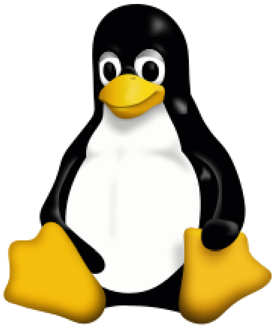
1. Introduction

(https://commons.wikimedia.org/wiki/File:Tux.svg)



Version control repositories like CVS, Subversion or Git can be a real gold mine for software developers. They contain every change to the source code including the date (the "when"), the responsible developer (the "who"), as well as a little message that describes the intention (the "what") of a change.

In this notebook, we will analyze the evolution of a very famous open-source project – the Linux kernel. The Linux kernel is the heart of some Linux distributions like Debian, Ubuntu or CentOS. Our dataset at hand contains the history of kernel development of almost 13 years (early 2005 - late 2017). We get some insights into the work of the development efforts by

- identifying the TOP 10 contributors and
- visualizing the commits over the years.

In [204]:

```python
# Printing the content of git_log_excerpt.csv
with open("./datasets/git_log_excerpt.csv") as f:
    print(f.read())
```

```
1502382966#Linus Torvalds
1501368308#Max Gurtovoy
1501625560#James Smart
1501625559#James Smart
1500568442#Martin Wilck
1502273719#Xin Long
1502278684#Nikolay Borisov
1502238384#Girish Moodalbail
1502228709#Florian Fainelli
1502223836#Jon Paul Maloy
```

In [205]:

```
%%nose

def test_listing_of_file_contents():

    # FIXME1: if student executes cell more than once, variable _i2 is then not defined. Solution?

    #PATH = "datasets/git_log_excerpt.csv"
    # hard coded cell number: maybe a little bit fragile
    #cell_input_from_sample_code = _i2
    #assert PATH in cell_input_from_sample_code, \
    #"The file %s should be read in." % PATH

    # FIXME2: can't access the sample code cell's output here because of the use of 'print'

    # test currently deactivated: too hard to create a table test case
    assert True
```

Out[205]:

```
1/1 tests passed
```

2. Reading in the dataset

The dataset was created by using the command `git log --encoding=latin-1 --pretty="%at#%aN"` in late 2017. The `latin-1` encoded text output was saved in a header-less CSV file. In this file, each row is a commit entry with the following information:

- `timestamp` : the time of the commit as a UNIX timestamp in seconds since 1970-01-01 00:00:00 (Git log placeholder " `%at` ")
- `author` : the name of the author that performed the commit (Git log placeholder " `%aN` ")

The columns are separated by the number sign `#` . The complete dataset is in the `datasets/` directory. It is a `gz` -compressed csv file named `git_log.gz` .

In [206]:

```python
# Loading in the pandas module as 'pd'
import pandas as pd

# Reading in the log file
git_log = pd.read_csv(
    'datasets/git_log.gz',
    sep ='#',
    encoding='latin-1',
    header=None,
    names=['timestamp', 'author']
)

# Printing out the first 5 rows
git_log.head()
```

Out[206]:

	timestamp	author
0	1502826583	Linus Torvalds
1	1501749089	Adrian Hunter
2	1501749088	Adrian Hunter
3	1501882480	Kees Cook
4	1497271395	Rob Clark

In [207]:

```
%%nose


def test_is_pandas_loaded_as_pd():

    try:
        pd # throws NameError
        pd.DataFrame # throws AttributeError
    except NameError:
        assert False, "Module pandas not loaded as pd."
    except AttributeError:
        assert False, "Variable pd is used as short name for another module."


def test_is_git_log_data_frame_existing():

    try:
        # checks implicitly if git_log by catching the NameError exception
        assert isinstance(git_log, pd.DataFrame), "git_log isn't a DataFrame."

    except NameError as e:
        assert False, "Variable git_log doesn't exist."


def test_has_git_log_correct_columns():

    expected = ['timestamp', 'author']
    assert all(git_log.columns.get_values() == expected), \
        "Expected columns are %s" % expected


def test_is_logfile_content_read_in_correctly():

    correct_git_log = pd.read_csv(
        'datasets/git_log.gz',
        sep='#',
        encoding='latin-1',
        header=None,
        names=['timestamp', 'author'])

    assert correct_git_log.equals(git_log), \
        "The content of datasets/git_log.gz wasn't correctly read into git_log. Check the parameters of read_csv."
```

Out[207]:

```
4/4 tests passed
```

3. Getting an overview

The dataset contains the information about every single code contribution (a "commit") to the Linux kernel over the last 13 years. We'll first take a look at the number of authors and their commits to the repository.

In [208]:

```
# calculating number of commits
number_of_commits = len(git_log)

# calculating number of authors
number_of_authors = len(git_log['author'].dropna().unique())

# printing out the results
print("%s authors committed %s code changes." % (number_of_authors, numb
er_of_commits))
```

```
17385 authors committed 699071 code changes.
```

In [209]:

```
%%nose

def test_basic_statistics():
    assert number_of_commits == len(git_log), \
    "The number of commits should be right."
    assert number_of_authors == len(git_log['author'].dropna().unique
()), \
    "The number of authors should be right."
```

Out[209]:

```
1/1 tests passed
```

4. Finding the TOP 10 contributors

There are some very important people that changed the Linux kernel very often. To see if there are any bottlenecks, we take a look at the TOP 10 authors with the most commits.

In [210]:

```
# Identifying the top 10 authors
top_10_authors = git_log['author'].value_counts().head(10)

# Listing contents of 'top_10_authors'
top_10_authors.head(10)
```

Out[210]:

```
Linus Torvalds           23361
David S. Miller           9106
Mark Brown                6802
Takashi Iwai              6209
Al Viro                   6006
H Hartley Sweeten         5938
Ingo Molnar               5344
Mauro Carvalho Chehab     5204
Arnd Bergmann             4890
Greg Kroah-Hartman        4580
Name: author, dtype: int64
```

In [211]:

```
%%nose


def test_is_series_or_data_frame():

    assert isinstance(top_10_authors, pd.Series) or isinstance(top_10_au
thors, pd.DataFrame), \
    "top_10_authors isn't a Series or DataFrame, but of type %s." % type
(top_10_authors)


def test_is_result_structurally_alright():

    top10 = top_10_authors.squeeze()
    # after a squeeze(), the DataFrame with one Series should be convert
ed to a Series
    assert isinstance(top10, pd.Series), \
    "top_10_authors should only contain the data for authors and the num
ber of commits."


def test_is_right_number_of_entries():

    expected_number_of_entries = 10
    assert len(top_10_authors.squeeze()) is expected_number_of_entries,
 \
    "The number of TOP 10 entries should be %r. Be sure to store the res
ult into the 'top_10_authors' variable." % expected_number_of_entries


def test_is_expected_top_author():

    expected_top_author = "Linus Torvalds"
    assert top_10_authors.squeeze().index[0] == expected_top_author, \
    "The number one contributor should be %s." % expected_top_author


def test_is_expected_top_commits():
    expected_top_commits = 23361
    assert top_10_authors.squeeze()[0] == expected_top_commits, \
    "The number of the most commits should be %r." % expected_top_commit
s
```

Out[211]:

```
5/5 tests passed
```

5. Wrangling the data

For our analysis, we want to visualize the contributions over time. For this, we use the information in the `timestamp` column to create a time series-based column.

In [212]:

```
# converting the timestamp column
git_log['timestamp'] = pd.to_datetime(git_log['timestamp'], unit="s")

# summarizing the converted timestamp column
git_log['timestamp'].describe()
```

Out[212]:

```
count                      699071
unique                     668448
top       2008-09-04 05:30:19
freq                           99
first     1970-01-01 00:00:01
last      2037-04-25 08:08:26
Name: timestamp, dtype: object
```

In [213]:

```
%%nose

def test_timestamps():

    START_DATE = '1970-01-01 00:00:01'
    assert START_DATE in str(git_log['timestamp'].min()), \
    'The first timestamp should be %s.' % START_DATE

    END_DATE = '2037-04-25 08:08:26'
    assert END_DATE in str(git_log['timestamp'].max()), \
    'The last timestamp should be %s.' % END_DATE
```

Out[213]:

```
1/1 tests passed
```

6. Treating wrong timestamps

As we can see from the results above, some contributors had their operating system's time incorrectly set when they committed to the repository. We'll clean up the `timestamp` column by dropping the rows with the incorrect timestamps.

In [214]:

```
# determining the first real commit timestamp
first_commit_timestamp = git_log.iloc[-1]['timestamp']

# determining the last sensible commit timestamp
last_commit_timestamp = pd.to_datetime('2018')

# filtering out wrong timestamps
corrected_log = git_log[
    (git_log['timestamp'] >= first_commit_timestamp) &
    (git_log['timestamp'] <= last_commit_timestamp)]

# summarizing the corrected timestamp column
corrected_log['timestamp'].describe()
```

Out[214]:

```
count                   698569
unique                  667977
top        2008-09-04 05:30:19
freq                        99
first      2005-04-16 22:20:36
last       2017-10-03 12:57:00
Name: timestamp, dtype: object
```

In [215]:

```
%%nose

def test_corrected_timestamps():

    FIRST_REAL_COMMIT = '2005-04-16 22:20:36'
    assert FIRST_REAL_COMMIT in str(corrected_log['timestamp'].min()), \
    'The first real commit timestamp should be %s.' % FIRST_REAL_COMMIT

    LAST_REAL_COMMIT = '2017-10-03 12:57:00'
    assert LAST_REAL_COMMIT in str(corrected_log['timestamp'].max()), \
    'The last real commit timestamp should be %s.' % LAST_REAL_COMMIT
```

Out[215]:

```
1/1 tests passed
```

7. Grouping commits per year

To find out how the development activity has increased over time, we'll group the commits by year and count them up.

In [216]:

```
# Counting the no. commits per year
commits_per_year = corrected_log.groupby(
    pd.Grouper(key='timestamp', freq='AS')).count()

# Listing the first rows
commits_per_year.head()
```

Out[216]:

	author
timestamp	
2005-01-01	16229
2006-01-01	29255
2007-01-01	33759
2008-01-01	48847
2009-01-01	52572

In [217]:

```
%%nose

def test_number_of_commits_per_year():

    YEARS = 13
    assert len(commits_per_year) == YEARS, \
    'Number of years should be %s.' % YEARS


def test_new_beginning_of_git_log():

    START = '2005-01-01 00:00:00'
    assert START in str(commits_per_year.index[0]), \
    'DataFrame should start at %s' % START
```

Out[217]:

```
2/2 tests passed
```

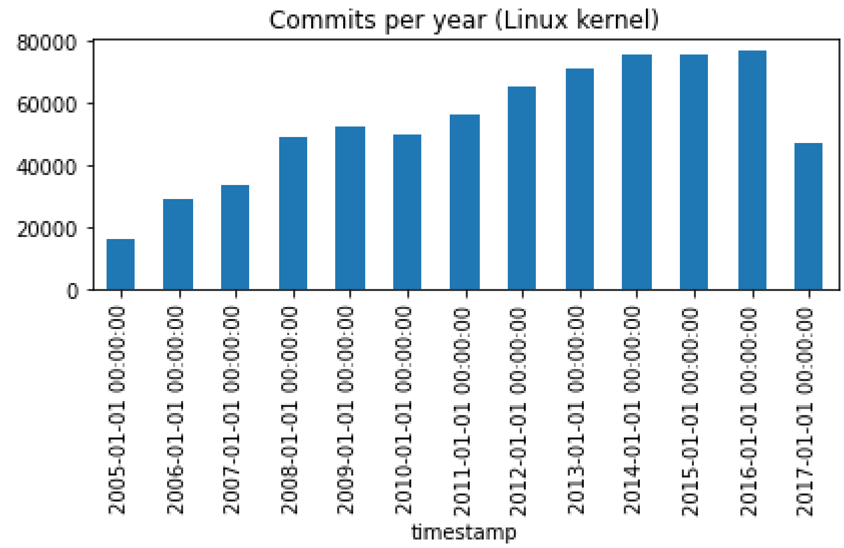
8. Visualizing the history of Linux

Finally, we'll make a plot out of these counts to better see how the development effort on Linux has increased over the the last few years.

In [218]:
```
# Setting up plotting in Jupyter notebooks
%matplotlib inline

# plot the data
commits_per_year.plot(kind='bar', title="Commits per year (Linux kerne
l)", legend=False)
```

Out[218]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x7f4da1909e48>
```



In [219]:
```
%%nose

def test_call_to_plot():

    # FIXME: Different results local and on build server.
    # - local (expected): AssertionError: Plot type should be a bar char
t.
    # - build server: NameError: name '_i20' is not defined
    # deactivating tests

    #assert "kind='bar'" in _i20, "Plot type should be a bar chart."

    # test currently deactivated: too hard to create a table test case
    assert True
```

Out[219]:
```
1/1 tests passed
```

9. Conclusion

Thanks to the solid foundation and caretaking of Linux Torvalds, many other developers are now able to contribute to the Linux kernel as well. There is no decrease of development activity at sight!

In [220]:
```
# calculating or setting the year with the most commits to Linux
year_with_most_commits = 2016
```

In [221]:

```
%%nose

def test_year_with_most_commits():
    assert str(year_with_most_commits).endswith("16") , \
        "Write the year with the most commits as 20??, but with ?? repla
ced."
```

Out[221]:

```
1/1 tests passed
```