

W4995 Applied Machine Learning

Tools and infrastructure

01/23/17

Andreas Müller

Hey. Welcome to the second lecture on applied machine learning. Last week we talked a bit about very general and high-level issues in machine learning. Today, we'll go right down to the metal. Actually, things we will talk about today are more generally about software engineering. If you want to build real-world systems, software engineering practices are really incredibly important.

For those of you from CS, I hope this is a repetition, for those of you from DSI, this is probably new.

We'll talk about version control, in particular git, about some basic python, about testing, continuous integration and about documentation.

Also, I apologize in advance for the slides. There's a lot of bullet points and few diagrams and I don't like that, but I didn't have a lot of time.

So you think you know git?

So this first part is about git. I assume you all know about git. Who here has not used git so far?

If you haven't this might be a bit steep for you, and you should read up on it later. Let me know if I'm going to fast.

I assume that you know what version control is and why it's important. You should really use version control any time you work on any code or any other document. And it looks like we'll have to use git for now and some time to come.

So you all have used git, but who of you thinks they understand git? Ok I'll keep you in mind and ask you trick questions along the way.



Here's a comic from xkcd that might seem familiar. I know many people that use git by just memorizing some commands and if things go south, they just delete the repository and start fresh.

Who of you have done that? I certainly have.

The goal of today is that you never have to throw your hands up ever again.



Personally I like to think that git was a practical joke played on the open source community by Linux Torvals.

Unfortunately, I think the real issue is that kernel developers, genius as they might be, are not the best at creating user interfaces. Git is great, but the user interface is horrible. But let's try and understand what's going on.

git Basics

- Repository

- \$ git init

- \$ rm .git

- Commit

- Remote

There are some basic units of git that I want to go through. The first is a repository. What's a repository?

A folder whose content we want to track with version control.

And how can you create one?

Run git init in a folder, right

And how can you make a folder not a repository anymore?

By removing the .git subfolder. That's it.

A repository is entirely self-contained. There is no service or database or anything like that, it's all plain files within that folder. That also means if you remove the folder, you lose everything if you don't have a backup.

git Basics

- Repository

```
$ git init
```

```
$ rm .git
```

- Commit

- Remote

The next basic concept is a commit. What's that?

It's a snapshot of the state of the folder, with a message attached, and identified with an ID. It's also a node in a graph that describes the history of the snapshot, and usually has one or two parents, and can have arbitrary many children.

The ID is a hash of the state and all the parents, so if you change anything, you change the hash.

The last basic concept I want to talk about is the remote. That's just a pointer to another repository, often github. You can synchronize with that by pushing or pulling changes.

Typical Workflow

- Clone
- Branch
- Add, Commit, add add, commit, add commit
- Merge
- push

So here's a typical workflow for making a change to an existing project. You clone the project from some remote repository. You change some files, add them, and commit them.

Then you push them to the remote repository.

That's more or less what you'll do with your homework.

The advanced version of this is after cloning you create a branch, you make all your changes on the branch, and once you're done, you merge the changes from your branch into the master branch.

Everybody good so far?

I want to give some basic tips that will help you even with these simple workflows.

Some tips

- Git status
- Install shell plugins to show branch (oh-my-zsh)

```
/home/andy/checkout/scikit-learn [git::master *] [andy@dsi-amueller] [16:35]  
> █
```

- Set your editor, pager and diff-tool

You should always call git status, to see what's happening with your repository. It will tell you whether you changed anything, and which branch you're on.

I actually highly recommend using a plugin for your shell, that will give you the status in every line. If you use zsh, you can use oh-my-zsh for example. If you use something else like bash, there are also plenty of options out there.

You should also set your editor and pager to something that is familiar to you. The default editor is vim, and if you're not a vim user, you might want to change that. It's also helpful to set up a diff program that you like.

Git log

```
/home/andy/checkouts/scikit-learn [git::master *] [andy@dsi-anue] [16:43]
git log --oneline --decorate --all --graph -n 50
* 9616ac7 (HEAD -> master, upstream/master) CI remove obsolete comment
* 7978119 [MRG] #8218: in FAQ, link deep learning question to GPU question (#8228)
* b6305ce TST/FIX Add check for estimator: parameters not modified by 'fit' (#7846)
* aaebee1 FIX Issue #8173 - pass n_neighbors in MI computation (#8181)
* 4826883 Call sorted on lfw folder path contents (#7648)
* 4907029 [MRG+3] FIX Memory leak in MAE; Use safe_realloc; Acquire GIL only when raising; Propagate all errors to python interpreter level (#7811) (#80)
* 08772c4 FIX Ensure coef_ is an ndarray when fitting LassoLars (#8160)
* 5d6460d MNT/BLD Use GitHub's merge refs to test PRs on CircleCI (#8211)
* 66443aa [MRG+1] Add prominent mention of Laplacian Eigenmaps (#8155)
* bddda7b [MRG+2] [MAINT] Update to Sphinx-Gallery 0.1.7 (#7986)
* aea6462 [MRG+1] Fixes #8198 - error in datasets.make_moons (#8199)
* 0eb33ad TRAVIS fix flake8 diff.sh check files (#8208)
* eabae3f DOC add missing parentheses in TfIdfTransformer docstring
* 3dc822f DOC additional fixes to 20 newsgroups to prevent TypeError (#8204)
* cbddb92 removed stray space in '_main_' (#8203)
* ca6870a Upgrade html documentation to jupyter v3.1.1 (#8145)
* 04cc67b Clarify error message for min_samples_split. (#8167)
* 2a1408a fixing typo in cs_mse_path deprecation (#8176)
* fd84a56 [MRG+1] Fix the cross_val_predict function for method='predict_proba' (#7889)
* 4910e11 DOC Fix link (#8171)
* 8998856 Fix Ridge Floating point instability (#8154)
* 21775a1 [MRG+1] Add fowlkes-mallows and other supervised cluster metrics to SCORERS dict so it can be used in hyper-param search (#8117)
* 8695ff5 [MRG+1] add partial fit to multioutput module (#8054)
* 0b02125 [MRG] FIX Avoid default mutable argument in constructor of AgglomerativeClustering (#8153)
* d0ce009 [MRG+2] Avoid failure in first iteration of RANSAC regression (#7914)
* e874398 [MRG+1] DOC: complete list of online learners (#8152)
* e0c50fe FIX sphinx gallery rendering of plot digits pipe example
* 84349a7 [MRG+1] Deprecate ridge alpha param on SparsePCA.transform() (#8137)
* c49ced9 DOC: updating GridSearchCV's n_jobs parameter (#8106)
* 2cb7e47 [MRG+1] fowlkes_mallows score: more unit tests (Fixes #8101) (#8140)
* 543b056 [MRG+1] Add DBSCAN support for additional metric params (#8139)
* d7e77ce [MRG+1] Fix "cite us" link in sidebar (#8142)
* 288827b [MRG] update copyright years for 2017 (#8138)
* e2adb77 DOC Fix typo in FAQ (#8132)
* 986a49b FIX Split data using safe_split in permutation test score (#5697)
* ab1c4d4 [MRG+1] Catch cases for different class size in MLPClassifier with warm start (#7976) (#8035)
* dc124b9 (origin/repr_give_up, repr_give_up) i have no idea what I'm doing
* b25fa4b pep8
* 8b38325 playing around, then giving up
* 0d1398a (upstream/ignore_lambda_to_diff_errors) MNT Ignore E731: Use a def instead of lambda
//
d97d13e DOC add sklearn-crfsuite to related projects (#7878)
* fcb706a [MRG+3] Fused types for MultiTaskElasticNet (#8061)
* 096a9cb [MRG] MAINT Python 3.6 fixes (#8123)
* e08868b DOC Fix indentation errors and username links (#8121)
* 4f3c60c [MRG+2] FIX IsolationForest(max_features=0.8).predict(X) fails input validation (#5757)
```

You should also become friends with git log. Git log allows you to view the history of your repository. It has a couple of very helpful options. Plain git log will have very long output and show full commit messages.

If you want short summaries, use oneline.

Often, it's also useful to annotate branches, which you can do with the decorate option.

If you want to show more than just the branch you're on, you can use the all option

Now it's a bit hard to track the relations of the different commits, though, and you might want to use the graph option to show the structure of the history.

You might want to alias a command like that because it's rather long to type, but very informative. I'll show you an alternative in a bit.

**I DON'T DO GIT LOG ALL THE TIME,
BUT WHEN I DO JUST REMEMBER**



**(A DOG) --ALL --DECORATE --ONELINE --
-GRAPH**

memegenerator.net

Understanding Git!

- Working directory
- Index
- Staging area
- Branches
- Head

So now, I want to work towards really understanding git, and here are some lower level concepts that are important.

First, the working directory. That's the actual current content of the folder on disk.

Then the index. The index is the graph of all the commits that we just saw. Creating a commit means writing to the index.

Slightly less obvious is the staging area. That's a sort of purgatory in which you can accumulate changes to write them to the index in a single commit.

Branches are quite simple. They are just pointers into the index. They point to certain commits.

Head is another pointer. Head points to the current active branch. That is the branch that will be updated if you make a commit. (unless you're on no branch, in which case you are headless)

The state of you repository is really much more than the state of the directory, it's the state of all of these five things together. And if you think about commands, you should think about them in terms of what they do to each of these five.

Commands – And what they do

- Git add
puts files from working director into staging area. If not in index, adds them to tracked files.
- Git commit
commits files from staging area to index, moves current branch with HEAD
- Git checkout [<commit>] [<file>]
Set <file> in working directory to state at <commit> *and stages it.*
- Git checkout [-b] <branch>
moves HEAD to <branch> (-b creates it), changes content of working dir
- Git reset --soft <commit>
moves HEAD to <commit> (takes the current branch with it)
- Git reset --mixed <commit>
moves HEAD to <commit>, changes index to be at <commit> (but not working directory)
- Git reset --hard <commit>
moved HEAD to <commit>, changes index and working tree to <commit>

Ok so now, let's got through some of the commands
and talk about what they do in terms of these five
concepts
[read slide]

Merge

- Fast-forward merge:

```
/tmp/git_graphs [git::master] [andy@dsi-amueller] [15:38]
> git merge another_one
Updating 513ced1..6cec4ed
Fast-forward
 D | 0
 E | 0
 2 files changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 D
 create mode 100644 E

* 6cec4ed (HEAD -> another_one) E
* 1e96a3b D
* 513ced1 (master) C
* b5ba00a B
* db928a0 A

* 6cec4ed (HEAD -> master, another_one) E
* 1e96a3b D
* 513ced1 C
* b5ba00a B
* db928a0 A
```

- Merge-commits:

```
/tmp/git_graphs [git::master] [andy@dsi-amueller] [15:43]
> git merge another_one
Merge made by the 'recursive' strategy.
 D | 0
 E | 0
 2 files changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 D
 create mode 100644 E

* 43563a5 (HEAD -> master) G
* c3ea8c8 F
* 6cec4ed (another_one) E
* 1e96a3b D
//
* 513ced1 C
* b5ba00a B
* db928a0 A

* ^ d6fedb0 (HEAD -> master) Merge branch 'another_one'
  / \
  * 6cec4ed (another_one) E
  * 1e96a3b D
  * | 43563a5 G
  * | c3ea8c8 F
  //
  * 513ced1 C
  * b5ba00a B
  * db928a0 A
```

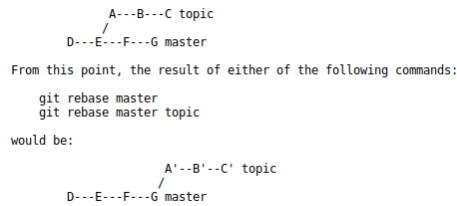
And then there's the more complex operations to the index, merging and rebasing. They change the index, and possibly the working tree. I find it most helpful to think of them as graph operations on the index.

There's two kinds of merging: feed-forward merges, and merges that require a commit. If one commit is a descendants of the other and you merge them, it will be a feed-forward, and just move the branch.

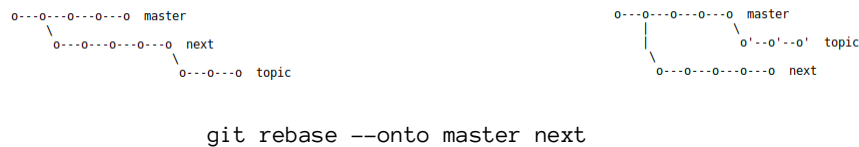
However, if one isn't a descendant of the other, git will create a "merge commit" that will unite the two, and have both as it's parent. Git will attempt its best but if there's conflicting changes, you might need to resolve the conflicts by hand.

Rebase

- Rebase



- Rebase onto:



Rebase is a whole different beast. It allows more or less arbitrary modifications of the graph, and therefore rewriting history.

You should be aware that if you use rebase, you will change a commits hash because the hash depends on the history.

Basically what rebase does is place a range of commits on top of another commit. If your on branch A and you do `git rebase B` (or any other commit), what will happen is that it will find the common ancestor, take everything up to that ancestor, and place it on branch B.

You can change the commit that it puts the changes on top of by using the `--onto` flag. So if I want to take the last five changes I do `git rebase HEAD~5 --onto B`. That will take the common ancestor with `HEAD~5`, which is `HEAD~5`, and place it onto B.

Squash before rebase!

Try to “squash” commits before doing a rebase – it might save you lots of conflict resolution!

Rebasing can also create conflicts, that need to be resolved the same way as merge conflicts. However, rebasing “plays back” all the commits that you moved on top of the target commit. That means that you might have to resolve conflicts on the same file multiple times – which you probably want to avoid.

A good way to get around that is to squash all the commits you want to rebase into a single one, and then rebase that single commit. That means you only have to do conflict resolution once.

One way to squash commits is using interactive rebase

Interactive rebase

```
git rebase -i <commit>
```

```
pick bddda7b [MRG + 2] [MAINT] Update to Sphinx-Gallery 0.1.7 (#7986)
pick 66443aa [MRG+1] Add prominent mention of Laplacian Eigenmaps (#8155)
pick 5d6460d MNT/BLD Use GitHub's merge refs to test PRs on CircleCI (#8211)
pick 08772c4 FIX Ensure coef is an ndarray when fitting LassoLars (#8160)
pick 4907029 [MRG+3] FIX Memory leak in MAE; Use safe_realloc; Acquire GIL only when rai
pick 4826883 Call sorted on lfw folder path contents (#7648)
pick aaebee1 FIX Issue #8173 - pass n_neighbors in MI computation (#8181)
pick be305ce TST/FIX Add check for estimator: parameters not modified by 'fit' (#7846)
pick 7978119 [MRG] #8218: in FAQ, link deep learning question to GPU question (#8220)
pick 9616acf CI remove obsolete comment

# Rebase aea6462..9616acf onto aea6462 (10 command(s))
#
# Commands:
# p, pick = use commit
# r, reword = use commit, but edit the commit message
# e, edit = use commit, but stop for amending
# s, squash = use commit, but meld into previous commit
# f, fixup = like "squash", but discard this commit's log message
# x, exec = run command (the rest of the line) using shell
# d, drop = remove commit
#
# These lines can be re-ordered; they are executed from top to bottom.
#
# If you remove a line here THAT COMMIT WILL BE LOST.
#
# However, if you remove everything, the rebase will be aborted.
#
# Note that empty commits are commented out
```

You can make any rebase interactive, but it's most common to rebase on an ancestor with interactive – a non-interactive rebase would have no consequences.

For each commit in the range that you want to rebase, interactive rebase allows you to pick the commit, which is leave it alone, squash it, which means incorporating it into the previous commit, removing it or amending it.

Interactive can be useful for cleaning up your history after you worked on a feature, so that the remaining commits are logical units.

Interactive adding

```
> git add -i
staged      unstaged path
1:   unchanged      +1/-0 setup.py

*** Commands ***
1: status      2: update      3: revert      4: add untracked
5: patch      6: diff        7: quit        8: help
What now> p
staged      unstaged path
1:   unchanged      +1/-0 setup.py
Patch update>> 1
staged      unstaged path
* 1:   unchanged      +1/-0 setup.py
Patch update>>
diff --git a/setup.py b/setup.py
index fb42749..8f9d283 100755
--- a/setup.py
+++ b/setup.py
@@ -34,6 +34,7 @@ MAINTAINER_EMAIL = 'amueller@ais.uni-bonn.de'
URL = 'http://scikit-learn.org'
LICENSE = 'new BSD'
DOWNLOAD_URL = 'http://sourceforge.net/projects/scikit-learn/files/'
+ADDED_JUST_FOR_YOU = "nothing"

# We can actually import a restricted version of sklearn that
# does not need the compiled code
Stage this hunk [y,n,q,a,d,/,,e,?]? y

*** Commands ***
1: status      2: update      3: revert      4: add untracked
5: patch      6: diff        7: quit        8: help
What now> █
```

What I find even more important is interactive adding.

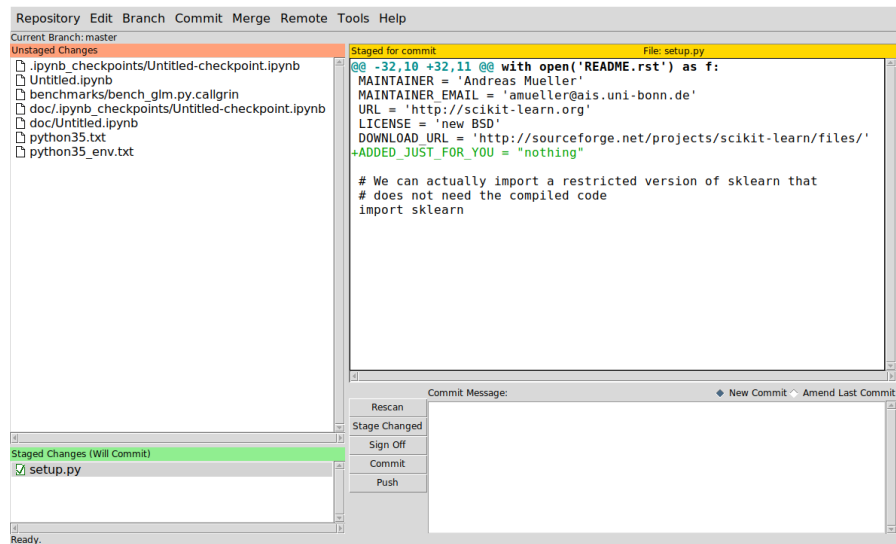
With git add, you usually add whole files to the staging area. I rarely ever do that. Usually I want to check line by line what changes I made and what changes I want to commit.

Git add -i allows me to through all the different hunks or lines I changed.

However, the command line interface is a bit clumsy for my taste. Or maybe I'm just not as used to it.

I prefer a gui, which you can summon with git gui

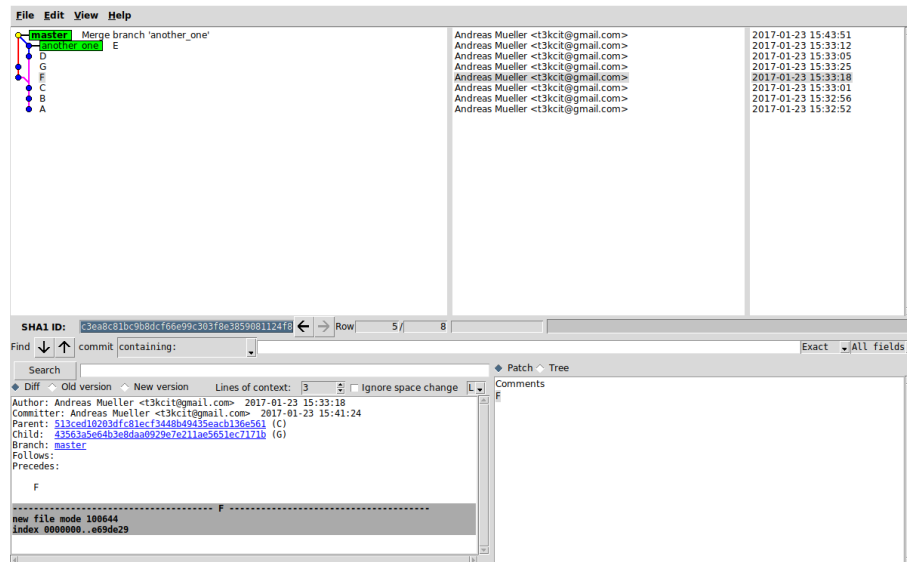
Git gui



Git gui is basically an interface for git add and git commit. You can also use it to push, but not much more.

This is the way I create most of my commits. You can see very easily which files have changed, see the changes, and stage single lines or whole files. Then you enter the commit message here, and click commit.

gitk



Another tool that I use all the time is gitk.

This is basically a graphical interface to git log, showing you the history of your project. But you can also use it as a graphical interface for reset, and for cherry-picking, which we won't go into.

Often I run gitk with `gitk --all`, which will show you all the branches. This is quite similar to the `Git log --oneline --annotate --graph --all` that I did earlier – but now in a gui!

This is usually how I do resets and how I look at the graph before I do any merges or rebases.

You can also use gitk to search commit messages, and there are many options on what to display and how. I mostly use the options to selectively show some branches that I'm interested in.

reflog

```
git reflog
```

```
d6fedb0 HEAD@{0}: merge another_one: Merge made by the 'recursive' strategy.
43563a5 HEAD@{1}: rebase finished: returning to refs/heads/master
43563a5 HEAD@{2}: rebase: G
c3ea8c8 HEAD@{3}: rebase: F
513ced1 HEAD@{4}: rebase: checkout 513ced1
4db426c HEAD@{5}: merge feature: Fast-forward
b5ba00a HEAD@{6}: checkout: moving from master to master
b5ba00a HEAD@{7}: reset: moving to HEAD~3
6cec4ed HEAD@{8}: merge another_one: Fast-forward
513ced1 HEAD@{9}: checkout: moving from another_one to master
6cec4ed HEAD@{10}: reset: moving to 6cec4ed
4db426c HEAD@{11}: checkout: moving from feature to another_one
4db426c HEAD@{12}: checkout: moving from master to feature
513ced1 HEAD@{13}: reset: moving to HEAD~4
4db426c HEAD@{14}: checkout: moving from feature to master
4db426c HEAD@{15}: checkout: moving from master to feature
4db426c HEAD@{16}: commit: G
125e957 HEAD@{17}: commit: F
6cec4ed HEAD@{18}: commit: E
1e96a3b HEAD@{19}: commit: D
513ced1 HEAD@{20}: commit: C
b5ba00a HEAD@{21}: commit: B
db928a0 HEAD@{22}: commit (initial): A
```

```
> git log --oneline --graph --decorate --all
* d6fedb0 (HEAD -> master) Merge branch 'another_one'
|
| * 6cec4ed (another_one) E
| * 1e96a3b D
| * 43563a5 G
| * c3ea8c8 F
|/
|
| * 513ced1 C
| * b5ba00a B
| * db928a0 A
```

The last command I want to mention is reflog.

Reflog is a very powerful tool that allows you to go through the history of your project, but not the same as log. Reflog actually tracks the changes to HEAD that you do with all the crazy commands that we talked about.

Log and gitk will only show you commits that are part of branches. If you do a rebase, for example, all the previous commits are still there, but not part of a branch any more. Remember, rebasing creates new hashes, so the rebased commits are different commits.

Imagine you want to go back to some states that are not part of any branch any more. Reflog allows you to do that. So if you break something during a rebase, or you lost a commit, you can always find it with reflog!

Git for ages 4 and up:
<https://www.youtube.com/watch?v=1ffBJ4sVUb4>
(with play-doh!)

For some of you, this was probably too slow, and for some of you, this was probably too fast.

Who here knew already all of this?

So for those of you who I managed to totally confuse, I recommend you go through these slides again, and also watch this video. It's pretty good, but it's also pretty long.

And for those of you who thought this was wayyy too much detail: this was still not all the important parts. There's also tags, and bare repositories and the stash...

Github – just another remote!

I want to just briefly talk about github.

For the purposes of git, github is just another remote.

And it's really nothing special. In terms of being a remote, you can replace github with a usb stick that you hand around.

There is a lot of tools for user management and issue tracking which is great, but not really central what we're talking about here. The main thing that's different in using github from using any other remote, is the use of pull requests, and the ability to integrate with remote services.

For now we'll only talk about pull requests.

GitHub pull request workflow

What are pull requests for?

Basically they allow you to contribute to a repository to which you don't have write permissions.

Let's say you want to contribute to scikit-learn.

There's the main repository, which we usually call "upstream", scikit-learn/scikit-learn. You want to change something there, but you don't have write permissions.

What you can do is "fork" it on github – that's a github concept, not a git concept. It's just a clone that also lives on github. You can then clone this fork locally, add a feature branch, make changes and push them. Then you can ask the owners of the repository if they want to merge your changes with a pull request.

GitHub pull request workflow

There is a bit of a weird asymmetry here, though. So obviously thing in the upstream repository change.

How do you get these changes onto your laptop?

You have to add upstream as an additional remote, and then you can pull from there.

You can create new features on top, and push the feature branches to your fork (which is usually the “origin” remote).

But what happens with the master branch on your fork? It never gets updated and there is no point in pushing to it, so it will just sit there and rot. I think that’s kind of weird, but that’s how github works.

So you always pull from upstream to get their changes, and push to your fork.

End version control – but github will come back
later ;)

Ok, so that's enough version control for now. But we'll
get back to githubs integration of third party services
later.

General coding guidelines

Next, I want to talk about Python and some software engineering principles. But before we do that, I want to mention two famous quotes that provide great general guidelines for software development.

Programs must be written for people to read, and
only incidentally for machines to execute.
- Harold Abelson (wizard book)

The first one is by Harold Abelson from the foreword of
“structure and interpretation of computer programs” a
classic in programming languages and compilers.

[read]

The gist is that the main point of code is to
communicate ideas to your peers and to your future
self.

A similar sentiment is expressed in the statement that
“code is read more often than it is written”. Take
more time writing code, so that you and other can
spend less time reading it. Don’t focus on what’s
“easy for the computer” or “elegant”. Focus on what’s
easy to understand for people.

Everyone knows that debugging is twice as hard
as writing a program in the first place. So if you're
as clever as you can be when you write it, how
will you ever debug it?
- Brian Kernighan

This one is from Brian Kernighan, who wrote the first book on C together with Dennis Richie. Kernighan is also the creator of awk and many other parts of unix, in particular the name. Anyhow...

[read]

This is another call for simplicity. Make code easy to understand. For yourself now to debug, for your future self to know what the hell you were thinking, and for others that might want to use this code in the future.

Hopefully all the code you write will be read again. Otherwise, what's the point. That might not be true for your assignments in most classes, but the point of the assignments is to practice for the real world. So I want to make sure that the code that you write in class is on the same standard that it needs to be out there on your job. Also, think of the poor course assistants.

Don't be clever!
Make it readable!

Future you is the most likely person to try to
understand your code.

Avoid writing code.

So to summarize:
[read]

Python basics

So now I want to go over some Python basics. And don't worry, we won't go over syntax or the standard library.

Why Python

- General purpose language
- Great libraries
- Easy to learn / use
- Contenders: R (Scala?)

First, a short defense on why I'm teaching this class in Python.

Python is a general purpose programming language, unlike matlab or R, so you can do anything with it.

It is very powerful, mostly because there are so many libraries for python that you can do basically anything with just a couple of lines.

Python is arguably easy to learn and easy to use, and it allows for a lot of interactivity.

The only real contender in the data science space that I can think of is R, which is also a good option, but well, I'm not an R guy, and you might have better chances with Python in industry jobs.

There's also Scala, but I'd argue that's way too complicated and doesn't have the right tools for the kind of data analysis and machine learning we want to do in this course.

The two language problem

Python is sloooooow...

- Numpy: C
- Scipy: C, fortran
- Pandas: Cython, Python
- Scikit-learn: Cython, Python

- CPython: C

So there's one thing that I really don't like about Python. Any idea what that is?

Python is sloooow. Like really slow.

So you know all these great libraries for Python for data science, like numpy, scipy, pandas and scikit-learn. Do you know what language they are written in?

Numpy is written in C,

Scipy is written in C and Fortran, pandas and scikit-learn are written in Cython and Python. And Cpython, the interpreter we use, is written in ... C obviously

That creates a bit of a divide between the users, who write python, and the developers, who write C and Cython. I have to admit, I don't write a lot of Cython myself, mostly Python... but that's not great.

So you need to be aware that if you actually want to implement new algorithms, and you can't express them with pandas and numpy, you might need to learn cython.

For this course, this won't really be a problem, though. We'll stay firmly on the Python side.

Python 2 vs Python 3

- “current” : 2.7, 3.4, 3.5, 3.6

Changes:

- Print
- Division
- Iterators (range, zip, map, filter, dictionary keys, values, items)
- Strings

There's another thing that you could call the two language problem, it's python 2 vs python 3.

The last version of python 2 is python 2.7, and really no-one should be using anything earlier. The commonly used versions of python 3 are 3.4 and 3.5, 3.6 was released on Christmas.

There is really no reason to use python 2 any more.

Unless you already wrote lots of code earlier. If you're at a company it might not be easy to make the transition, and that's why python 2 is still around.

So the important part are the changes. Anyone know what changed?

The print statement was removed, now print is a function, so it need parenthesis. The most common and trivial change. Division was changed to produce floats, so if you devided 2 by 3, it was 2 before, and now it's 2.5

Python 2 vs Python 3

- “current” : 2.7, 3.4, 3.5, 3.6

Changes:

- Print
- Division
- Iterators (range, zip, map, filter, dictionary keys, values, items)
- Strings

Then, many things that returned lists now return iterators, which is more memory efficient, but means you can't necessarily retrieve things by index any more. That's true for range, which now behaves like xrange before, zip, which behaves like izip, map and filter.

Also, dictionary keys, values and items are now iterables and not lists. If you tried to index any of those, you probably had a bug anyhow.

And really the main thing that changed is strings. And they changed completely. In 2, there was a string type and a unicode type. Now there's a string type, which is always unicode, and a bytes type that is the raw bytes and needs an encoding to interpret.

The story is really a bit complicated, and I suggest you read up on it.

Python 2 && Python 3

- `from __future__ import print_function`
- Six – tools for making 2 and 3 compatible code
- 2to3 – convert python2 code to python3 code

I suggest writing code that is compatible with both python 3 and python 2. For the first assignment I'll make you do that, for later assignments the code only needs to run in 3.4.

Generally it's easy for code to run on both. You can just always use parenthesis on print, it will have no effect in python2. Or you can import features, like float division from the future. `__future__` is a module that exists in python2 and python3 and that allows you to use certain python3 feature in python 2.

Some cases are not covered, though, and the six package helps out in these cases. It provides some common names between python2 and python3.

If you want to convert a codebase from python2 to python3, there's an automatic converter called 2to3, which can do that for you.

Python ...

Package management:

- Virtual environments
- pip (and wheels)
- Conda (and conda forge)

So one thing you should educate yourself a bit about when becoming a serious python user is package management. Unfortunately, it's a bit tricky, partly due to the two language problem, which means packages have dependencies that are not in python.

First of, you should be aware of the environment you are using. Usually it's a good idea to work with virtual environments, in which you can control what software is installed in what version. If you're on OS X or linux, your system will come with some python, but you don't really want to mess with that. Create your own environments, and are aware of which environment you are using at any moment.

The standard python way to install packages is pip, which is part of setuptools. Pip allows you to install all python packages in the pypi repository, which is basically all of them.

Python ...

Package management:

- Virtual environments
- pip (and wheels)
- Conda (and anaconda and conda forge)

Until not so long ago, pip needed to compile all C code locally on your machine, which was pretty slow. Now, there are binary distributions, called wheels, which mean no compilation any more! If you're compiling something when you're installing, you're probably doing it wrong.

The issue with pip is that it only works for python packages, and some of our packages rely on linear algebra packages like blas and lapack, and you need to install them some other way.

A really easy work-around for that is using a different package manager, called conda. It was created by a company called continuum IO and they ship a bundle called anaconda, which installs basically all the important packages. I recommend you use that for the course.

Conda can be used with different source repositories. By default, it uses the anaconda one that is managed by continuum IO, so that's managed by that company. There's also an open repository that is managed by the community that's called conda-forge.

The benefit of using commercial source is that it has intel's MKL library which is much faster than the open source alternative, OpenBlas.

In practice I use both conda and pip. But never try to upgrade a package you installed with one with the other! That's certain doom.

Pip and upgrades

Pip upgrade works on dependencies (unless you do `-no-dep`)

Oh and one word of warning: if you do `pip upgrade somepackage`, it will also update all the dependencies. That is often not what you want, in particular if you are using it in a conda environment or if you installed a particular version of numpy or scipy that you don't want upgraded.

Dynamically typed, interpreted

- Invalid syntax lying around
- Code is less self-documenting

One of the reasons Python is so easy to learn and use is because it's a dynamically typed languages.

So who of you have worked with statically typed languages like C, C++, Java or Scala?

It's often a bit cumbersome that you have to declare the type of everything, but it provides some safety nets. For example you know that if the code compiles, the syntax is correct everywhere. You don't know whether the code does what you want, but you know it'll do something.

Also, dynamically typed code is less self-documenting.

If I write a function without documentation, it's very hard for you to guess what I expect the input types to be. There's now type annotations for Python, which is great, but they are not supported in Python2.

So how can we get back our safety nets?

Editors

- Flake8 / pyflake
- Scripted / weak typing: Have a syntax checker!
- use autopep8 if you have code lying around

One of the simplest fixes is to have a syntax checker in your editor. Whatever editor you're using, make sure you have something like flake8 or pyflake installed that will tell you if you have obvious errors.

These will also tell you if you have unused imports, undeclared variables or unused variables. All that helps you to immediately fix problems, and not wait until you run your program.

I also recommend having a style checker. Flake8 also enforces pep8, which is the python style guide.

You should write pep8 compatible code. It will make it easier for others to read your code, and once you're used to it, it'll make it easier for you to read other's code.

If you want to convert code to be pep8 compatible, check out the autopep8 package.

Unit Tests and integration tests

So we guarded against some simple issues with our syntax checkers, but that doesn't find all errors, and it doesn't tell us if our code works. So we now we'll talk about unit testing and integration testing.

Who of you has worked with an automatic testing framework?

What is it for and why would we want it?

Why test?

- Ensure that code works correctly.
- Ensure that changes don't break anything.
- Ensure that bugs are not reintroduced.
- Ensure robustness to user errors.
- Ensure code is reachable.

So yes, we want to make sure that our code is correct.

We also want to make sure that if we rewrite something, it remains correct. If you have good tests, it is much easier to aggressively refactor your code.

If your tests are passing, everything works!

Another important kind of test is no-regression tests.

You found a bug, you fixed it, and now you want to make sure you don't re-introduce it. The easiest way is to write a test that tests for the bug.

You also want to make sure that your program behaves reasonable, even in edge-cases such as invalid input.

And finally, testing can help you find code that is actually unreachable by measuring coverage. If you can't write a test that will reach a particular part of the code, it's never executed, and you should probably think about that.

Test-driven development?

I love tests. But there are some people that love tests even more, and they practice what's called test-driven development. Who here has heard about that?

In test driven development, you write the tests before you write the code. And there are advantages to that. I don't usually do that myself.

However, I do test-driven debugging. If I find a bug, I first write a test that fails if the bug is present. I need to do that later anyhow, because I need a non-regression test, and it usually makes debugging much easier.

Types of tests

- Unit tests – function does the right thing.
- Integration tests – system / process does the right thing.
- Non-regression tests – bug got removed (and will not be reintroduced).

I usually think of tests in terms of three kinds: unit tests, that test the smallest possible unit, usually one function.

Then, there's integration tests, that test that the different parts of the software actually work together in the right way. That is often by testing several application scenarios.

And finally, there's non-regression tests, which can be either a unit tests, an integration test, or both, but they are added for a specific scenario that failed earlier, and you'll accumulate them as your project ages.

How to test?

- `py.test` – <http://doc.pytest.org>
- Searches for all `test_*.py` files, runs all `test_*` methods.
- Reports nice errors!
- Dig deeper:
<http://pybites.blogspot.com/2011/07/behind-scenes-of-pytests-new-assertion.html>

There are several frameworks to help you with unit testing in python. There's the built-in `unittest` module, there's the now somewhat abandoned `nose` tests. For the course we'll be using the `py.test` module, which is also what I'd recommend for any new projects. What it does is it searches all files starting with tests, runs them, and reports a summary. The tests should contain `assert` statements, and if any of them fail, you'll get an informative error. This is actually done with a considerable amount of magic, which you can read up on here, if you're into rewriting the AST.

Example

```
content of test_sample.py
def inc(x):
    return x + 2

def test_answer():
    assert inc(3) == 4

> py.test test_sample.py
===== test session starts =====
platform linux -- Python 3.5.2, pytest-2.9.2, py-1.4.31, pluggy-0.3.1
rootdir: /tmp, inifile:
plugins: cov-2.4.0, timeout-1.2.0
collected 1 items

test_sample.py F

===== FAILURES =====
test_answer
>
def test_answer():
    assert inc(3) == 4
E       assert 5 == 4
E       + where 5 = inc(3)

test_sample.py:7: AssertionError
===== 1 failed in 0.01 seconds =====
```

So here I have a slightly modified version of the example from their website. We have a function called `inc` that's supposed to increment a number. But there's a bug: it adds two instead of one.

And we have a test, which is called `test_answer`, that calls the `inc` function with the number three and checks if three is correctly incremented.

If we call `py.test` on this file, or on the folder of this file, `py.test` will run the `test_answer` function – because it starts with `test_`.

It tells us it ran one test, and that test failed. We also get a traceback showing us the line and what the actual value was. We got 5 instead of 4.

Example

```
''' content of test_sample.py
def inc(x):
    return x + 1

def test_answer():
    assert inc(3) == 4

'''

> py.test test_sample.py
===== test session starts =====
platform linux -- Python 3.5.2, pytest-2.9.2, py-1.4.31, pluggy-0.3.1
rootdir: /tmp, inifile:
plugins: cov-2.4.0, timeout-1.2.0
collected 1 items

test_sample.py .

===== 1 passed in 0.00 seconds =====
```

So now we go back and fix our increment function, and run `py.test` again. This time, it tells us the tests passed.

Does that mean the function is correct?

No, but we could test more. How? We could do the same with more numbers. However, tests are usually only necessary, not sufficient for correctness.

Usually all test for a project are in a separate file or even a separate folder. In this example we had the test and the function to be tested in the same file, but that's not good style.

The actual implementation should be completely separate from the tests.

Check coverage!

```
# inc.py
def inc(x):
    if x < 0:
        return 0
    return x + 1

def dec(x):
    return x - 1
```

```
test_inc.py
from inc import inc

def test_inc():
    assert inc(3) == 4
```

```
> py.test
===== test session starts =====
platform linux -- Python 3.5.2, pytest-2.9.2, py-1.4.31, pluggy-0.3.1
rootdir: /tmp/myproj, inifile:
plugins: cov-2.4.0, timeout-1.2.0
collected 1 items

test_inc.py .

===== 1 passed in 0.01 seconds =====
```

Here's a more complex example, with `inc.py` containing two functions and `test_inc.py` containing the same test. I added an `if` into the `inc` function, and a `dec` function.

If we run the test, they still pass. But clearly we're not testing everything. In this example, it's easy to see that we don't test the `if` branch and we don't test the `dec` function at all. If your project is larger and more complex, it's much harder to figure out whether you covered all the edge-cases, though.

That's where test coverage tools come in handy.

Check coverage!

```
# inc.py
def inc(x):
    if x < 0:
        return 0
    return x + 1

def dec(x):
    return x - 1

test_inc.py
from inc import inc

def test_inc():
    assert inc(3) == 4

/tmp/myproj [andy@dsi-amueller] [10:51]
> py.test --cov inc
===== test session starts =====
platform linux -- Python 3.5.2, pytest-2.9.2, py-1.4.31, pluggy-0.3.1
rootdir: /tmp/myproj, inifile:
plugins: cov-2.4.0, timeout-1.2.0
collected 1 items

test_inc.py .

----- coverage: platform linux, python 3.5.2-final-0 -----
Name      Stmts  Miss  Cover
-----
inc.py      6      2    67%

===== 1 passed in 0.01 seconds =====
```

So instead of just calling `py.test`, we specify `--cov inc`, which means we want to test the coverage of the `inc` module or file.

Now we get a coverage report that tells us that out of the 6 statements in `inc.py`, two were not covered, resulting in 67% coverage.

There are several ways to figure out which lines we missed, I like the html report the most.

HTML report

```
> py.test --cov inc --cov-report=html
```

Coverage report: 67%

Module ↓	statements	missing	excluded	coverage
inc.py	6	2	0	67%
Total	6	2	0	67%

coverage.py v4.2, created at 2017-01-23 10:53

Coverage for **inc.py** : 67%

6 statements 4 run 2 missing 0 excluded

```
1 # inc.py
2 def inc(x):
3     if x < 0:
4         return 0
5     return x + 1
6
7
8 def dec(x):
9     return x - 1
```

So we specify `--cov-report=html`, and `py.test` will create a html report for us. We get an overview, that contains the same information as before, but we'll also get a detailed view of `inc.py`, which shows us which lines we covered and which lines we missed.

And now we can clearly see that we never reached the `return 0` line in `inc` and we never tested `dec`.

Whenever you write tests, you should make sure they cover all cases in your code, in particular the different algorithmic pieces. Covering all error messages is possibly not as important. You should usually aim for coverage in the high nineties.

If you look at projects on github, you can often see a badge that says “coverage X %”, showing the code coverage. These badges are actually automatically generated, and I next I want to talk about how that's done.

Continuous integration (with GitHub)

This is the magic of continuous integration.
Who has heard of continuous integration before?
Continuous integration is a general paradigm in software engineering, of automatically running integration tests whenever you change your software.
I want to talk about it in particular in the context of github, because that's what you'll likely be using, both here and later in industry – at least in a startup. If you go to google or facebook or amazon, they all have their own frameworks, but the same principles apply.

What is Continuous integration?

- Run command on each commit (or each PR).
- Unit testing and integration testing.
- Can act as a build-farm (for binaries or documentation).
- requires clear declaration of dependencies.
- Build matrix: Can run on many environments.
- Standard serviced: TravisCI, Jenkins and CircleCI

Ok so what's continuous integration in more detail?

It runs some sequence of commands on a cloud machine every time a commit is made, either just for a particular branch, or for each pull request.

Usually the command is just running the test suit, but you can also check coverage or style or build binaries or rebuild your documentation.

This is done on a clean cloud machine, so you need to be very explicit about your dependencies, which is good. You can specify a build matrix of different systems you want to run, such as linux and os X, different versions of Python and different versions of the dependencies, like older and newer numpy versions.

There are some standard services out there that are free for open source and educational purposes, travis, jenkins and circle are some of them. We'll use travis for this course.

Benefits of CI

- Can run on many systems
- Can't forget to run it
- Contributor doesn't need to know details
- Can enforce style
- Can provide immediate feedback
- Protects the master branch (if run on PR)

So what's the benefits of doing this?

You can run it on many systems, and in parallel. You might not have all the operating systems that you want it to run on, and you might not go through the hassle of trying out every patch on every machine.

Because CI is automatic, you can't forget to run it. There's github integration that will give you a red x or a green check mark, and everybody will know whether your commit was ok or not.

You can make a change to a package without knowing all the requirements and even without knowing how to run the tests. The CI will complain if you broke something.

And you get immediate feedback while you're working, because tests are run every time you commit!

Most projects will only merge pull requests that pass CI, and that means the master branch can not break and will always be in working condition, which is important.

What does it do?

- Triggered at each commit / push
- Sets up a virtual machine with your configuration.
- Pulls the current branch.
- Runs command. Usually: install, then test.
- Reports success / Failure to github.

Ok so let's go through the steps that the CI performs. Let's say you configured travis for one of your branches on github. If you push to that branch, travis will be triggered.

A virtual machine will be set up with the configuration that you specified. The machine pulls your current version of the project. If it's a pull request, it might also use the code that would be the result of a merge, so it would be your changes applied to the current project.

Then some commands are run that you can specify. Usually these are installing the package, possibly including dependencies, and then running the test suite.

After the test-suite runs, it will report back to github.

Setting up TravisCI

- Create account linked to your github account:

<https://travis-ci.org/>



AppliedMachineLearning/Homework-I-starter

- Check out <https://docs.travis-ci.com/>

```
└─ .travis.yml
language: python
python:
  - "2.7"
  - "3.4"
  - "3.5"
# command to install dependencies
install: "pip install -r requirements.txt"
# command to run tests
script: pytest
~
```

So in general to set up travis you have to create an account that is linked to your github account, and enable travis-ci on the repository you want.

For public repositories and students its free, otherwise you have to pay for the service.

For the homework, I already set this up for the github classroom organization.

There a docs online at this website, and the only thing you need to do is create a .travis.yml file with the configuration.

Here, we specify the versions of python we want to run, the install command, and the final command. This will run pip install with a requirements file. You could also run python setup.py install, or specify requirements here directly.

Using Travis

- Triggered any time you push a change
- Integrated with Pull requests
- Try a pull request on your own repository!

Travis is run automatically each time you push a change, so in the homework, push a change and then see the status page.

You can also do a pull request on your own repository, so you can see the pull request integration. It's pretty cool and you should try it out.

Documentation

So now, we come to the last part of today's lecture, documentation. We talked earlier about how important it is to write readable code.

But no code is perfectly documenting itself. You definitely need to write additional documentation.

Also, many people don't like digging through code, so some documentation that's more easily accessible is helpful.

Why document?

- Your code is harder to understand than you think.
- Input types and output types are unclear in dynamic languages.
- Often implicit assumptions about input.

So why do we document?

To make reading the code easier. Others will thank you, including your future self.

While you should document any function or class in any language, in dynamically typed languages in particular, it's often unclear what the assumptions about the input are, and what the type of the output is. So these are really crucial to document.

And if the input is some complex object like a dictionary or a custom class, often you make assumptions about the content of this object that are not obvious from the code.

Python documentation standards

- PEP 257 for docstrings for class, methods and functions

```
def inc(x):  
    """Add one to a number.  
  
    This function takes as argument a number,  
    and adds one to it.  
    The result is returned.  
    """  
    if x < 0:  
        return 0  
    return x + 1
```

- Additional inline documentation

```
if x < 0:  
    # x is less than zero  
    return 0  
return x + 1
```

Python has some standards for documentation, described in pep 257.

Every class, method or function should have a docstring, at the very least all public ones.

These are particularly helpful because they can be easily viewed inside a python session or with Jupyter Notebook.

For docstrings we use triple quotes, with a single line Explanation in the first line, then an empty line, then a more detailed explanation.

This docstring is stored in the `__doc__` attribute of the object.

In addition to that, you can use inline documentation when you think it might be useful, using the pound.

NumpyDoc format

- See https://github.com/numpy/numpy/blob/master/doc/HOWTO_DOCUMENT.rst.txt

```
Parameters
-----
x : type
    Description of parameter `x`.
```

You might have noticed that the python folks like standards. In the data science community there's a stricter standard, which is the numpydoc format.

It's what you'll see in numpy, scipy, pandas and scikit-learn for example.

The documentation has several sections for Parameters, return values, examples, notes and more.

The most important parts are the parameters and return values, which you should document as you can see here.

A detailed description of the documentation format is in the doc I linked to. You should always document all parameters and all return values.

This documentation is in restructured text format, which we'll come to in a bit.

This format is a bit idiosyncratic with a space before the colon, but it's important that you actually stick to this.

Examples

```
class MultinomialNB(BaseDiscreteNB):
    """Naive Bayes classifier for multinomial models

    The multinomial Naive Bayes classifier is suitable for classification with
    discrete features (e.g., word counts for text classification). The
    multinomial distribution normally requires integer feature counts. However,
    in practice, fractional counts such as tf-idf may also work.

    Read more in the :ref:`User Guide <multinomial_naive_bayes>`.

    Parameters
    -----
    alpha : float, optional (default=1.0)
        Additive (Laplace/Lidstone) smoothing parameter
        (0 for no smoothing).

    fit_prior : boolean, optional (default=True)
        Whether to learn class prior probabilities or not.
        If false, a uniform prior will be used.

    class_prior : array-like, size (n_classes), optional (default=None)
        Prior probabilities of the classes. If specified the priors are not
        adjusted according to the data.

    Attributes
    -----
    class_log_prior_ : array, shape (n_classes,)
        Smoothed empirical log probability for each class.

    intercept_ : property
        Returns "class_log_prior_" for interpreting MultinomialNB
        as a Linear model.

    feature_log_prob_ : array, shape (n_classes, n_features)
        Empirical log probability of features
        given a class. "P(x_i|y)".

    coef_ : property
        Returns "feature_log_prob_" for interpreting MultinomialNB
        as a Linear model.

    class_count_ : array, shape (n_classes,)
        Number of samples encountered for each class during fitting. This
        value is weighted by the sample weight when provided.

    feature_count_ : array, shape (n_classes, n_features)
        Number of samples encountered for each (class, feature)
        during fitting. This value is weighted by the sample weight when
        provided.

    Examples
    -----
    >>> import numpy as np
    >>> X = np.random.randint(5, size=(6, 100))
    >>> y = np.array([1, 2, 3, 4, 5, 6])
    >>> from sklearn.naive_bayes import MultinomialNB
    >>> clf = MultinomialNB()
    >>> clf.fit(X, y)
    >>> MultinomialNB(alpha=1.0, class_prior=None, fit_prior=True)
    >>> print(clf.predict(X[2:3]))
    [3]

    Notes
    -----
    For the rationale behind the names 'coef_' and 'intercept_', i.e.
    naive Bayes as a linear classifier, see J. Rennie et al. (2003):
    Tackling the poor assumptions of naive Bayes text classifiers. IJML.

    References
    -----
    C.D. Manning, P. Raghavan and H. Schuetze (2008). Introduction to
    Information Retrieval. Cambridge University Press, pp. 234-265.
    http://nlp.stanford.edu/IR-book/html/htmledition/naive-bayes-text-classification-1.html
    """

    def fit(self, X, y, sample_weight=None):
        """Fit Naive Bayes classifier according to X, y

        Parameters
        -----
        X : {array-like, sparse matrix}, shape = [n_samples, n_features]
            Training vectors, where n_samples is the number of samples and
            n_features is the number of features.

        y : array-like, shape = [n_samples]
            Target values.

        sample_weight : array-like, shape = [n_samples], optional (default=None)
            Weights applied to individual samples (1. for unweighted).

        Returns
        -----
        self : object
            Returns self.
        """
```

Here you can see two examples from scikit-learn. On the right is the docstring for the `DummyClassifier` class. The class docstring follows the class definition, and the parameters are the parameters to the `init` function.

You can see parameters, attributes, examples, notes and references.

Next to it is the `fit` method, which just has a one-line description and then the parameters and return values.

You can see that the type descriptions can get pretty specific, giving the matrix shapes.

You might ask why we're using this specific format and why we're using restructured text. And the answer to this is sphinx.

Sphinx and reStructured Text

```
.. _svm:

=====
Support Vector Machines
=====

.. currentmodule:: sklearn.svm

**Support vector machines (SVMs)** are a set of supervised learning
methods used for :ref:`classification <svm_classification>`,
:ref:`regression <svm_regression>` and :ref:`outliers detection
<svm_outlier_detection>`.

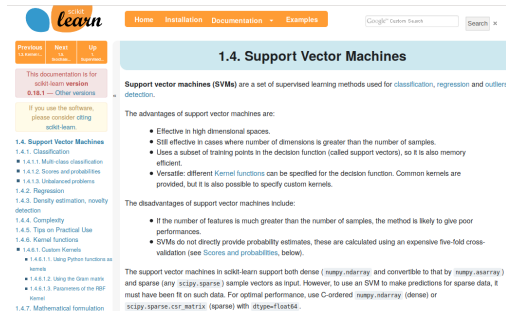
The advantages of support vector machines are:

- Effective in high dimensional spaces.
- Still effective in cases where number of dimensions is greater
  than the number of samples.
- Uses a subset of training points in the decision function (called
  support vectors), so it is also memory efficient.
- Versatile: different :ref:`svm kernels` can be
  specified for the decision function. Common kernels are
  provided, but it is also possible to specify custom kernels.

The disadvantages of support vector machines include:

- If the number of features is much greater than the number of
  samples, the method is likely to give poor performances.
- SVMs do not directly provide probability estimates, these are
  calculated using an expensive five-fold cross-validation
  (see :ref:`Scores and probabilities <scores_probabilities>`, below).

The support vector machines in scikit-learn support both dense
(`numpy.ndarray` and convertible to that by `numpy.asarray`) and
sparse (any `scipy.sparse` sample vectors as input. However, to use
an SVM to make predictions for sparse data, it must have been fit on such
data. For optimal performance, use C-ordered `numpy.ndarray` (dense) or
`scipy.sparse.csr_matrix` (sparse) with `dtype=float64`.
```



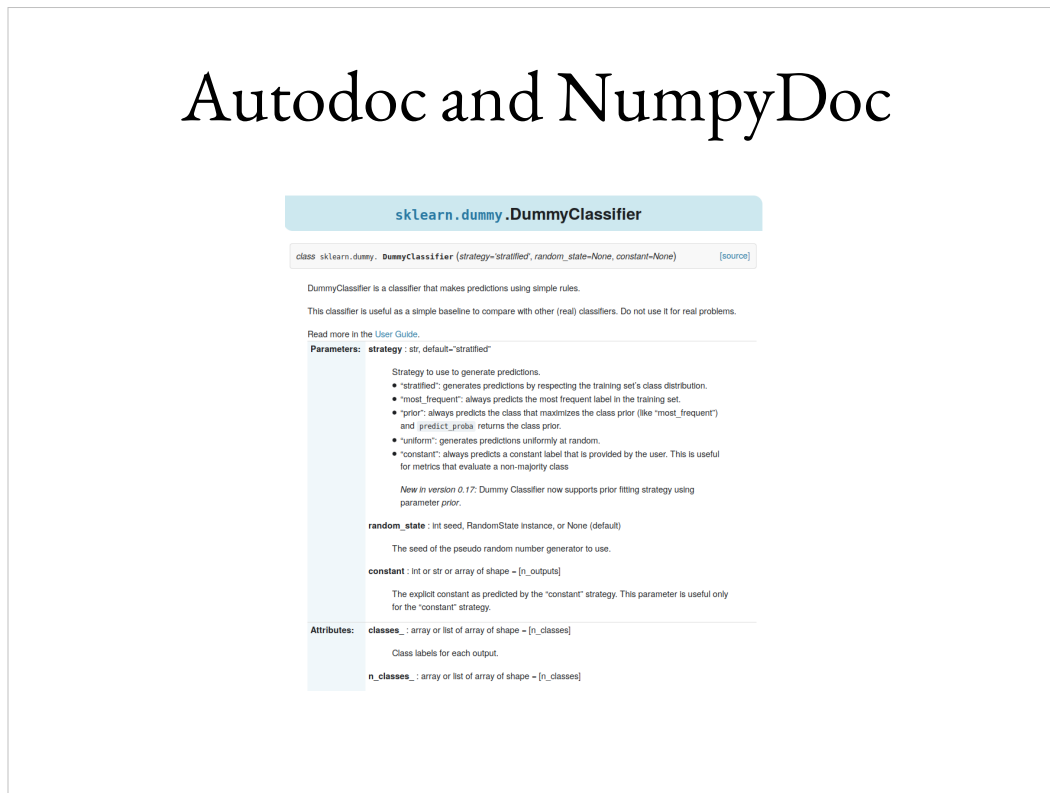
So what's this sphinx? It's basically a static page generator that takes restructured text and produces HTML and pdf output.

This is how the scikit-learn documentation is generated for example. You can see the rst code on the left and the generated website on the right. Obviously there's also a theme file that governs the layout.

RST is similar to markdown but it also allows internal references, figures and simple ways to extend the format with custom patterns.

Sphinx can also automatically generate html and pdf documentation from the docstrings itself, using a module called autodoc. That's how the API documentation is generated, and that's where numpydoc comes in.

Autodoc and NumpyDoc



Here is the documentation of the DummyClassifier that we saw earlier as rendered by Sphinx using autodoc. You can see the different sections here.

There's more sections, and also documentation for the methods down below.

Sphinx also adds in links to the source code on github if you like.

So now obviously you want to show this to people interested in your project, and you don't want them to install sphinx and build the docs before they can read them.

One option would be to build this locally and then use github pages to host the site.

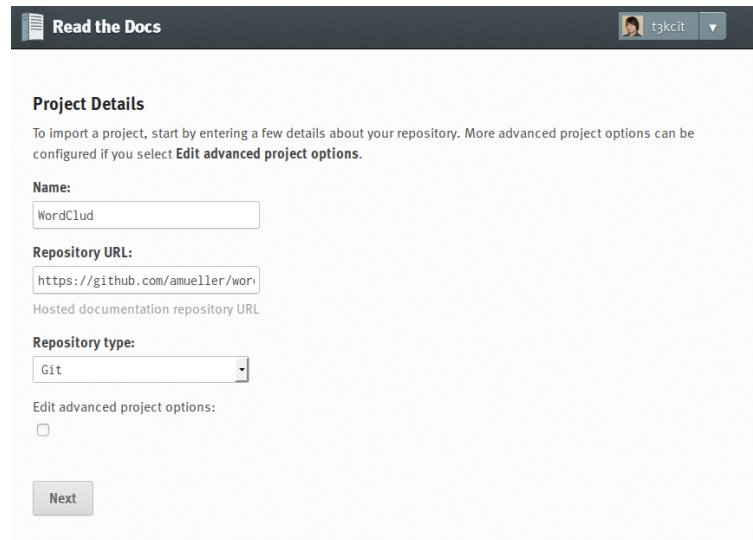
That's fine, but also a little cumbersome.

Setting up Sphinx

- Install sphinx
- Run sphinx-autogen
- Edit conf.py – pick a theme like https://github.com/snide/sphinx_rtd_theme
- Edit your index.rst



Setting up ReadTheDocs



The screenshot shows the 'Project Details' form on the ReadTheDocs website. At the top, there's a dark header with the 'Read the Docs' logo and a user profile 't3kcit'. The main content area is light gray and contains the following fields:

- Project Details**
To import a project, start by entering a few details about your repository. More advanced project options can be configured if you select **Edit advanced project options**.
- Name:**
A text input field containing 'WordClud'.
- Repository URL:**
A text input field containing 'https://github.com/amueller/word-clud'.
- Repository type:**
A dropdown menu with 'Git' selected.
- Edit advanced project options:**
A checkbox that is currently unchecked.
- Next**
A button at the bottom of the form.

We could also use another cloud service, called ReadTheDocs.

That's basically a service for building and hosting sphinx documentation.

You just have to specify your repository, and it will automatically build and host the documentation for your.

Pretty easy, right?