier(**param_dist)

# Train the model
hmm_clf.fit(X_train, y_train)
```

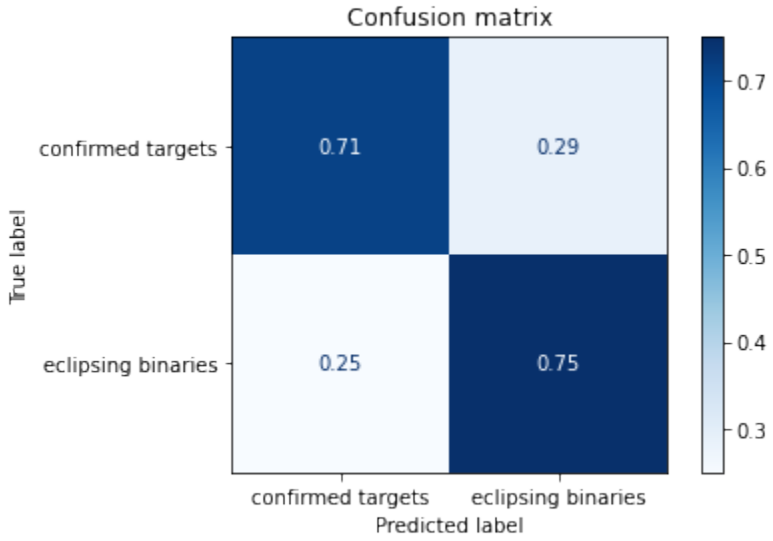
[23]:
```
DecisionTreeClassifier(ccp_alpha=0.0, class_weight=None, criterion='gini',
                       max_depth=1, max_features=None, max_leaf_nodes=None,
                       min_impurity_decrease=0.0, min_impurity_split=None,
                       min_samples_leaf=1, min_samples_split=2,
                       min_weight_fraction_leaf=0.0, presort='deprecated',
                       random_state=0, splitter='best')
```

Results

Here we include some visualization results for the xgBoost algorithm classification. As the first result, we just print the model eval metrics, here the *log loss* of the model, for both the trainning and testing data.

[24]:
```
import matplotlib.pyplot as plt
from sklearn.metrics import plot_confusion_matrix

disp = plot_confusion_matrix(hmm_clf, X_test, y_test,
                             display_labels=le_hmm.classes_,
                             cmap=plt.cm.Blues,
                             normalize='true')
disp.ax_.set_title('Confusion matrix')
plt.show()
```

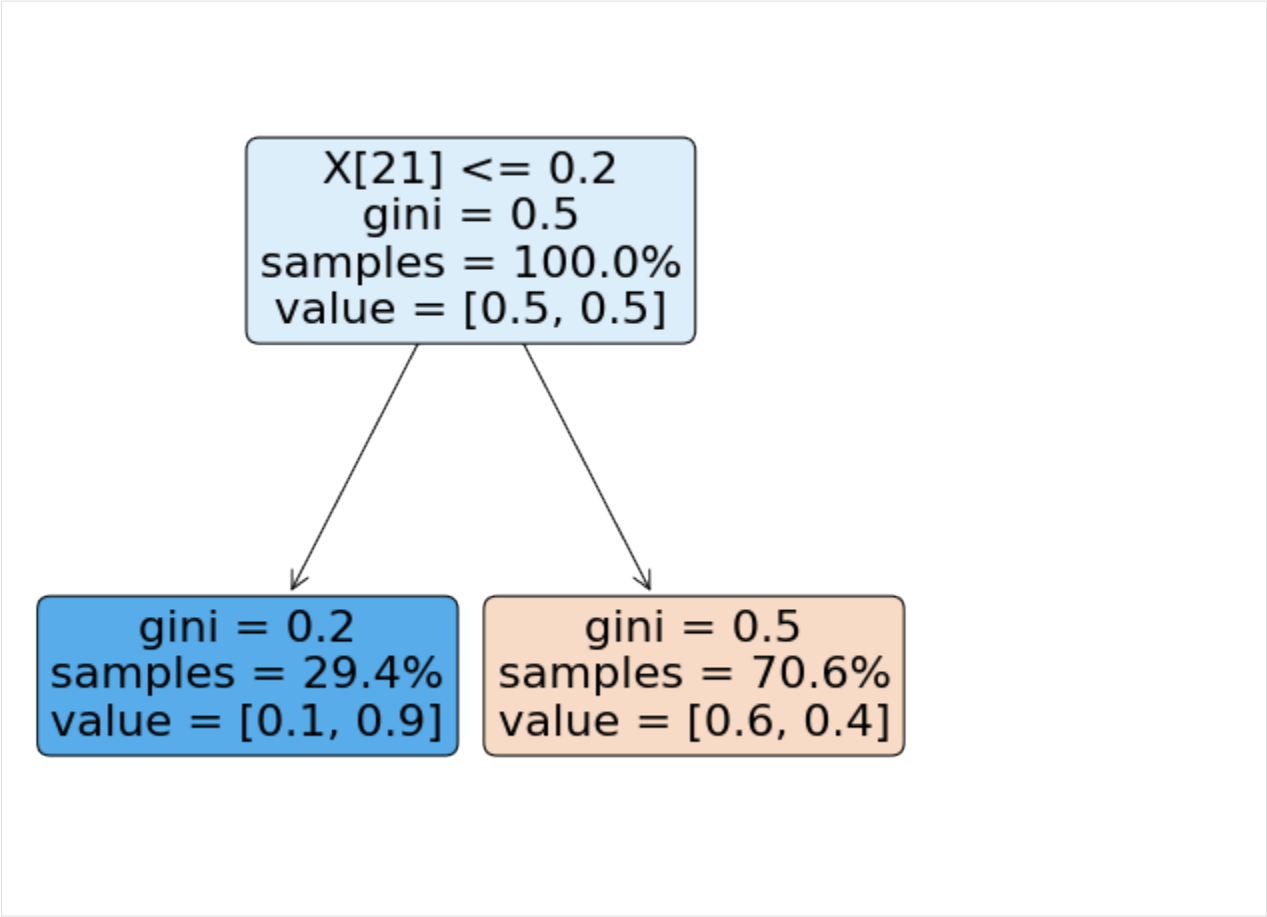


Comments

Note that here, the same happened with the XGBoost classifier... so the conclusion must be the same as the presented one from the Naive Bayes method. The information from the Hidden Markov Models, only contain so much information that only allows the classifier to get to this performance. This is shown by the best complexity found for this algorithm, which is :math:'n_x=1', therefore the simplest model as possible. To illustrate that, we can show the tree such as:

[25]:
```
from sklearn import tree

plt.figure(figsize=(8,8))
p_tree = tree.plot_tree(hmm_clf,
                        precision=1,
                        filled=True,
                        proportion=True,
                        rotate=True,
                        rounded=True)
```



4.14 CoRoT Contributions

4.14.1 utils Package

Intro

copyright 2010 Marcelo Lima

license BSD-3-Clause

Data helper

This module is responsible to read the *.fits* files and return the readed data as a `Curve` objects, which is a more suitable format to read the data. Or one might choose to only receive the data in a simple type as possible such as lists and dictionaries, thus by choosing this method, one will loose the hability to use the inherited computing pre processing features from the `Curve` format.

class utils.data_helper.**Reader**

Bases: object

The reader module is responsible for reading the raw *.fits* files and provide the data in a more friendly data structure for the python environment. It supports both unique *.fits* files such as a batch processing approach,

using folders filled with several *.fits* files.

As simple as it is to call a class, the reader package has no usage complication. For one to import and use it's methods, it is only necessary to do:

```
import utils.data_helper as dh
myReader = dh.Reader()
```

Retrieved data structure

One must understand that the retrieved data will be or `Curve` objects or they will be composed of simple `list` and `dict` outputs. All methods have a particular parameter, with default `simple_out=False` thence returning the data as `Curve`. If one defines the parameter `simple_out=True`, one will receive the obtained data as dictionaries of indexed curves in lists, which is not advisable.

New in version 0.1: The *from_file* and *from_folder* methods where added.

from_file (*path=None*, *label=None*, *index=None*, *feature=None*)

Get the white light curve from the provided *.fits* file, and label this curve with the provided label (or with the folder name, if *label=None*) and return the data_sctruc.Curve.

Parameters

- **path** (*str*) – Paths to the desired *.fits* file
- **label** (*str*) – The label for the returned curve
- **index** (*int*) – The HUD table index
- **feature** (*str*) – The light curve specific feature name

Returns The specific feature of light curve in more suitable format

Return type *data_struc.Curve*

from_folder (*folder=None*, *label=None*, *index=None*, *items=None*)

Get the white light curve of each *.fits* file presented inside the provided folder, label each one or with the provided label, or with the folder name, and return a list with one data_struc.Curve variable for each *.fits* file.

Parameters

- **folder** (*str*) – Paths to the folder with the *.fits* files
- **label** (*str*) – The label for the returned light curves
- **index** (*int*) – The HUD table index
- **items** (*int*) – The amount of random files to read from folder

Returns A list of data_struc.Curve

Return type list

from_txt ()

list_features (*folders=None*)

Get only the features list, for each *.fits* curve file, inside the folders list.

Parameters **folders** (*list*) – Paths to the folders with the *.fits*

Returns The list with features available for each *.fits* found

Return type list

Data structure

This module gathers all the custom data structures created to replace the *.fits* format provided by the raw data-set.

It represents the direct interface from the complex representation of the astropy.table modules, to simple *array* python variables.

class utils.data_struc.**Curve**(*hdu=None*, *index=None*, *label=None*)

Bases: object

This object maintain the light curve HUD table, and represents an interface to get data from the HUD tables of the astropy library and the python environment. Therefore, it just represent a simple interface to transcript the HUD table informations to simple python variables such as *dict*, *list* and *array*.

For example to get create and extract the data from a curve object the user just need to:

```
import utils.data_struc as ds
curve = ds.Curve(hdu=hduTable, index=hduTableIndex, label=curveLabel)
feature = curve['FEATURE NAME']
```

By using the *Reader* this is even more transparent. Here you need to provide

Parameters

- **hdu** (*astronomy.table.Table*) – The HDU table from astronomy library
- **index** (*int*) – The desired HDU table index to be used
- **label** (*str*) – The desired label for this light curve

And then, by just argumenting the column name of the table, the user can get the values of the column as a ndarray variable.

Advantages

One must question why use this particular structure instead of just using the astropy.table.Table objects. But, since in most time we just use the last HDU table (usually *index=2*), this structure automatically removed all other tables and just save the desired one in a memory map format (that has a better performance).

This approach is memory efficient because, the information removed is usually something close to 10 times bigger than the used one... that is, the first and second table are at least ten times bigger than the third table (wich is the most used one, *index=2*). Therefore, for this application, this approach is actually efficient when compared to just loading the data and dealing with the information in astropy tables.

New in version 0.1: The *from_file* and *from_folder* methods where added.

index_tables(*index=None*)

Handly index the HDU table from the Curve object, by replacing the raw_table variable with only the desired HDU table.

Parameters **index** (*int*) – The index of the desired HDU table

julian_to_stdtime()

Change the julian date variable of the HDU tables to standard time representations.

Visualization

This module simplifies the usage of the bokeh library by reducing the amount of code to plot some figures. For example, without this library, to plot a simple line with the bokeh library, one must do something like:

```
from bokeh.palettes import Magma
from bokeh.layouts import column
from bokeh.plotting import figure, show
from bokeh.models import ColumnDataSource
from bokeh.io import output_notebook, push_notebook

p = figure(
    title="Some title",
    plot_width=400,
    plot_height=600)

# Style the figure image
p.grid.grid_line_alpha = 0.1
p.xgrid.band_fill_alpha = 0.1
p.xgrid.band_fill_color = Magma[10][1]
p.yaxis.axis_label = "Some label for y axis"
p.xaxis.axis_label = "Some label for x axis"

# Place the information on plot
p.line(x_data, y_data,
        legend_label="My legend label",
        line_width=2,
        color=Magma[10][2],
        muted_alpha=0.1,
        line_cap='rounded')
p.legend.location = "right_top"
p.legend.click_policy = "disable"

show(p)
```

Wich, with the visual library one might do with just:

```
p = visual.line_plot(x_data, y_data,
                     legend_label='My legend label',
                     title='Some title',
                     y_axis={'label': 'Some label for y axis'},
                     x_axis={'label': 'Some label for x axis'})
visual.show_plot(p)
```

Simple as that... It follows a defualt plotting style for each plot, presented in the */utils/configs/plot.yaml*. And the user just need to pass the parameters that he want to change from this default style. It also provides some pre-computation to make more complex graphs, such as box plots, histograms and so on.

Note: This is just a library to simplify most plots used during the notebooks to not populate the study with unecessary code... The user can use any library desired to do this same plots.

Also the user can change the *plot.yaml* file to set any default plot style that one might want. Please, for that check out the file in */utils/configs/plot.yaml*.

---

utils.visual.**box_plot**(*score=None*, *labels=None*, *opts=None*, ***kwargs*)

Create a Bokeh figure object and populate its box plot properties with the provided information. To create a box plot, you actually need to combine two segment, a vbar, and a rect object from Bokeh. This method already does that for you. It also already computes the statistical median, mean and each quantile.

Parameters

- **score** (*list*) – The list with all values of the distributions

- **labels** (*list*) – The list with the group label for each value of score
- **opts** (*dict*) – The desired options of the *plot.yaml* in dictionary format
- **kwargs** – The desired options of the *plot.yaml* in directive format

Returns A Bokeh figure object with the box plot necessary properties filled

Return type bokeh.Figure

utils.visual.**handle_opts**(*default*, *provided*)
Merge the default (set by the plot.yaml file) and user provided plot options into one dictionary.

Parameters

- **default** (*dict*) – The default style guide dict from *plot.yaml*
- **provided** (*dict*) – The user provided properties

Returns A dict with the merged default and provided plot options

Return type dict

utils.visual.**hist_plot**(*hist=None*, *edges=None*, *opts=None*, ***kwargs*)
Create a Bokeh figure object and populate its histogram plot propertie with the provided information. To create a histogram plot, you actually need to the correct properties of the quad object from Bokeh. This method already does that for you. It also already computes the correct values, and create the bins correctly.

Parameters

- **hist** (*ndarray*) – The hist output from numpy.histogram
- **edges** (*ndarray*) – The histogram edges output from numpy.histogram
- **opts** (*dict*) – The desired options of the plot.yaml in dictionary format
- **kwargs** – The desired options of the plot.yaml in directive format

Returns A Bokeh figure object with the line properties filled

Return type bokeh.Figure

utils.visual.**line_plot**(*x_data=None*, *y_data=None*, *opts=None*, ***kwargs*)
Create a Bokeh figure object and populate its line propertie with the provided information.

Parameters

- **x_data** (*ndarray*) – The ndarray with x axis values
- **y_data** (*ndarray*) – The ndarray with y axis values
- **opts** (*dict*) – The desired options of the *plot.yaml* in dictionary format
- **kwargs** – The desired options of the *plot.yaml* in directive format

Returns A Bokeh figure object with the line properties filled

Return type bokeh.Figure

utils.visual.**multline_plot**(*x_data=None*, *y_data=None*, *opts=None*, ***kwargs*)
Create a Bokeh figure object and populate a line object of the bokeh library for each line data provided in the y_data list parameter of this function.

Parameters

- **x_data** (*list*) – The list with a ndarray data for the x axis of each line
- **y_data** (*list*) – The list with a ndarray data for the y axis of each line

- **opts** (*dict*) – The desired options of the *plot.yaml* in dictionary format
- **kwargs** – The desired options of the *plot.yaml* in directive format

Returns A Bokeh figure object with the line properties filled

Return type bokeh.Figure

utils.visual.**show_plot**(**args*)

This function shows the figures provided as arguments by default, in a column manner.

Parameters **args** – The bokeh.Figure objects to be show in a figure

4.15 Vanderlei Cunha Parro

Find me on LinkedIn Find me on GitHub

Send me a message Watch my channel!

4.16 Marcelo Mendes Lafetá Lima

Find me on LinkedIn Find me on GitHub Send me a message

4.17 Roberto Bertoldo Menezes

Send me a message

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Python Module Index

u

Index