MACHINE LEARNING FOR SOFTWARE ENGINEERS

ENSF611: Lecture Notes

Dr. Vikram Kumar

September 6, 2022

Contents

1 Overview

1.1 Goals of this course

1. Produce code using machine learning libraries.
2. Select and construct machine learning pipelines for engineering tasks.
3. Analyze machine learning model evaluation outcomes to optimize performance.
4. Document machine learning workflow and results.

1.2 Textbooks

Introduction to Machine Learning with Python, Müller and Guido, 1st ed, 2016 `https://github.com/amueller/introduction_to_ml_with_python`
and use parts from: Python Data Science Handbook, VanderPlas, 1st ed, 2016 `https://github.com/jakevdp/PythonDataScienceHandbook`
both books are available online through library.ucalgary.ca

1.3 Installing tools

We need:

- Anaconda Python 3.10 (or previous Anaconda Python)

`https://www.anaconda.com/distribution/`

- git (with gitbash on Windows)

`https://git-scm.com/book/en/v2/Getting-Started-Installing-Git`

- Github account

1.4 Setting up conda environment

To setup Anaconda Python in a terminal or *Anaconda prompt* on Windows:

```
$ conda create --name ensf-ml python=3.10 anaconda
$ conda activate ensf-ml
$ conda install -c conda-forge graphviz python-graphviz
$ conda install -c districtdatalabs yellowbrick=1.3
$ pip install mglearn
(optional)$ pip install nbdime
```

1.5 Python and Jupyter notebooks

With anaconda python, jupyter notebooks are available. In a terminal or *Anaconda prompt* on Windows, start a jupyter notebook server with:

```
$ conda activate ensf-ml
$ jupyter notebook
```

For a quick introduction to notebooks, see: `https://realpython.com/jupyter-notebook-introduction/`

Alternatively, to get the next generation interface with *JupyterLab* use:

```
$ conda activate ensf-ml
$ jupyter lab
```

Once a new notebook is started up, you can test your installation by pasting the following code into the first cell and running it (shift-enter):

```
import sys
print("Python␣version:", sys.version)

import pandas as pd
print("pandas␣version:", pd.__version__)

import matplotlib
print("matplotlib␣version:", matplotlib.__version__)

import numpy as np
print("NumPy␣version:", np.__version__)

import scipy as sp
print("SciPy␣version:", sp.__version__)

import IPython
print("IPython␣version:", IPython.__version__)
```

```
import sklearn
print("scikit-learn␣version:", sklearn.__version__)

import yellowbrick
print("yellowbrick␣version:", yellowbrick.__version__)
```

It will print the versions installed. My output is:

```
Python version: 3.8.8 (default, Apr 13 2021, 12:59:45)
[Clang 10.0.0 ]
pandas version: 1.2.4
matplotlib version: 3.3.4
NumPy version: 1.19.2
SciPy version: 1.6.2
IPython version: 7.22.0
scikit-learn version: 0.24.1
yellowbrick version: 1.3
```

2 Working with git

2.1 References

First, there is the git book: `https://git-scm.com/book/en/v2`, a complete reference to git.

When I first learned git, I followed the git immersion online tutorial available at: `http://gitimmersion.com`

The Software Carpentry course `http://swcarpentry.github.io/git-novice/` provides another step-by-step tutorial to learn git.

A short and useful introduction to git can be found in Section 2 of lecture notes available here: `http://columbia-applied-data-science.github.io/appdatasci.pdf`

One of the references used in this chapter, `http://marklodato.github.io/visual-git-guide/index-en.html`, provides a visual guide to git.

2.2 Configure git

Prior to using git on a fresh installation, it is important to configure name, email and line endings. Configurations has to be done only once. Name and email will be used to match commits on remote services like github. Line endings configurations makes sure that contributions made from Windows and unix-like systems are handled properly. To configure name and email type the following in a terminal (or git-bash on windows):

```
$ git config --global user.name "your␣name"
$ git config --global user.email "your_email@whatever.com"
```

Note that email address should match the email address you used on github. You can also use

`ID+your-user-name@users.noreply.github.com`,

if you want to keep your email private. Check settings -> emails on github.com. When you check *keep my email address private* it will show your noreply email address.

Additionally, I like to have an alias configured to view commit history[1]:

```
$ git config --global alias.hist \
"log␣--pretty=format:'%h␣%ad␣|␣%s%d␣[%an]'␣\
--graph␣--date=short"
```

Maybe you will like it too. This will allow viewing previous commits with `git hist` in a concise format.

2.3 Tracking changes

As a little exercise, why don't we use git to track code changes for a HelloWorld example. Assuming you have `hello_orld.py` and your terminal (git-bash on Windows) open in this directory. `ls` shows `hello_world.py`. Its contents are:

```
# Author: my name

# Greeting the world
print('Hello␣World!')
```

To start tracking changes with git, we first have to initialize the repository with `git init`. This command will create a folder `.git`, you can check with `ls -a`. Now, the repository is initialized. No files are yet under version control. `git status` shows, yes, the current status. Let's check-in version one of the code. This is a two stage process: first add the change to the index (also called stage), second, commit the staged changes to the repository, the history, pointed to by a reference called HEAD. A commit includes a short message. The commands are:

```
 $ git status
On branch master

No commits yet
```

[1] `http://gitimmersion.com/lab_11.html`

```
Untracked files:
  (use "git␣add␣<file>..." to include in what will be committed)

    hello_world.py

nothing added to commit but untracked files present (use "git␣add" to track)

  $ git add hello_world.py

  $ git status
On branch master

No commits yet

Changes to be committed:
  (use "git␣rm␣--cached␣<file>..." to unstage)

    new file: hello_world.py

  $ git commit -m 'Initial commit'
[master (root-commit) 38e9b3a] Initial commit
 1 file changed, 13 insertions(+)
 create mode 100644 hello_world.py

  $ git status
On branch master
nothing to commit, working tree clean
```

The three status commands in between are not necessary, but it is good to see what happens to the repository after each command. These different layers: working directory, stage and history can be confusing at first. The visual representation might help `http://marklodato.github.io/visual-git-guide/index-en.html`. OK, now we make some changes, edit author name. To review the changes use `git diff`. We can now add and commit:

```
$ git status
On branch master
Changes not staged for commit:
  (use "git␣add␣<file>..." to update what will be committed)
  (use "git␣checkout␣--␣<file>..." to discard changes in working directory)

    modified: hello_world.py
```

```
no changes added to commit (use "git␣add" and/or "git␣commit␣-a")

$ git add hello_world.py
$ git commit -m 'edited author name'
[master c11152f] edited author name
 1 file changed, 1 insertion(+), 1 deletion(-)
```

Let's see what we have so far:

```
$ git hist
*c11152f 2020-05-08 | edited author name (HEAD -> master) [Yves]
*38e9b3a 2020-05-08 | Initial commit [Yves]
```

The latest commit appears at the top, where HEAD is. master denotes the branch we are on. Each commit has a unique hash, our history shows the tail of this hash.

Notice that git status includes additional information related to undoing steps. Undoing is explained in the visual guide `http://marklodato.github.io/visual-git-guide/index-en.html`. In summary:

- `git reset -- filename` un-stages, it copies the version in HEAD to the stage (index), and un-does staging with `git add -- filename`.
- `git checkout -- filename` copies from stage (index) to working directory, and un-does working directory changes.
- `git reset HEAD~1 --hard` will undo the last commit. HEAD~1 can be thought of HEAD minus one, go back one commit.

In the next section we will look at working with remotes. When working with remotes, the preferred way to undo the last commit is to use `git revert HEAD`, indicating the commit to be reverted, rather than `git reset HEAD~1 --hard`, moving HEAD. Reverting creates an inverse change as a new commit, hence, playing nicely with remote repositories.

It is a good idea to practice committing and undoing commits in a toy repository. To remove git from a folder, delete the `.git` folder.

2.4 Working with remotes

In the previous section, we looked at tracking your changes with git on the local machine. Often there is a desire to have a copy remotely. Synchronizing your code history with a remote host makes working in teams easier. However, even if you work alone on a project, having a copy remotely adds redundancy.

There are providers such as github, gitlab, and bitbucket, which provide remote integration of git. We will look at github here as an example. In our scenario, we start by creating a new repository on the remote host first. In our case this is github. Once the repository is created, we clone it using the repository url with the form `git clone https://github.com/org-a/repo-a.git`. This command checks-out a copy to the local machine and sets up mechanisms to get updates from the remote location git pull, as well as, sending changes there git push.

The complete workflow looks like this:

- Create repository on remote host, e.g. github.
- `git clone <repo-url>`:

Using git-bash (Windows) or terminal (Mac/Linux), clone the repository using the repository url.

- Work on the code:
  - Make changes.
  - `git add <file(s)>` to stage changes.
  - ~git commit -m 'a short message'~ to save changes to the history.
- `git push` Update code on the remote host.
- `git pull` Download and merge code from the remote host.

An alternative would be to `git fetch` first to download, then `git merge` to incorporate changes into the local branch.

2.5 Viewing changes in Jupyter notebooks

If you have installed the `nbdime` package (see Section 1.4), you can use `nbdiff your-notebook.ipynb` or `nbdiff-web your-notebook.ipynb` to see un-commited changes of your notebook. When launching `jupyter lab`, there should be a button *git* that will show the same as `nbdiff-web` in the jupyter window.

For more information, check the online documentation: `https://nbdime.readthedocs.io/en/latest/`.

3 Introduction

3.1 Goals

- Introduce the course:
  - anaconda python, git, github, Jupyter notebooks
  - Two textbooks
  - Quizzes on D2L
  - Assignments for ENSF 410, assignments + project for ENSF 611 using github.
- Introduce machine learning:
  - AI->Machine learning->Deep learning
  - Machine learning: Supervised, unsupervised and reinforcement learning.
  - In this course: non-neural methods of supervised and unsupervised learning
  - Do machines learn? We optimize model parameters to fit a mathematical function to data.

3.2 Train your first model

Following the five steps from Ch 5 in *Data Science Handbook*:

1. Load and arrange data into feature matrix and target vector
2. Choose model class
3. Instantiate model
4. Fit model to data
5. Predict values for new data and evaluate results.

Highlight that feature matrix X has N rows with measurements, and M columns with features. In other words, `X.shape` is `(n_samples, n_features)`.

Highlight that the target vector y has N elements with lables for each of the measurements. In other words, `y.shape` is `(n_samples, )`.

As post-reading, might be good to read *Data Science Handbook* Chapter 5 Machine Learning p.331-342.

4 Review

4.1 Anaconda Python, git, github

See above.

4.2 Jupyter notebooks

See above.

4.3 Numpy, Pandas, Matplotlib/Seaborn

This is what lab0 is about.

4.4 Getting data

In the past, we were mostly given csv files. To do interesting things, we need to become good at getting other data. Sources are UCI, Kaggle, libraries, etc.

5 ML overview

5.1 Goal

Walking through the steps from a high level without explaning the details yet.

5.2 Content

Following Ch. 5 in Data Science Handbook.

Repeat the 5 steps (above) for:

- Supervised regression
- Supervised classification
- Dimensionality reduction
- Clustering

The three datasets: training, validation, testing. Using cross-validation.
Model complexity, overfitting and underfitting, bias and variance.
Effect of training data, learning curves.
Onehot encoding.
Combine PCA and modelling.

6 Linear models

6.1 Goals

- Introduce linear and logistic regression as two base models for regression and classification, respectively.
- Introduce optimization, gradient descent.
- Introduce l1 and l2 regularization.

7 Decision trees

7.1 Goals

- Introduce the concept of decision trees.
- Explain how regression and classification are done with trees.
- Expand to random forest (ensamble method)
- Boosted trees.