

Datorlaboration:
Introduktion till versionshantering med git och github
(för dataanalytiker och statistiker)

Måns Magnusson

11 februari 2016

Innehåll

1 Förutsättningar	2
1.1 Mjukvara	2
1.2 github account	2
2 Grundläggande git	2
2.1 Skapa ett repository	2
2.2 Lägga till filer i repot (Add)	3
2.3 Stage och Commit	6
2.4 Reset unstaged changes	11
2.5 Diff	12
2.6 Tags	13
2.7 .gitignore	14
3 Remote repositories	15
3.1 Skapa ett globalt (remote) repository	15
3.2 Push	17
3.3 Pull	18
3.4 Conflicts	20
4 Branch och merge	23
4.1 Branch	23
4.2 Merge	26
5 Korrigera fel i repot - Revert och Reset	28

1 Förutsättningar

1.1 Mjukvara

För att kunna genomföra denna laboration (och använda git framöver) behövs versionshanteringsverktyget git. För att underlätta arbetet med git rekommenderas också det grafiska gränsnittet SourceTree. SourceTree fungerar oavsett operativsystem eller program, således fungerar det lika bra att versionshanter SAS eller SPSS-kod som att versionshanter R-kod eller rapporter. Programmen är gratis och bygger på öppen källkod.

1. git: Det program som används för att versionshanter filer. Git går också att använda direkt från terminalen/kommandotolken för mer avancerade användare. Ladda ned och installera härifrån:
<https://git-scm.com/>
2. SourceTree: Grafiskt gränsnitt till git som gör det bekvämare och enklare att arbeta med git. Ladda ned och installera härifrån:
<https://www.sourcetreeapp.com/>
3. R-Studio: För de som arbetar med R i R-Studio går det också att använda R-Studio som grafiskt gränssnitt mot git. Dock klarar inte R-Studio (i dagsläget) att hantera "branch" och "merge" fullt ut. Ladda ned och installera härifrån (kräver R):
<https://www.rstudio.com/>

1.2 github account

Utöver mjukvaran ovan behövs också ett användarnamn på github.com för att arbeta med git remote repositorys. Skapa gratis ett konto på github.com.

2 Grundläggande git

Git är ett system för att versionshanter filer och förenkla samarbete när det gäller kod och filer i projekt. Det är gjort för att vara snabbt, distribuerat och ta minimalt med utrymme. För att versionshanter ett projekt beöver vi först skapa ett repository i git.

2.1 Skapa ett repository

Det mest grundläggande enheten i git är ett "repository" eller kortare "repo". Det är en samling med filer som ingår i ett projekt, som kodfiler, data och rapporttexter. Exakt vilken omfattning ett repo ska ha är inte helt klart, men ofta utgörs ett repo av ett projekt, en samling kod (exempelvis i form av ett R paket) eller ett dataset med tillhörande kod.

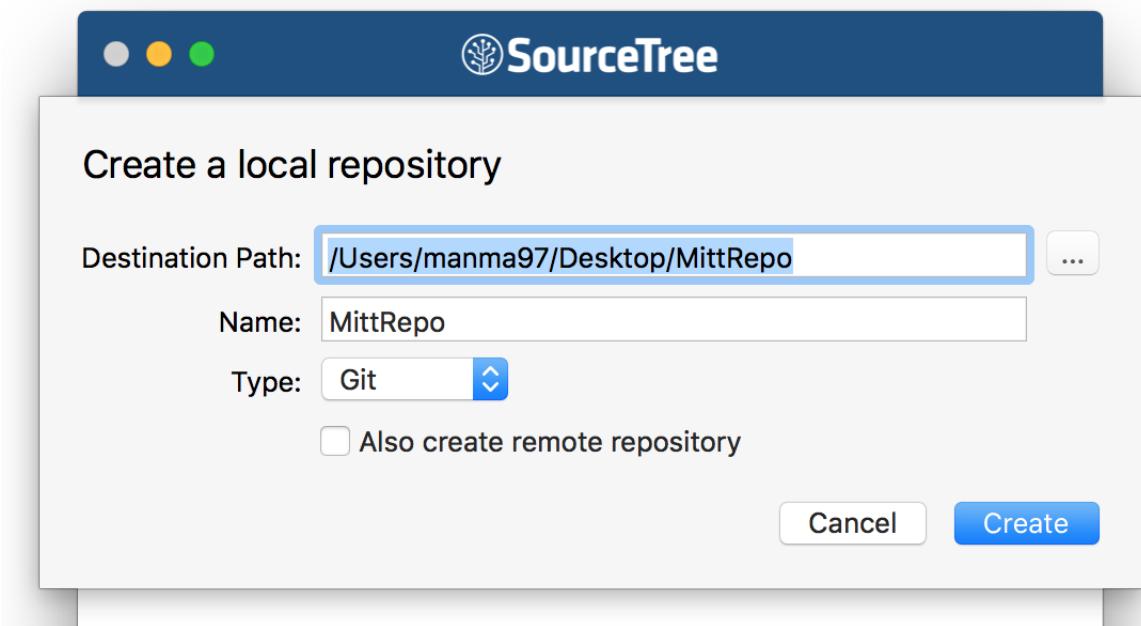
För att skapa ett nytt repo:

1. Starta SourceTree
2. Klicka på + New repository ⇒ Create local repository



Vi skapar nu ett lokalt repo, d.v.s. det kommer just nu bara finnas på din egen dator (vi kommer gå in på s.k. remote repositorys senare).

3. Skriv in var du vill ha ditt lokala repo. Namnet på ditt repo kommer vara den mapp du skapar repot i. Nedan är ett exempel på att jag skapar ett nytt repo som ligger på skrivbordet i mappen MittRepo och som därför också heter MittRepo.



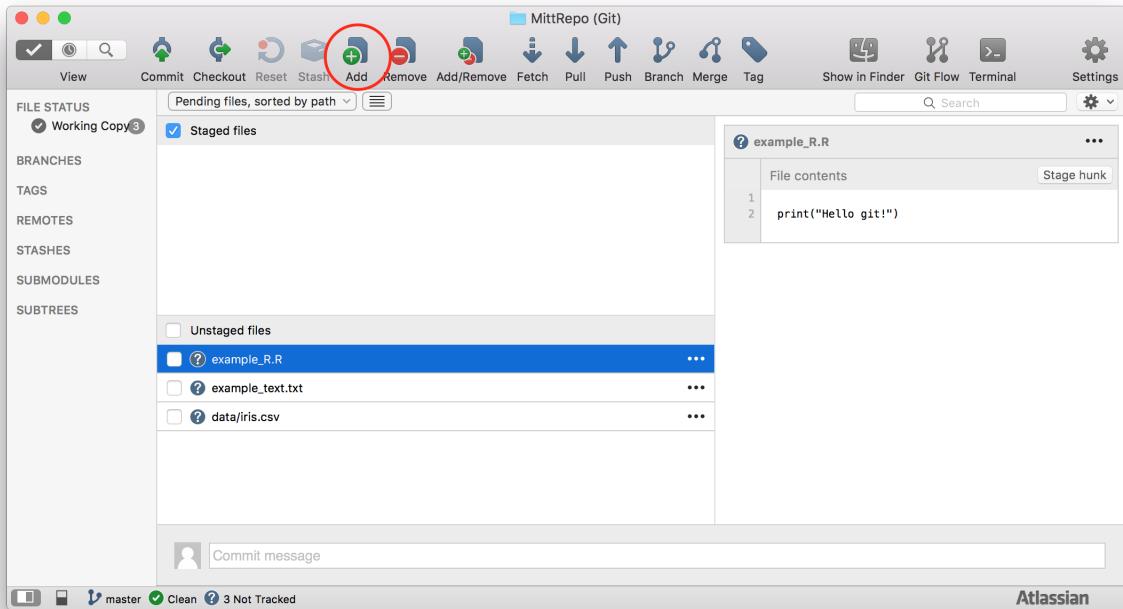
4. Nu har du skapat ditt första egna repo som dyker upp i SourceTrees förstasida. För att börja arbeta med filerna i repot, dubbelklicka på det nya repot.



2.2 Lägga till filer i repot (Add)

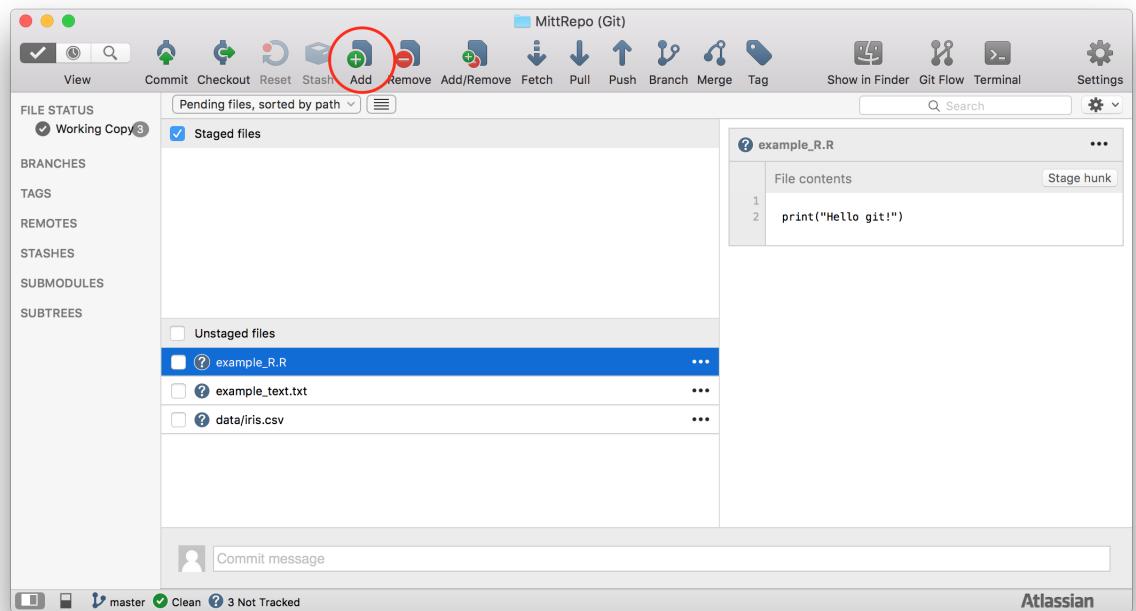
Nu har vi skapat ett helt nytt repo, lokalt på din dator. Nu är det dags att lägga till de filer vi vill versionshantera. Det finns tre stycken exempelfiler som vi kommer använda i följande exempel. Dessa filer finns att ladda ned från github [[här](#)].

Dessa filer kopierar jag till den mapp som utgör mitt nyligen skapade repo. Filen `iris.csv` ligger i en undermappen `data`. I SourceTree borde det då se ut ungefär såhär:



Vi ska nu lägga till dessa filer i vårt repo och börja versionhantera dem.

1. Vi börjar med att markera vilka filer vi vill lägga till i repot. Markera de filer du vill lägga till och klicka på **Add**.

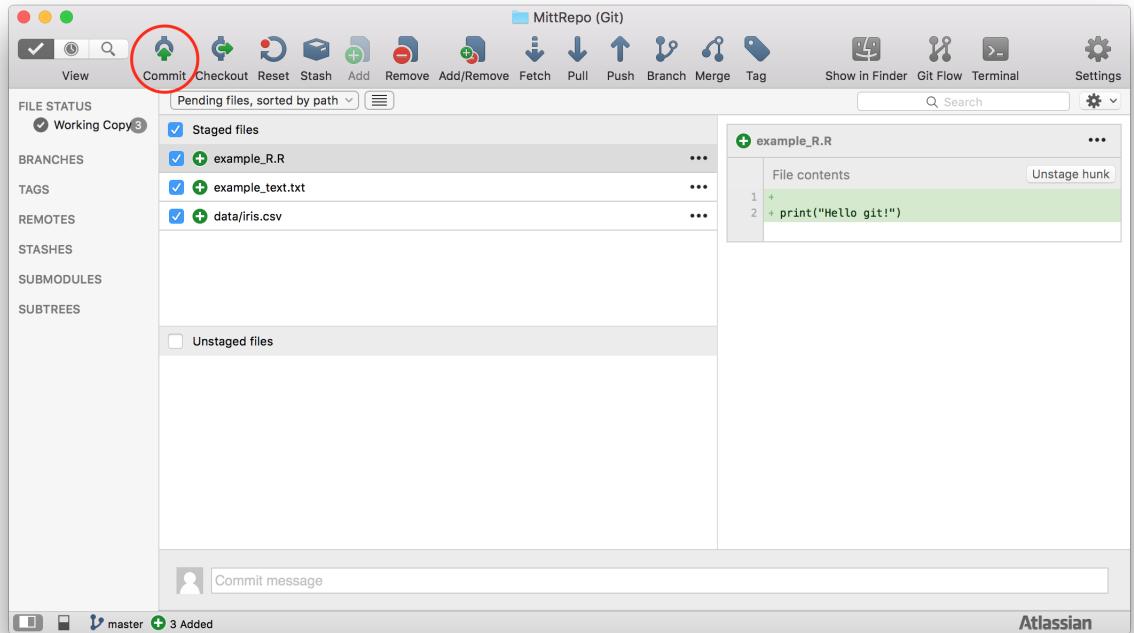


Detta innebär att vi har markerat de filer vi vill lägga till i vårt repo. Detta kallas att vi "Stage" de filer vi vill lägga till. I denna labb kommer jag kalla detta för att markera filer för att lägga till i repot.

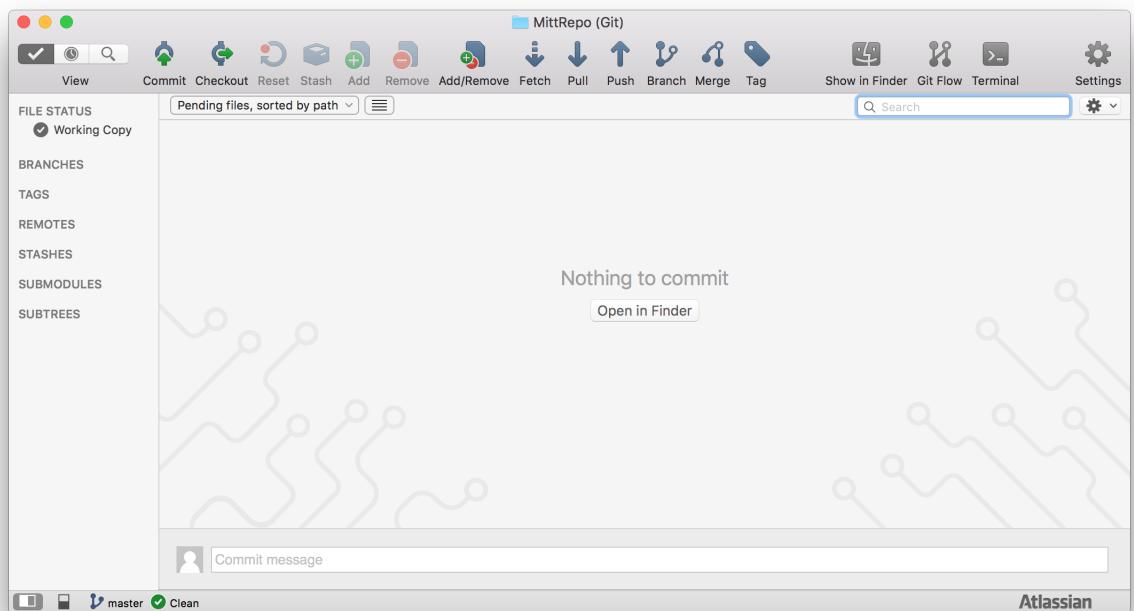
2. Nästa steg är att vi ska lägga till de filer vi markerat. Detta gör vi med genom att göra en så kallad **Commit**. En översättning av commit är fästa, vilket beskriver vad vi gör. Vi fäster dessa filer (och

framöver våra förändringar) i vårt repo.

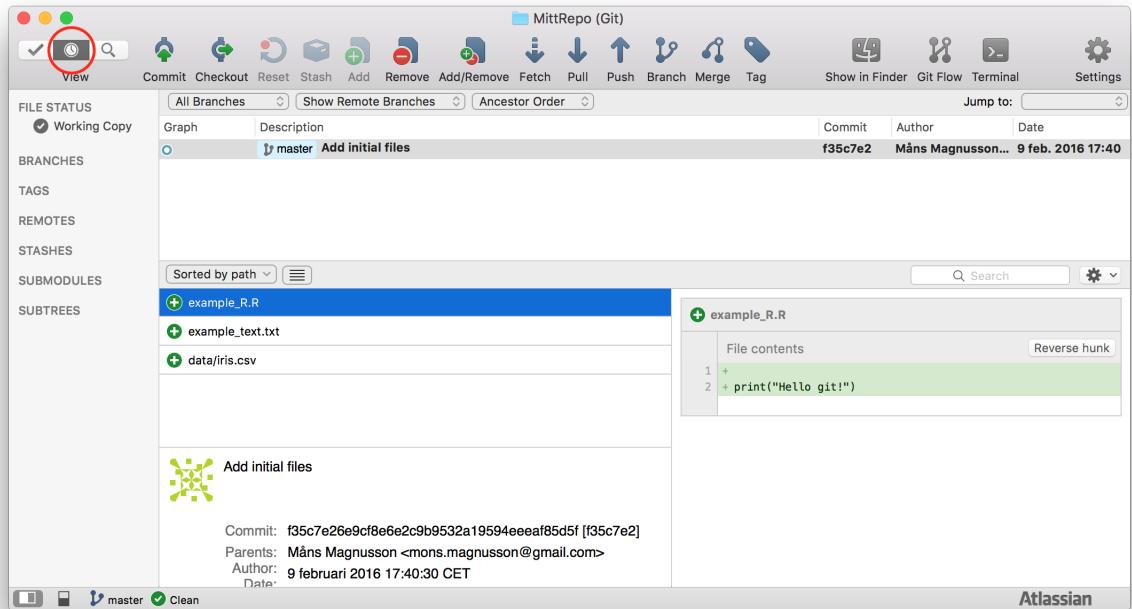
När vi klickar på **Commit** får då upp att vi ska ange ett ”commit message”. Detta är ett meddelande vi kommer gå in på mer i detalj senare. Nu anger vi ”Add initial files” som meddelande. Klicka sedan på knappen **Commit**.



3. Nu har vi skapat vårt första repo och lagt till de tre filer vi vill versionhantera. I SourceTree ser det nu ut som filerna saknas, det beror på att det inte finns några förändringar i filerna i vårt repo. Gå in i den mapp du versionshanterat (MittRepo) och titta. Filerna finns kvar.



- Vi kan vilja se vilka förändringar vi gjort (för spårbarheten). För att göra det klickar vi på ”klockan” i vänstra övre hörnet. Då får vi fram historiken över vilka förändringar (commits) vi gjort i våra filer. I denna vy kan vi se vilka tidigare commits som gjorts, när de gjordes och av vem de gjordes. Nu är det bara en commit gjord, men längre fram kommer vi se hela historiken med alla våra förändringar vi gjort.



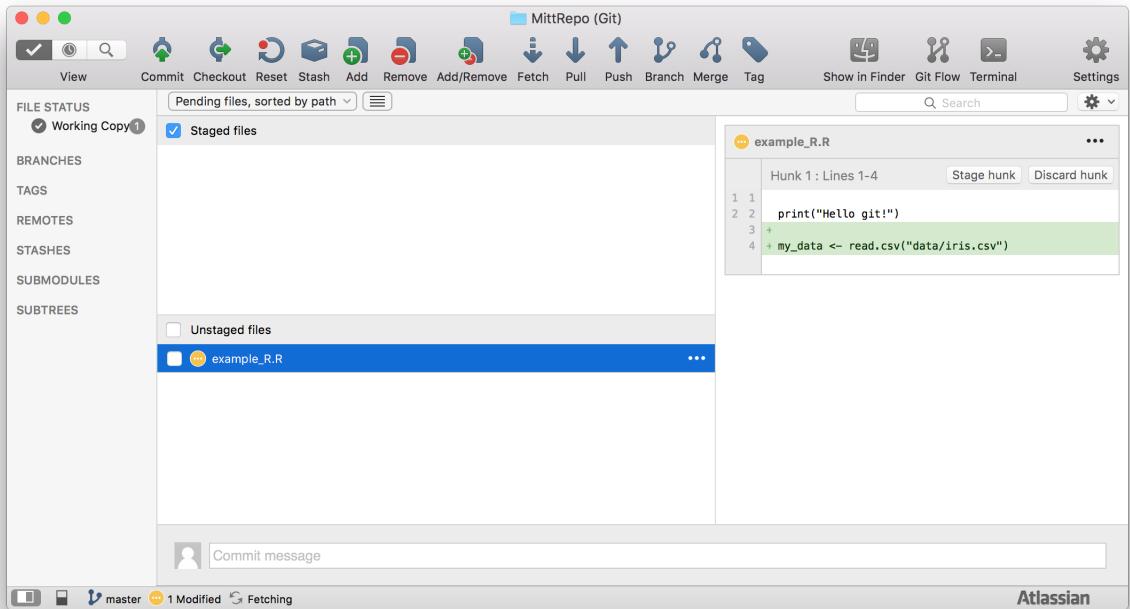
2.3 Stage och Commit

Versionshantering med git handlar om att vi gör förändringar i våra filer och vi sedan vill lägga till dessa förändringar. Som ett första steg ska vi ändra i en av våra versionshanterade filer. Vi ska lägga till en rad i example.R.R där vi läser in csv-filen iris.csv.

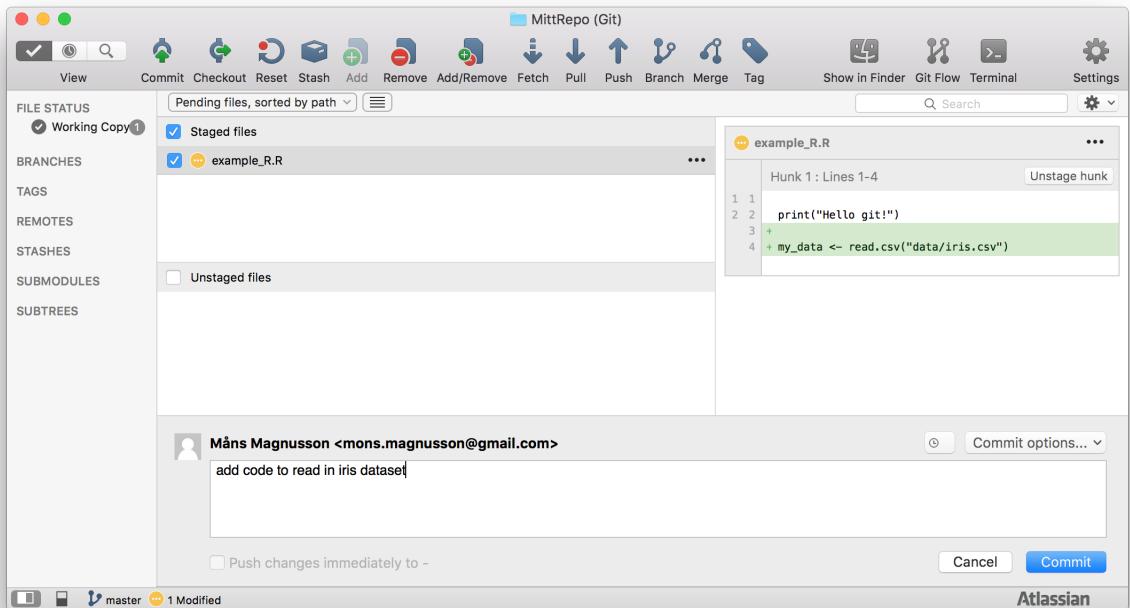
- Öppna filen `example.R.R` (om du inte har R kan du göra det med en vanlig texteditor) och lägg till följande rad kod:

```
my_data <- read.csv("data/iris.csv")
```

Spara sedan filen. Då borde det se ut på följande sätt i SourceTree:



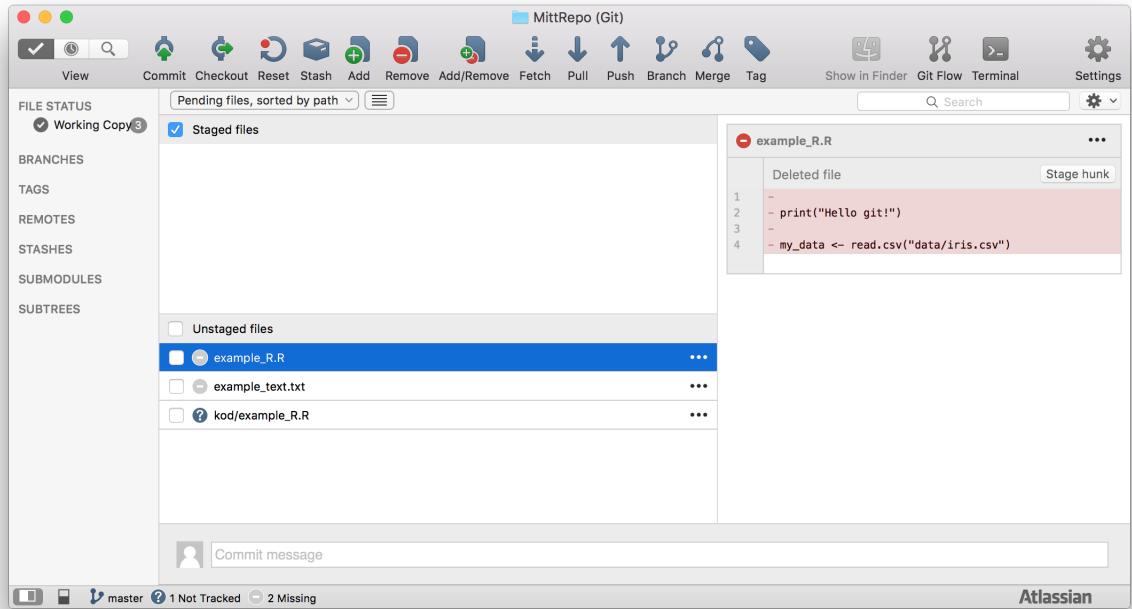
2. Filen har nu dykt upp i git och vi ser i grönt exakt vilket tillägg som gjorts i filen. Detta beror på att git jämför filerna med den senaste commiten och ser nu en skillnad.
3. Precis som tidigare ska vi nu lägga till denna förändring, först markerar vi denna förändring med **Add**. Då kommer vi flytta filen från "unstaged files" (ej markerade filer) till "staged files" (markerade filer). Det innebär att vi markerar att vi vill lägga till denna förändring i vårt repo.
4. Nästa steg är att lägga till denna förändring i vårt repo genom att commita de förändringar vi gjort. Klicka på **Commit** och fyll i ett commit message.



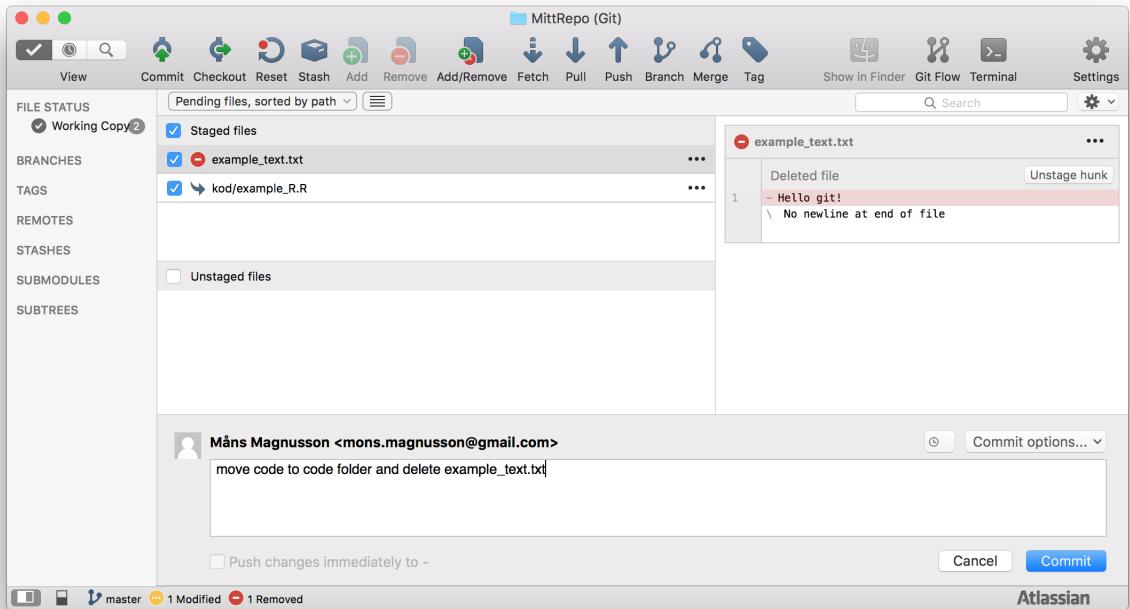
5. Nu har vi gjort en förändring och commitat in denna förändring i vårt repo.

På detta sätt kan vi hantera förändringar av filer. Vi kan självklart också lägga till nya filer (som tidigare), flytta filer och ta bort filer. Nu ska vi se hur det fungerar.

1. Vi ska nu prova att flytta vår fil `example.R.R` till en mapp ”kod” i vårt repo och ta bort vår fil `example_text.txt` fil. När detta är gjort borde det då se ut på följande sätt i SourceTree:



2. Precis som tidigare måste vi markera vilka förändringar vi vill ”stage to commit”. Markera alla tre förändringar som git har identifierat. När vi markerar dessa förändringar förstår git att vi genom att ta bort en fil (`example.R.R`) och skapa samma fil igen, bara flyttar filen mellan två olika mappar. När vi har markerat dessa förändringar kan vi också commita dessa två förändringar till vårt repo med **Commit**.



Det är inte alltid vi vill lägga till alla förändringar i en enskild fil i ett commit. Vi kanske har löst två buggar på en gång men vill committa in dem som två commits. Det är för dessa situationer som vi har riktig nytta av att använda "Stage" och "Unstage".

1. Lägg till följande kod i vår R-fil och spara filen.

```
# This is a comment to the first commit
# This is a comment to the second commit
```

2. Vi ska nu lägga till dessa kommentarer i vårt repo som två commits, där vi commitar in ett par rader i taget. Vi gör det genom att markera de rader vi vill commita i den högra vyn. Därefter klickar vi på "Stage lines". På detta sätt kan vi välja ut vilka rader vi vill markera för att commita in i repot.

The screenshot shows the MittRepo (Git) application window. The left sidebar includes 'FILE STATUS' (Working Copy), 'BRANCHES' (master), and 'STAGED FILES'. The main area displays a list of staged files under 'Pending files, sorted by path'. A file named 'kod/example_R.R' is selected, showing its content in a diff viewer. The content of the file is as follows:

```

1 1
2 2
3 3
4 - my_data <- read.csv("data/iris.csv")
\ No newline at end of file
5 +
6 + # This is a comment to the first commit
7 +
8 + # This is a comment to the second commit
\ No newline at end of file

```

The commit message field at the bottom is empty.

3. Commita nu de rader som du markerade ovan. När detta är gjort återstår den del av filen som vi inte redan markerat och committa in.

The screenshot shows the MittRepo (Git) application window after committing the changes from step 3. The staged files list now shows the file 'kod/example_R.R' with a status of 'Unstaged files'. The diff viewer shows the same content as before, but the last two lines have been removed, indicating they have been committed.

4. Commita in den sista kommentaren i repot.

Commits och commit messages

När det gäller exakt vad som bör ingå i ett givet commit beror det på från person till person. En bra rutin är att commita en bit kod, data eller text den lilla delen är ”klar”, oavsett om det som korrigeras

är ett enskilt tecken som orsakade en bugg eller om det är en helt ny funktion.

OBS! Committa inte in en del kod som inte fungerar.

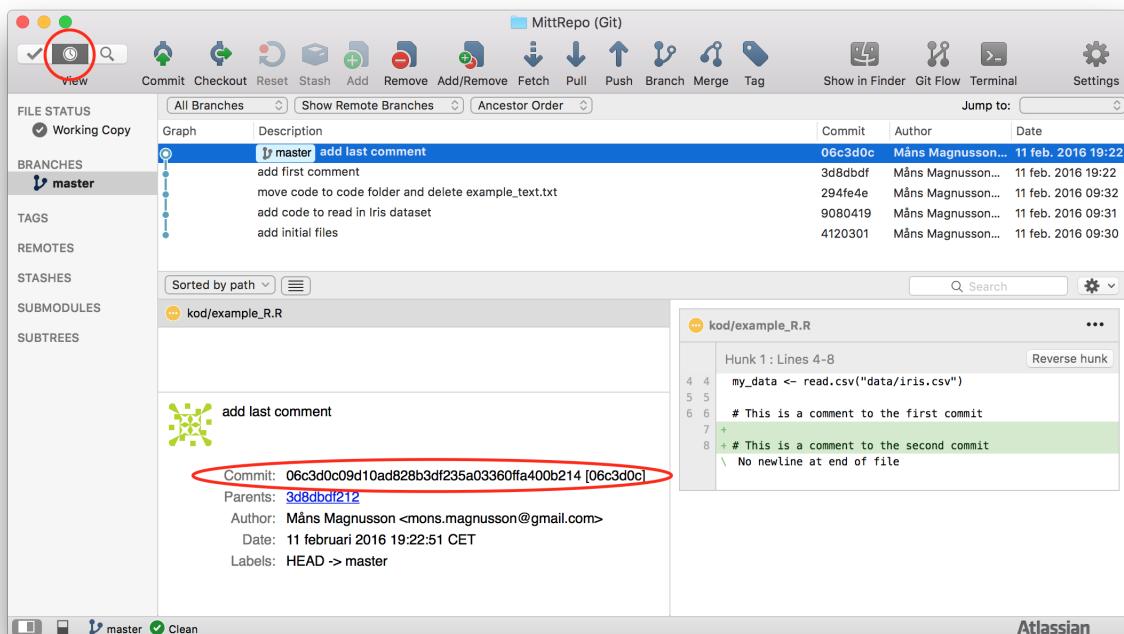
De meddelanden som anges till repektive commit är viktiga. Det är enkelt att efter tag bara skriva ”uppdateringar”, vilket är intetsägande både för sig själv eller andra. En best practice är att tänka att ett commitmeddelande ska kunna fortfölja följande mening:

If applied, this commit will [your commit message here]

Commit hash

Varje commit får en hash-kod. Denna hashkod är unik för varje commit och blir på så sätt ett fingeravtryck för exakt hur koden såg ut vid exakt denna tidpunkt. Det är inte ovanligt att program som körs loggar commit hashen i rapporter eller dylikt för att veta exakt hur koden såg ut när analysen gjordes.

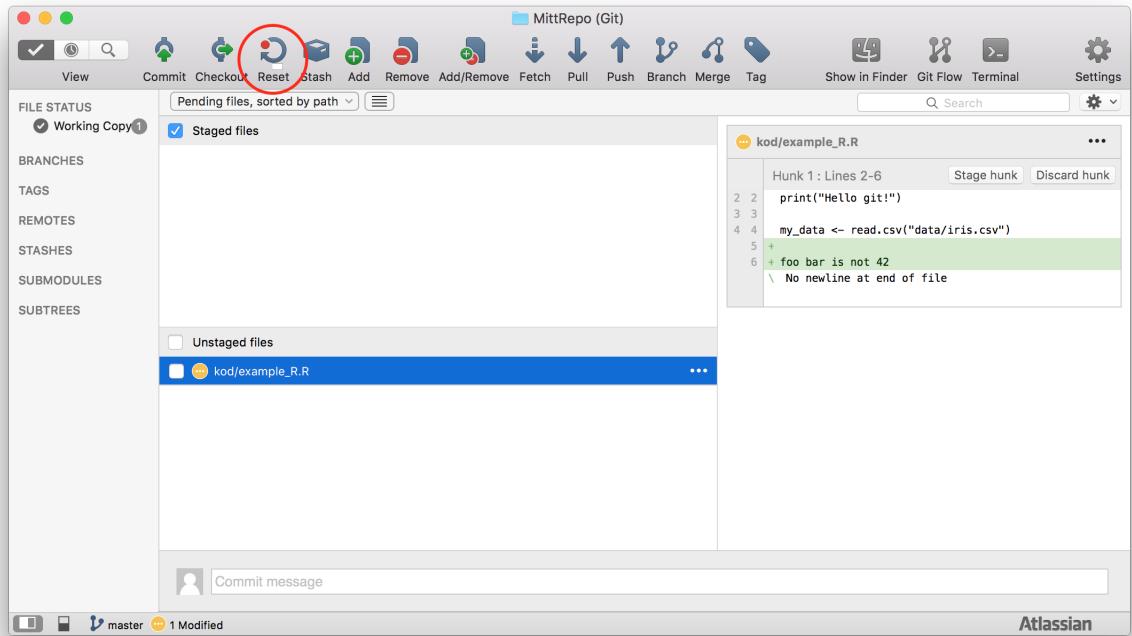
För att se historiken över vilka commits som finns i vårt repo, klicka på klockan i övre vänstra hörnet i SourceTree. Då kan vi se de commits vi gjort och deras commit hash (markerad nedan).



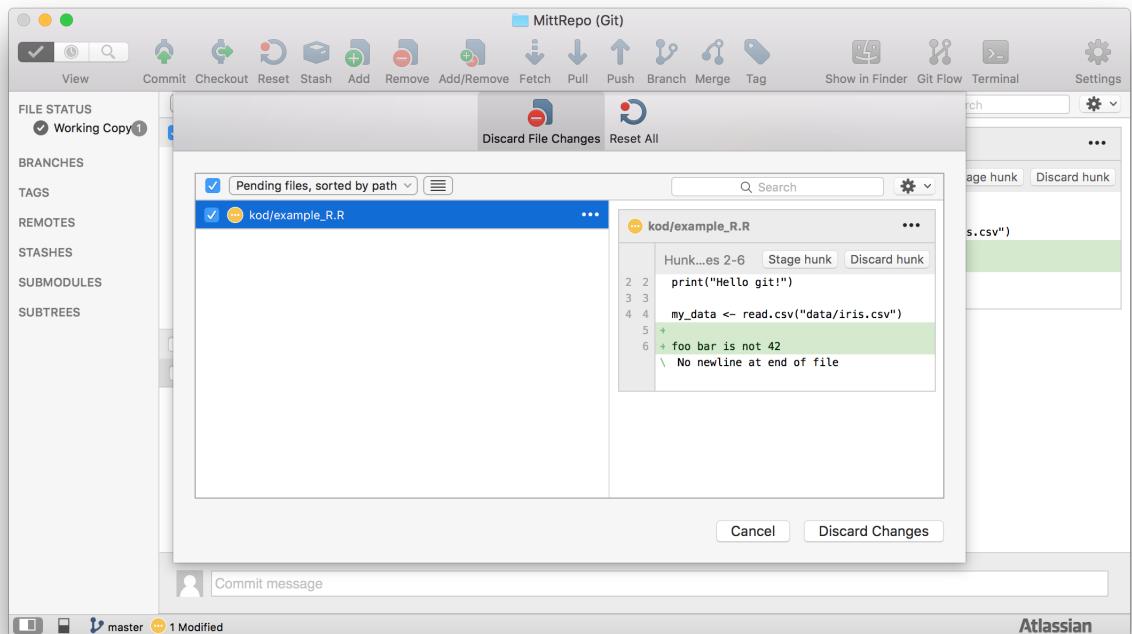
2.4 Reset unstaged changes

Att kunna versionshantera kod är bra, men vi vill också kunna ångra förändringar vi gjort och sparat i våra versionhanterade filer.

1. Gå in i `example_R.R` och lägg till en rad kod med text. Det spelar ingen roll vad det är. Spara filen.
2. Nu kan vi se i SourceTree att det skett en förändring i vår fil som vi inte vill ha kvar. Markera filen i SourceTree och klicka på **Reset**.



3. Vi kan nu markera de filer vi vill återställa till den senaste commiten vi gjort. Det som är markerat i grönt är det vi lagt till och som nu kommer tas bort. Klicka **Discard changes**, och sedan **Ok**. Nu har vi återställt den förändring vi gjort (och sparat över den gamla filen med).



2.5 Diff

Nu när vi lagt till och hanterat våra filer kan det vara intressant att jämföra olika commits. Exempelvis om vi vill se vilka förändringar vi gjort totalt sett över flera commits.

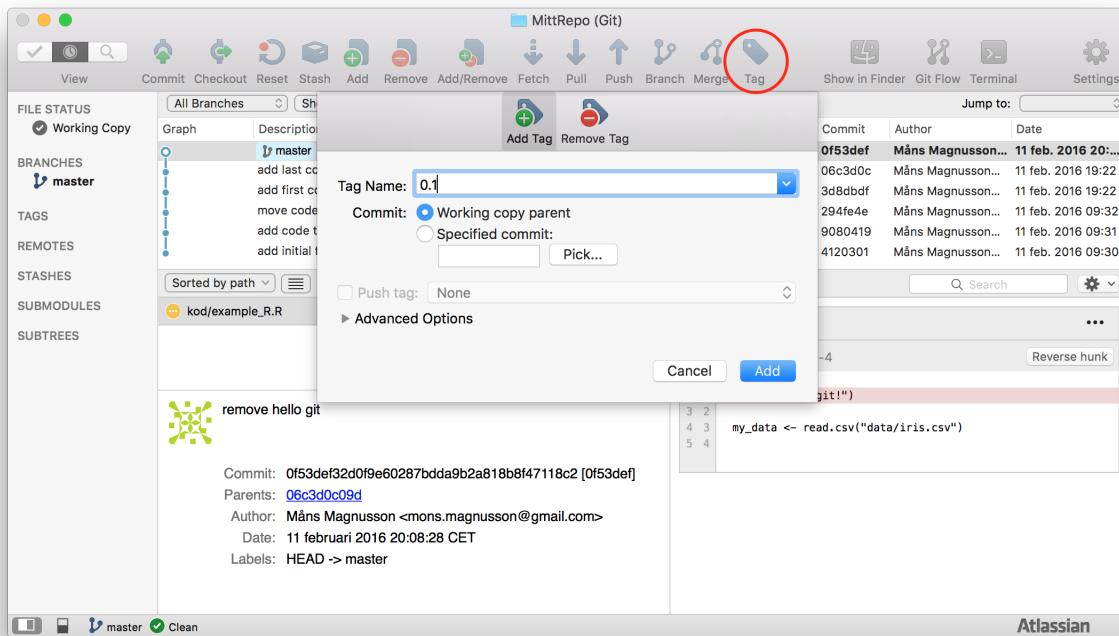
1. Öppna filen `example.R.R` och ta bort raden med `print('Hello git!')`
2. Spara, markera och commita denna förändring i repot.
3. Nu kan vi markera den sista commiten och den tredje commit som vi gjorde. Då kan vi se den totala skillnaden i menyn nere till höger.

The screenshot shows the Atlassian Git tool interface. On the left, there's a sidebar with sections for FILE STATUS (Working Copy), BRANCHES (master), TAGS, REMOTES, STASHES, SUBMODULES, and SUBTREES. The main area displays a commit graph titled "All Branches". A commit from "master" is selected, showing its description: "remove hello git", followed by three detailed log entries: "add last comment", "add first comment", and "move code to code folder and delete example_text.txt". Below this, another commit is listed: "add code to read in Iris dataset" and "add initial files". A message at the bottom says "Displaying all changes between 294fe4e79c73b63b57555ccb6fc1421f07783a6 and 0f53def32d0f9e60287bdda9b2a818b8f47118c2". To the right, a detailed diff view for the file "kod/example_R.R" is shown, specifically for Hunk 1: Lines 1-7. The diff highlights changes in the code, such as the removal of the `print("Hello git!")` line.

2.6 Tags

Vi kan enkelt spåra förändringar mellan två commits, men ibland vill vi märka ut en enskild commit extra (ex. för att vi har en given version). Då kan vi lägga till en tag till en enskild commit. Detta är ett sätt för att märka upp de större stegen som gjorts i ett projekt.

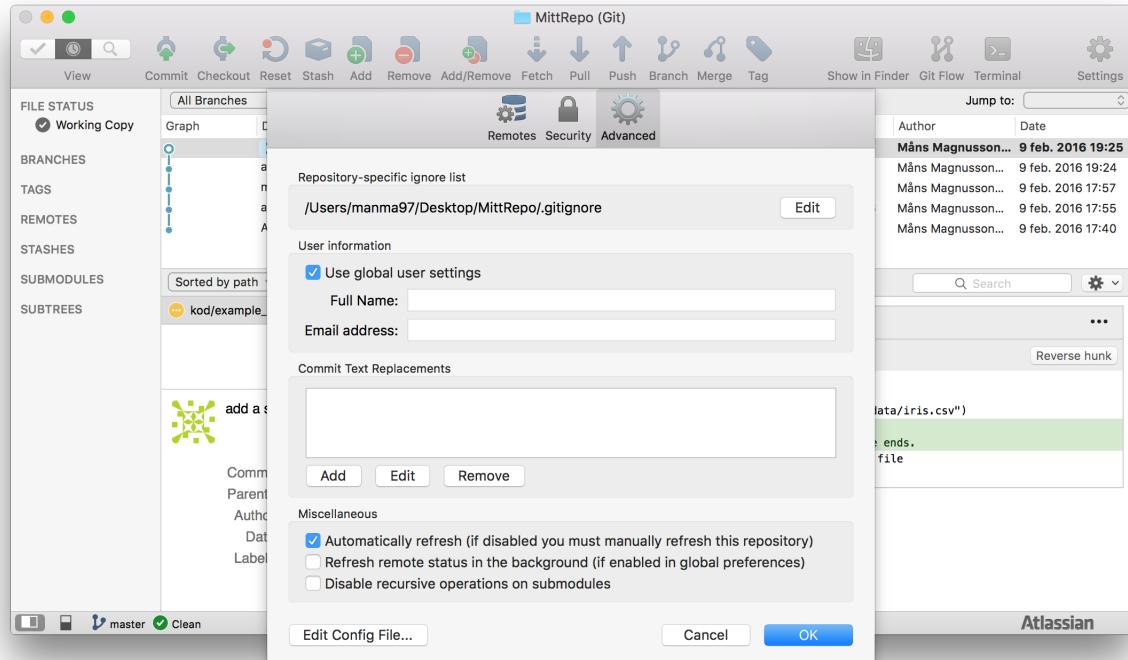
För att lägga till en tag klickar vi på **Tag**. Skriv namnet på taggen och klicka på **Add**.



2.7 .gitignore

Till sist kan det vara så att vissa delar i ett repo vill vi inte att git ska bevaka. För detta finns filen `.gitignore`.

1. Klicka på Repository ⇒ Repository settings... ⇒ Advanced ⇒ Edit



2. En vanlig textfil öppnas då och då kan vi lägga till vilka sorters filer som ska ignoreras. Lägg till: `*.pdf` Nu kommer alla pdf-filer att ignoreras. På liknande sätt kan vi ignorera enskilda mappar. Detta är bekvämt om vi har filer vi jobbar med med vi inte vill att git ska bevaka.

3. `.gitignore` kommer nu dyka upp som en fil i repot (det är bara en vanlig textfil).

Obs! committa inte in `.gitignore` i repot. Detta är en lokal fil som vi inte vill ska följa med repot utan som varje användare av repot själv ska bestämma över. Det enklaste är att lägga till `.gitignore` i `.gitignore`-filen.

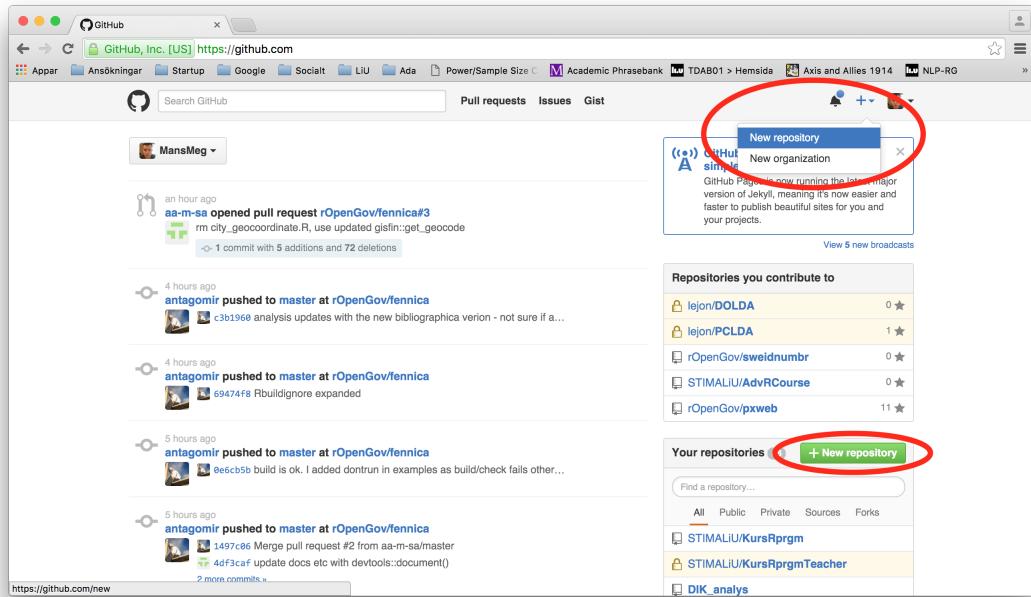
3 Remote repositories

Fram till nu har vi fokuserat på lokal versionshantering. Det fungerar bra om vi jobbar med ett eget projekt där vi inte samarbetar med andra och ingen annan har behov av vår kod. Vill vi däremot arbeta tillsammans med andra behöver vi ett remote repository (kallas också globalt repo i denna text) som lagrar våra commit på ett centralt plats och som andra kan komma åt. Här kommer vi använda github.com som exempel, men det finns andra möjligheter som BitBucket eller GitLab. I praktiken är det bara url-adressen till det globala repot som skiljer sig.

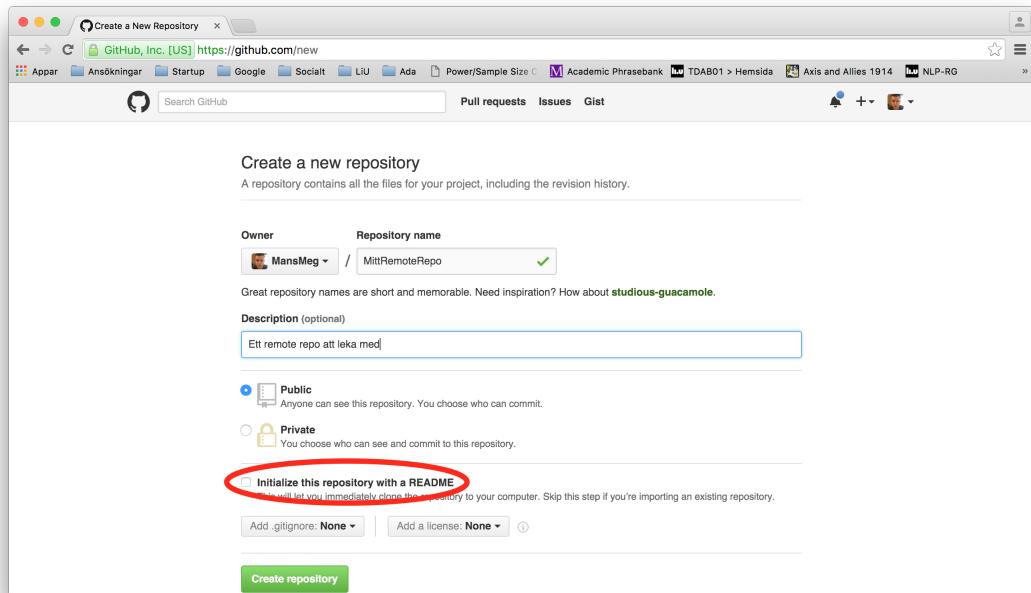
3.1 Skapa ett globalt (remote) repository

Vi gör detta på github, men exakt hur vi skapar ett remote repo är olika för olika tjänster som github, Bitbucket och GitLab.

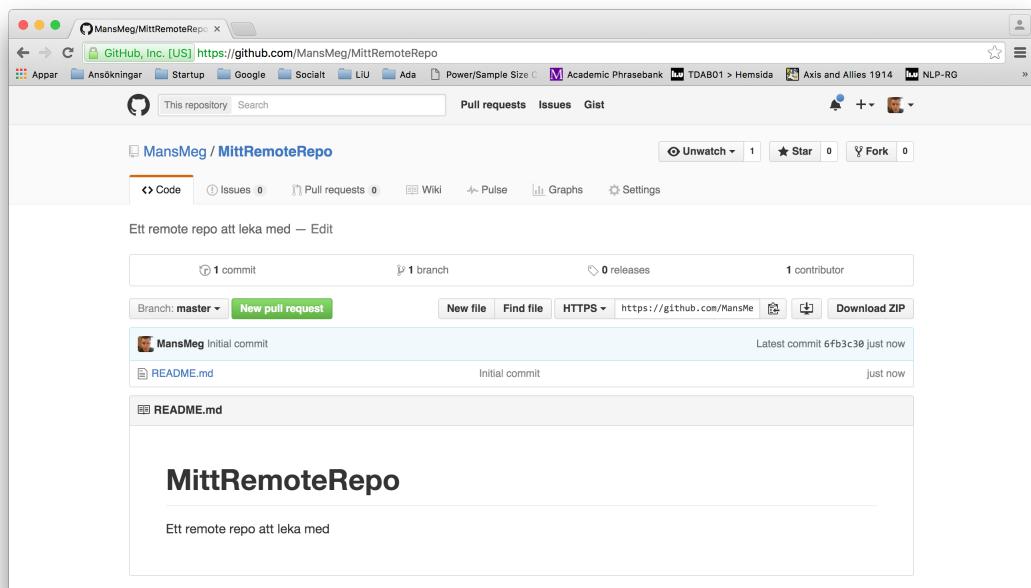
1. Logga in på github.com
2. Skapa ett nytt repo



3. Nästa steg är att ge repot ett namn, beskriva det kort och ange om det ska vara publikt (öppet för alla) eller privat (bara de du bestämmer kommer åt koden). Skapa nu ett öppet repo. Ange att repot ska skapas med en README. Klicka på **Create repository**.



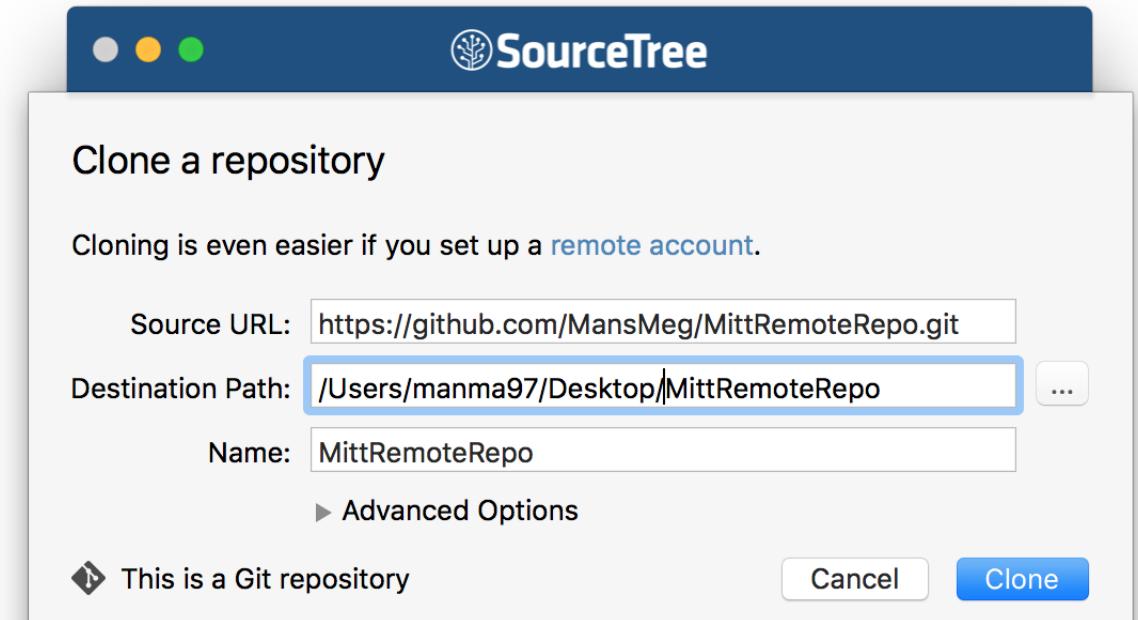
4. Nu har du skapat ett remote repository på github.com. Det borde se ut ungefär såhär:



5. Vi har nu skapat ett remote repo på github. Men eftersom git alltid arbetar lokalt behöver vi klonat detta repo till vår egen dator. Det gör vi i SourceTree genom att välja **Clone from URL**.



- För att klona ett remote repo behöver vi ange den sökväg (URL) till vårt remote repo samt var vi ska klona detta repo (på vår egen dator). Precis som tidigare behöver vi ange en mapp på vår dator som också blir namnet på vårt repo.

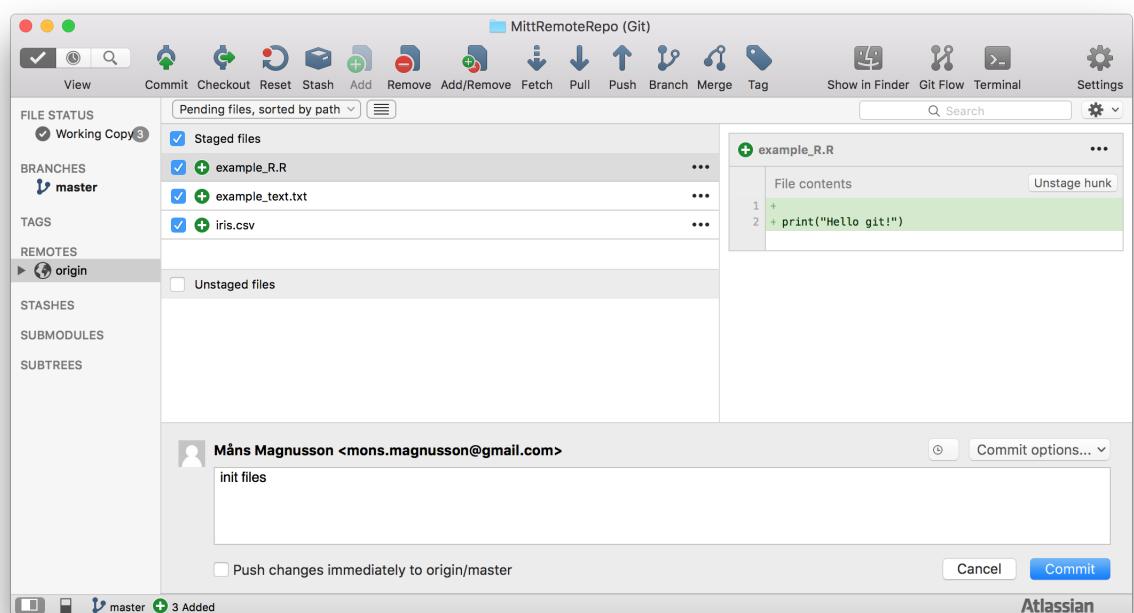


- Nu har vi klonat vårt remote repo så det också finns på vår egen dator.

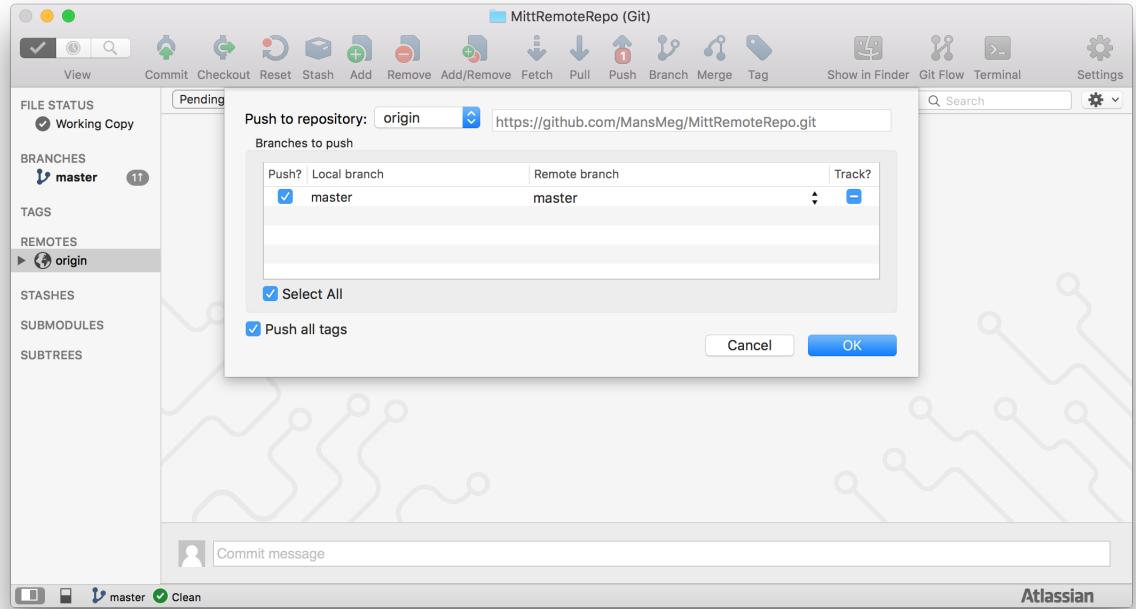
3.2 Push

När vi arbetar med vårt gemensamma repo arbetar vi på samma sätt som när vi har ett helt lokalt repo. När vi är klara och vill dela med oss av vad vi gjort använder vi oss av funktionen **Push**.

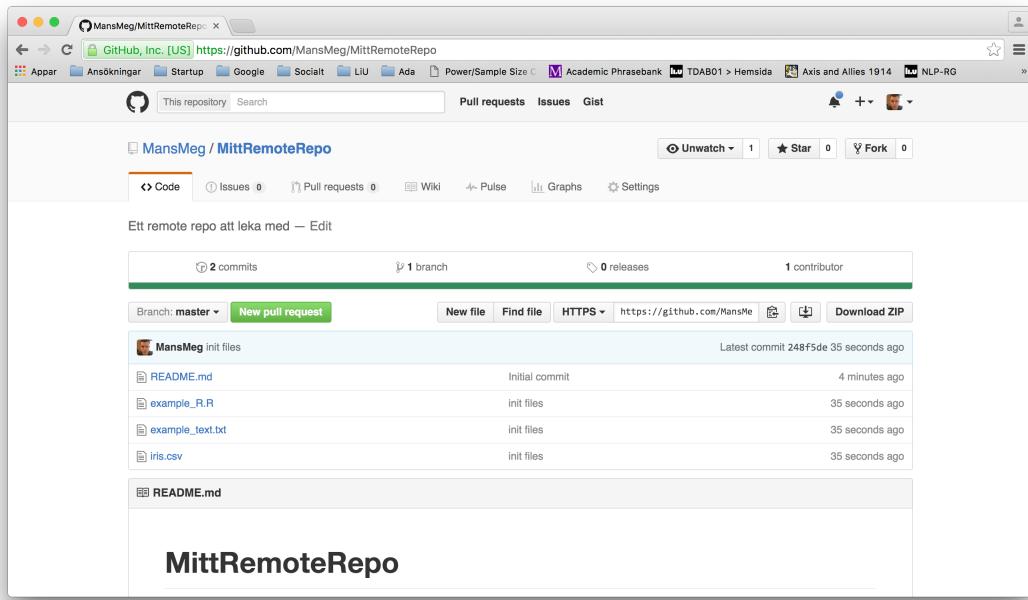
- Markera och committa in samma tre filer som du lade till i det lokala repot vi skapade tidigare. Filerna går att ladda ned från github [[här](#)].



2. Efter att vi lagt till filerna ska vi nu pusha dessa filer till vårt remote repo (som andra kan komma åt). När vi committar i ett remote repo kommer det automatiskt dyka upp en siffra som anger hur många commits vårt lokala repo ligger ”före” det globala repot. Klicka **OK**.



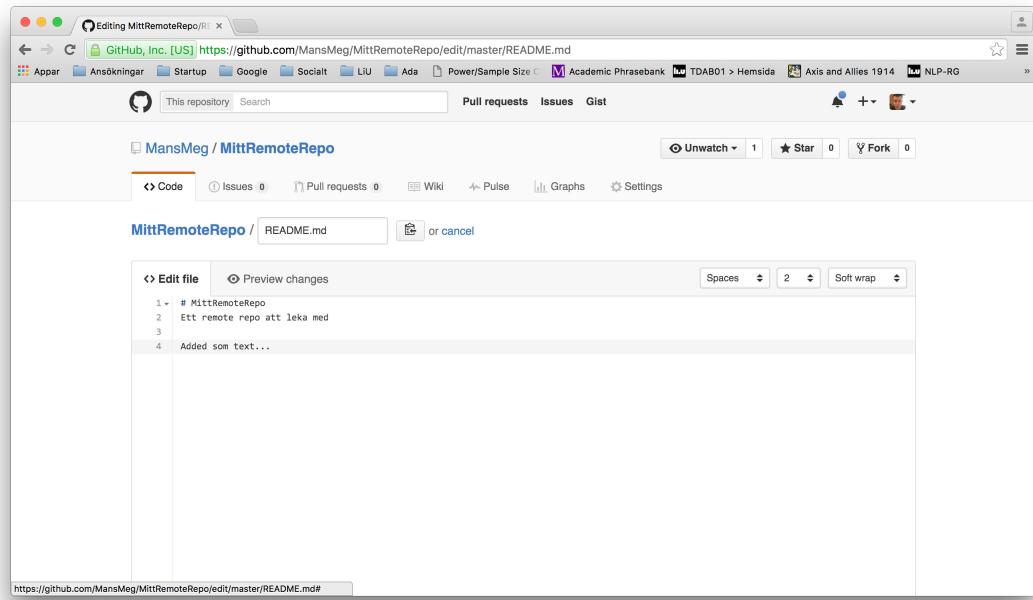
3. Nu kan vi titta på github igen och se att de nya filerna ligger på github och kan användas av andra.



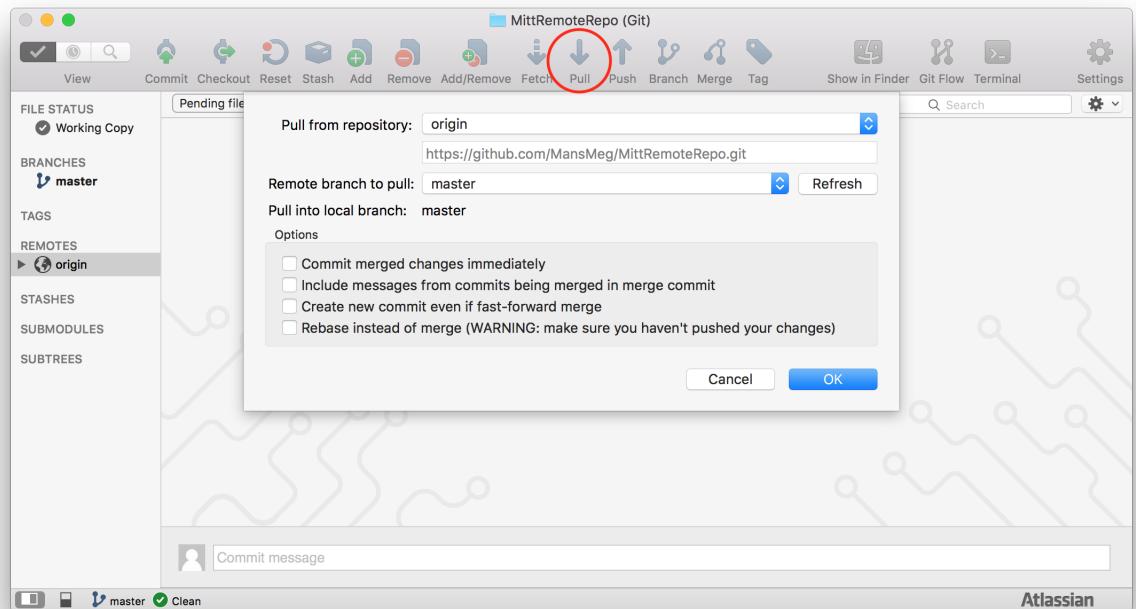
3.3 Pull

Om vi sitter med samma projekt och någon annan har pushat upp ny kod, data eller text behöver vi hämta ned detta för att vi ska kunna fortsätta där vår kollega slutade. Detta gör vi med **Pull**.

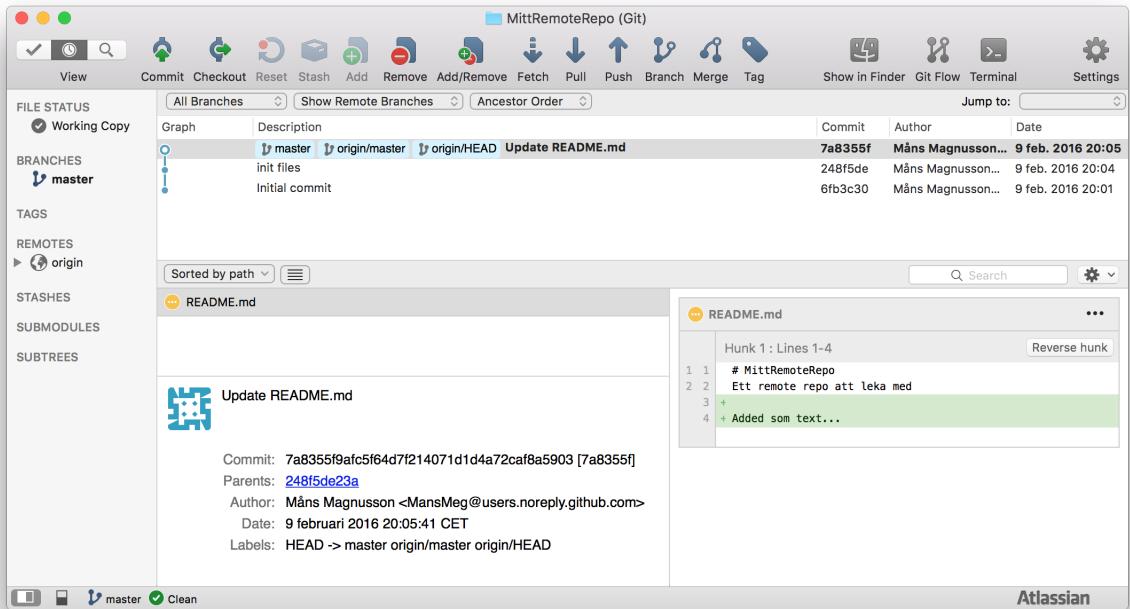
1. Låt oss börja med att ändra vår README direkt på github.



- Nu när vi har gjort en förändring direkt på github kan vi låtsas som att detta är en förändring vår kollega gjort. För att lägga till denna förändring klickar vi på **Pull**.



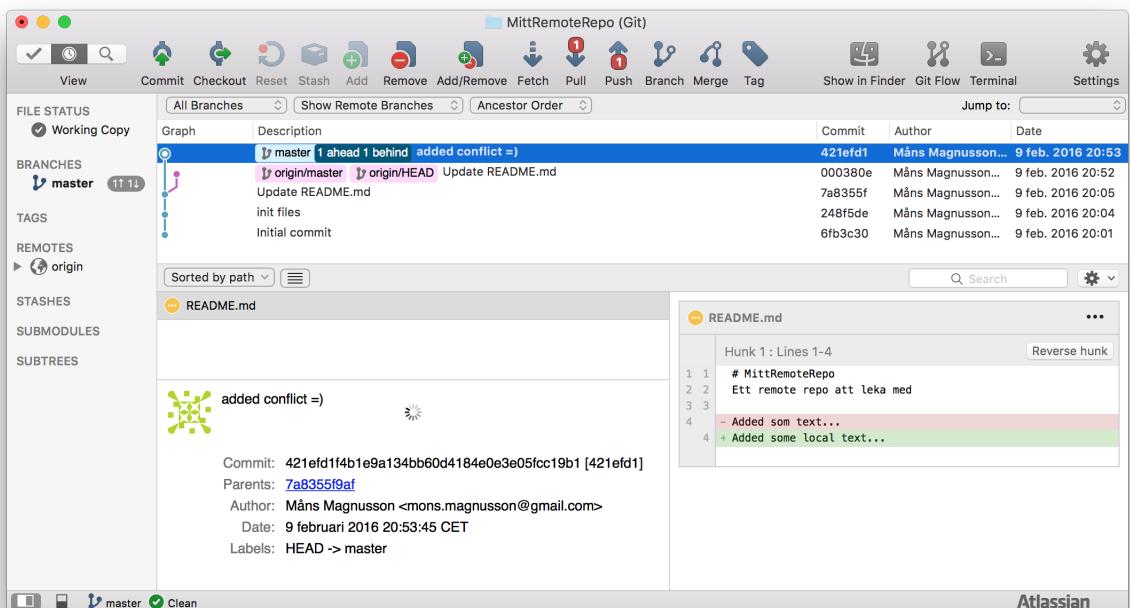
- Nu har vi uppdaterat vårt lokala git repo med de förändringar som vi lagt till i det globala repot.



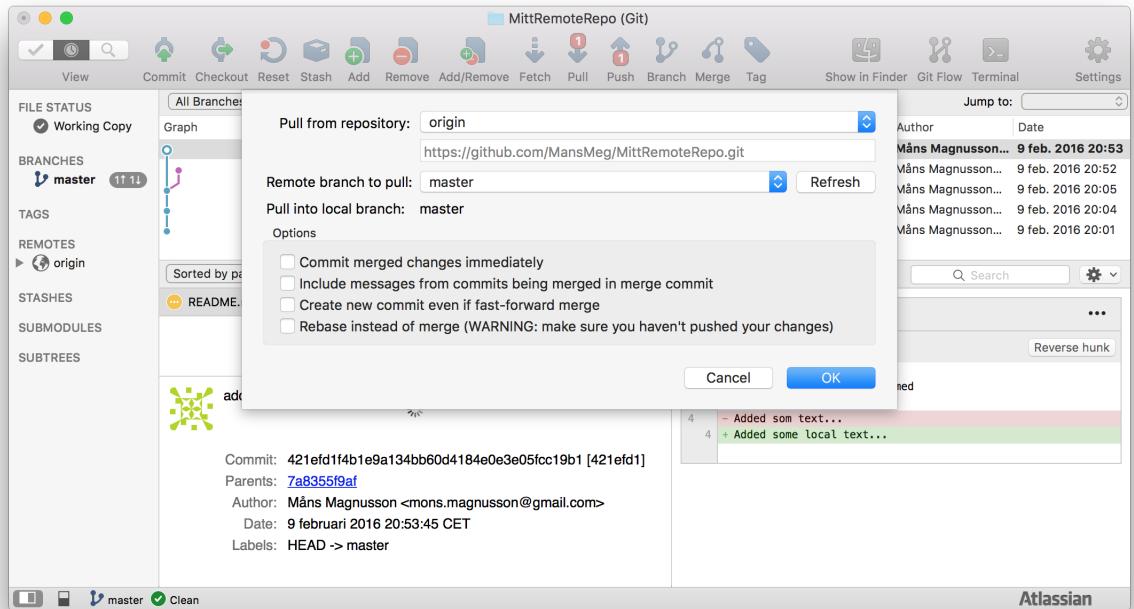
3.4 Conflicts

En av de största problemen med att arbeta med samma projekt flera personer är att det finns en risk att vi gör förändringar i samma fil, på exakt samma ställe. Utan git vore detta en mardröm. Nu ska vi se hur git löser dessa problem åt oss.

1. Ändra "Added some text" i README.md till "Added some remote text på github".
2. Utan att först klicka Pull, ändra på samma rad i det lokala repot README.md till "Added some local text" och commita denna förändring. Nu har vi skapat en konflikt - vi har ändrat på samma rad samtidigt både i det globala och det lokala repot. Vi ser nu att vi både ligger ett commit före och efter vårt remote repo - git skapar två olika grenar.



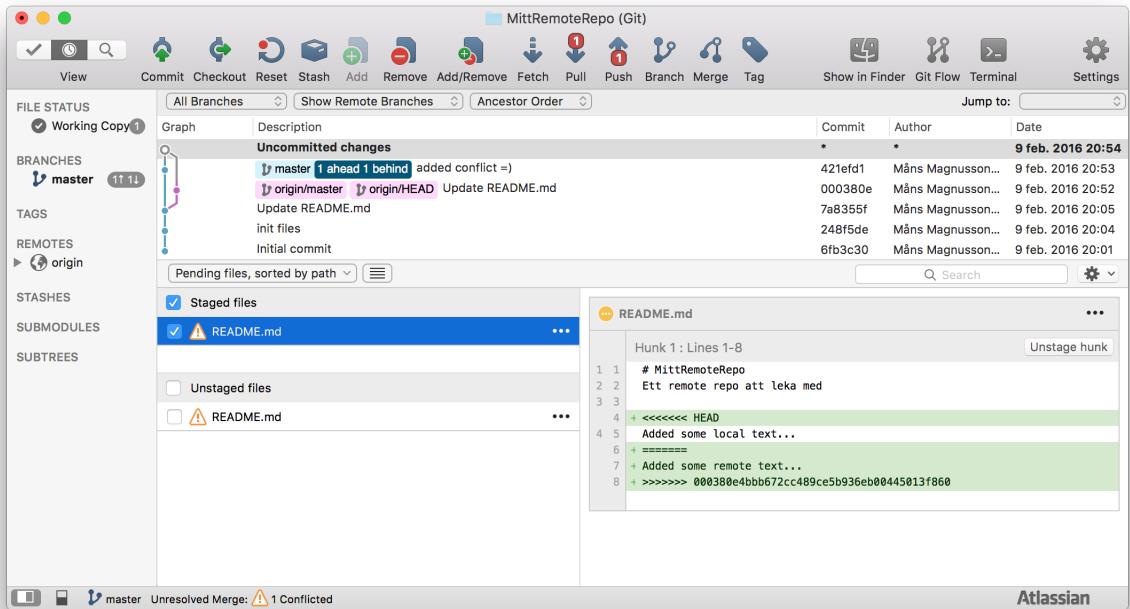
- För att lösa vår konflikt måste vi först använda **Pull** för att dra ned de commits i vårt remote repo som vi saknar lokalt. Vi löser sedan alltid konflikter lokalt. Klicka **OK**.



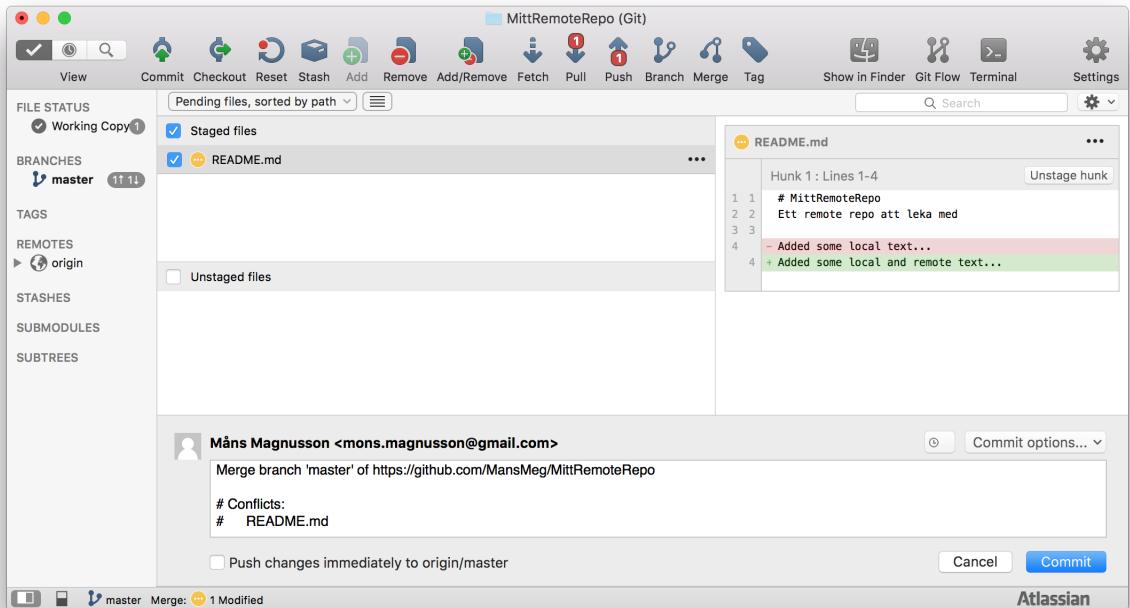
- Nu får vi en varning om att vi har en konflikt vi måste lösa lokalt.



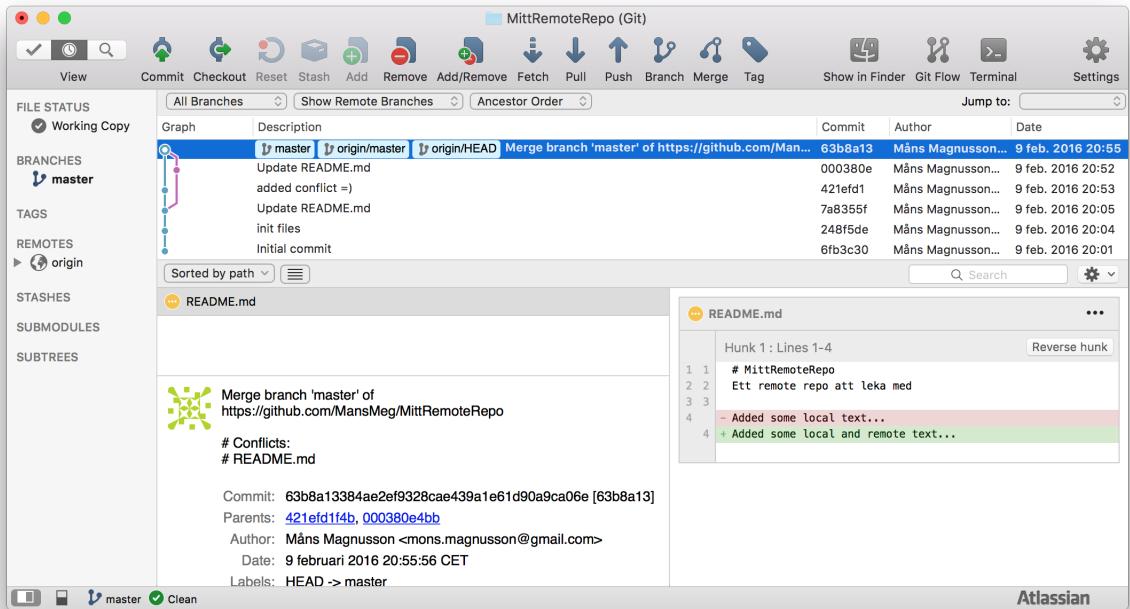
- När det skett en konflikt och vi använder Pull kommer vi få nya filer med skillnaderna mellan filerna som konflikten gäller. För att lösa vår konflikt går vi nu in i vår lokala fil där skillnaden har markerats ut. **HEAD** anger hur vår fil såg ut lokalt och **====** avgränsar den konflikt vi ha med vårt remote repo. Vi får också veta exakt i vilken commit (hashen) som vi hade vår konflikt.



- Nu ändrar vi bara vår fil och väljer vilken version vi vill ha. Vi kan också byta ut den rad konflikten handlar om mot en helt ny rad. Sedan commitar vi in vår fil igen.



- Nu har vi löst vår konflikt. Vi kan nu pusha upp den commit som löste konflikten till vårt remote repo med **Push**. I SourceTree ser det ut som att vi slagit ihop de två olika grenar som skapade konflikten.



4 Branch och merge

Nu kommer vi in på lite mer avancerade delar i git, men som är av stor betydelse för att kunna arbeta effektivt med kod, data och rapporter. Särskilt i större projekt är detta av stor betydelse.

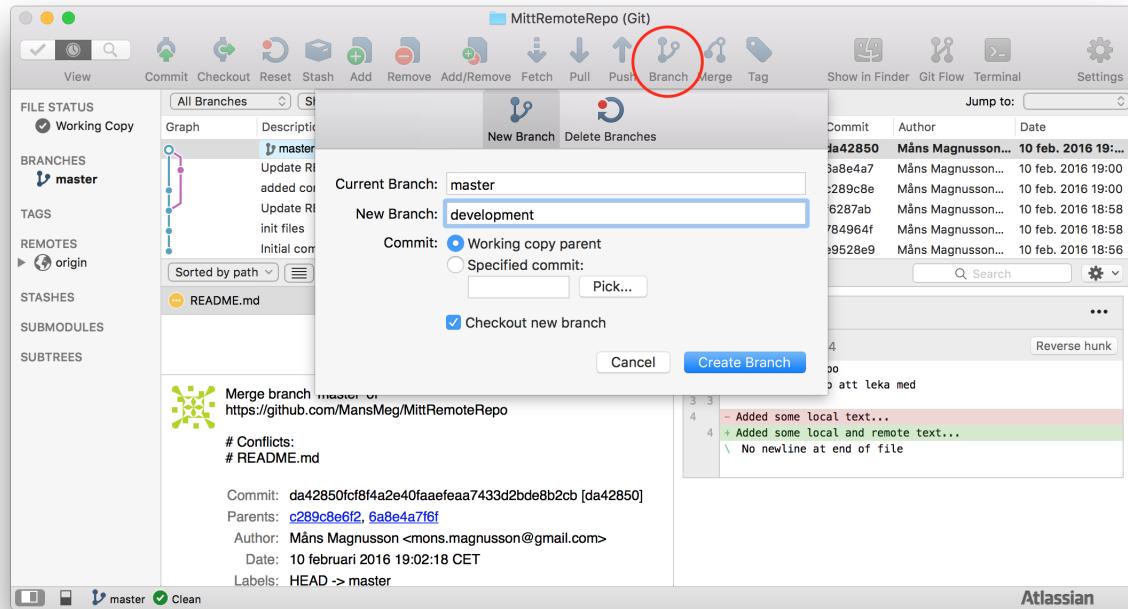
Låt oss säga att vi arbetar med ett projekt där vi ska lägga till en ny funktionalitet i ett skript eller lägga till en ny analys i en löpande rapport. I dessa fall vill vi kunna experimentera och pröva oss fram utan att vi riskerar att förstöra något i den kod som vi redan har och vi vet fungerar. Här kommer "branches" in. Genom att skapa en branch, eller gren, i vårt repo kan vi arbeta vidare i vårt repo utan att vi stör den "huvudsakliga" koden. Denna "huvudsakliga" kod är i sig en branch som heter **master**.

Om vi skapar en ny branch kan vi lägga till nya delar, när vi sedan är nöjda med vårt tillägg kan vi sedan kombinera ihop vår utvecklingsgren med vår huvudgren i repot, vår master branch.

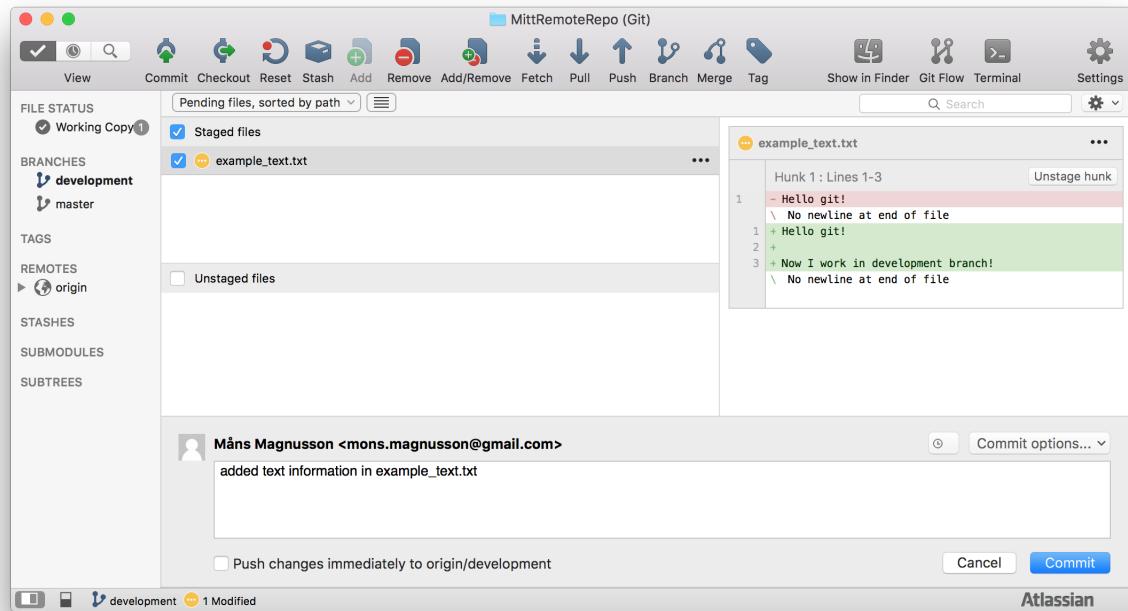
4.1 Branch

Initialt i vårt repo har vi alltid vår huvudgren master, eller master branch. Det är bra att se master som den huvudsakliga grenen i repot. En person som bara ska använda vår kod eller läsa vår rapport ska bara behöva titta i repots master branch. Alla andra grenar är till för de som arbetar i projektet och gör tillägg till master branch.

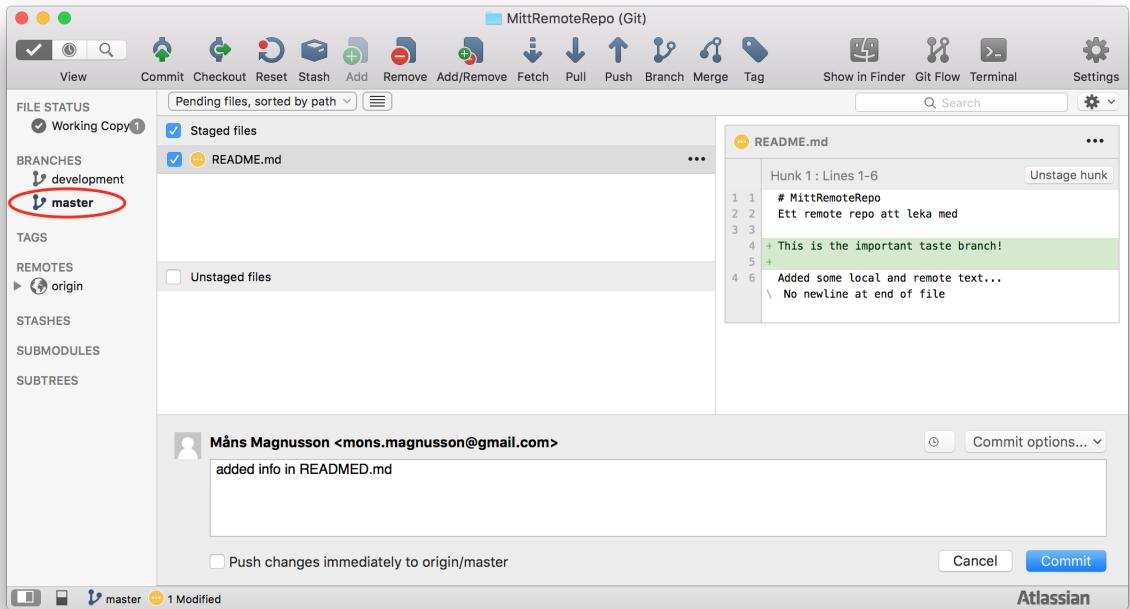
1. För att skapa en ny gren (branch) i vårt repo använder vi funktionen **Branch**. Skapa en branch du kallar "development". För att se vilka branches vi har i vårt repo kan vi använda den vänstra vyn.



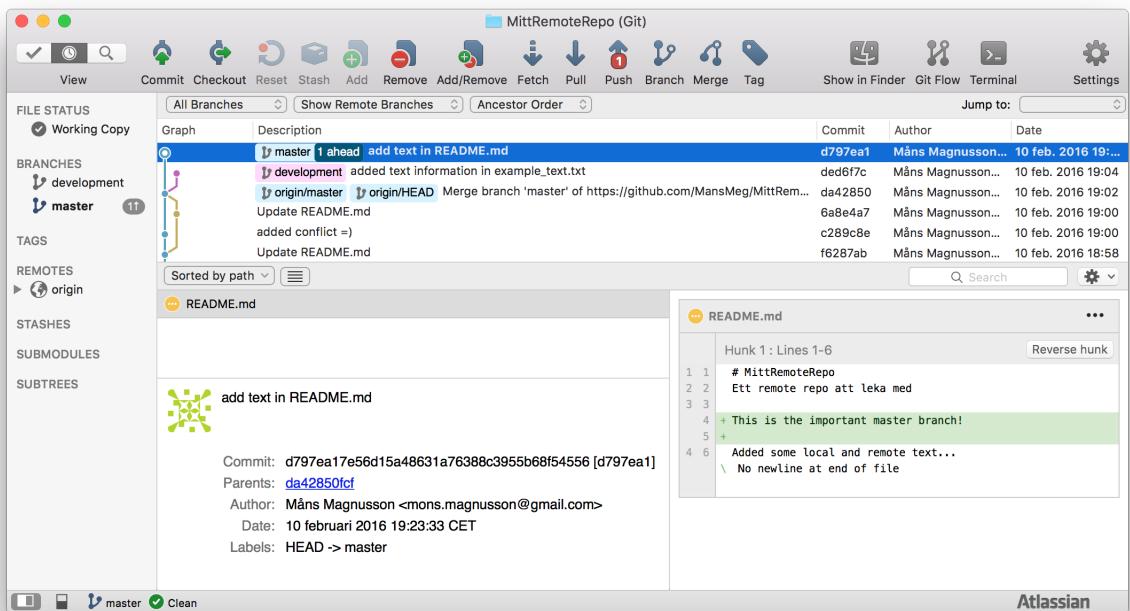
2. Nu har vi skapat en ny gren vi kallar ”development” och den syns i den vänstra vyn. Dubbelklicka på **development** för att byta branch. Ändra nu i vår R-fil och lägg till kommentaren som framgår nedan. Committa in förändringen i development branch. Det borde då se ut på följande sätt.



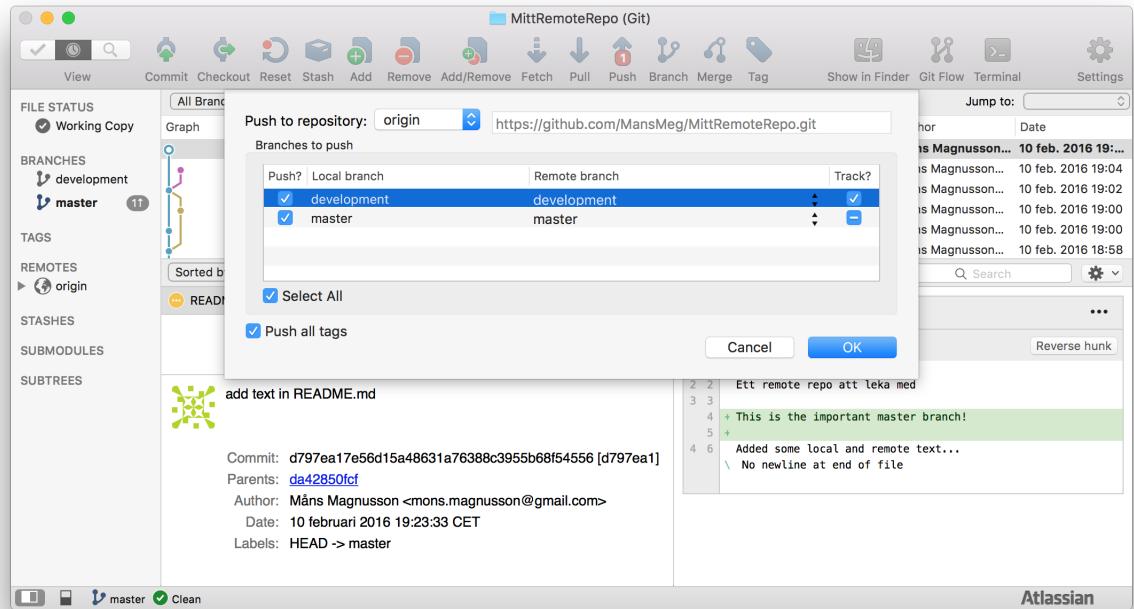
3. Byt nu tillbaka till vår master branch. Klicka på master i vänstervyn och committta in följande text i README.md.



- Nu borde det se ut på följande sätt. Under ”Graph” kan vi också se en bild över de två grenarna vi nu har, development och master.



- Pröva att byta branch från development till master och vice versa. Vad händer med filerna `README.md` och `example_text.txt` när vi byter branch?
- Som ett sista steg ska vi nu också pusha upp vår nya branch och våra commits till vårt remote repo (så andra kan ta del av vad vi gjort). Klicka i att du vill pusha båda våra branches och klicka **OK**.

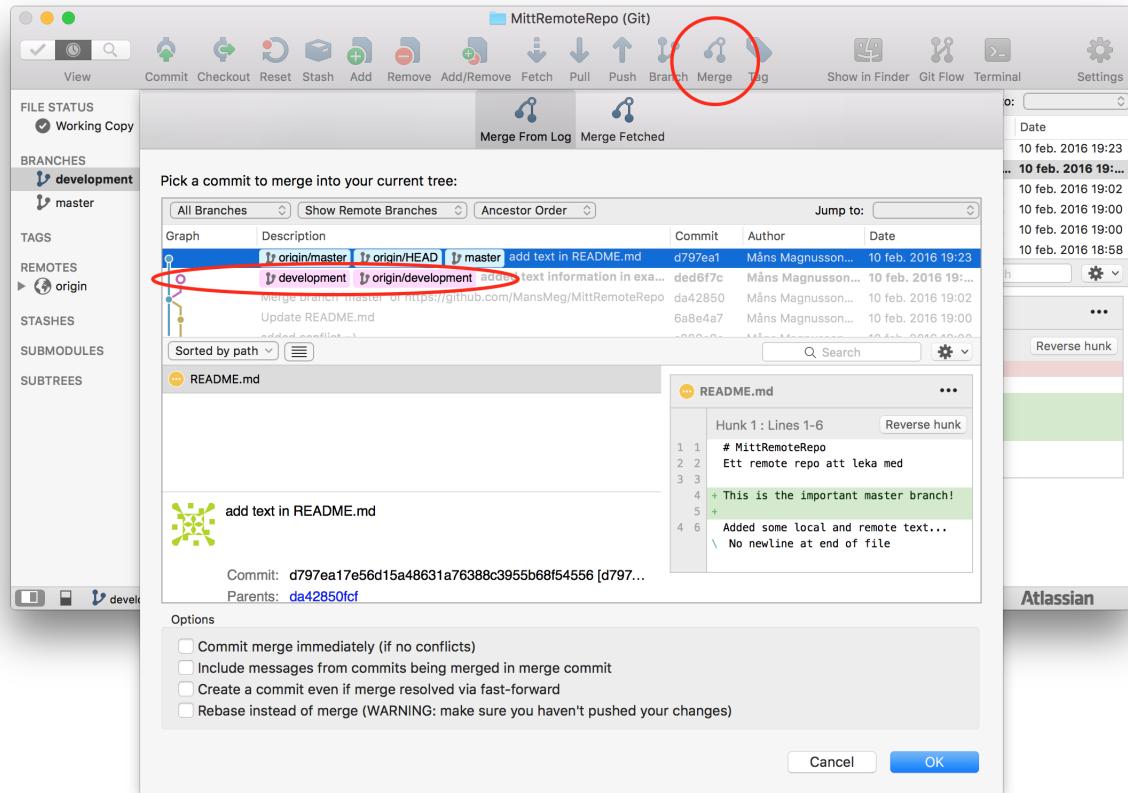


- Nu har vi pushat både master branch och development branch till vårt remote repo. Vi ser det genom att det finns en origin/development- och origin/master-indikator.

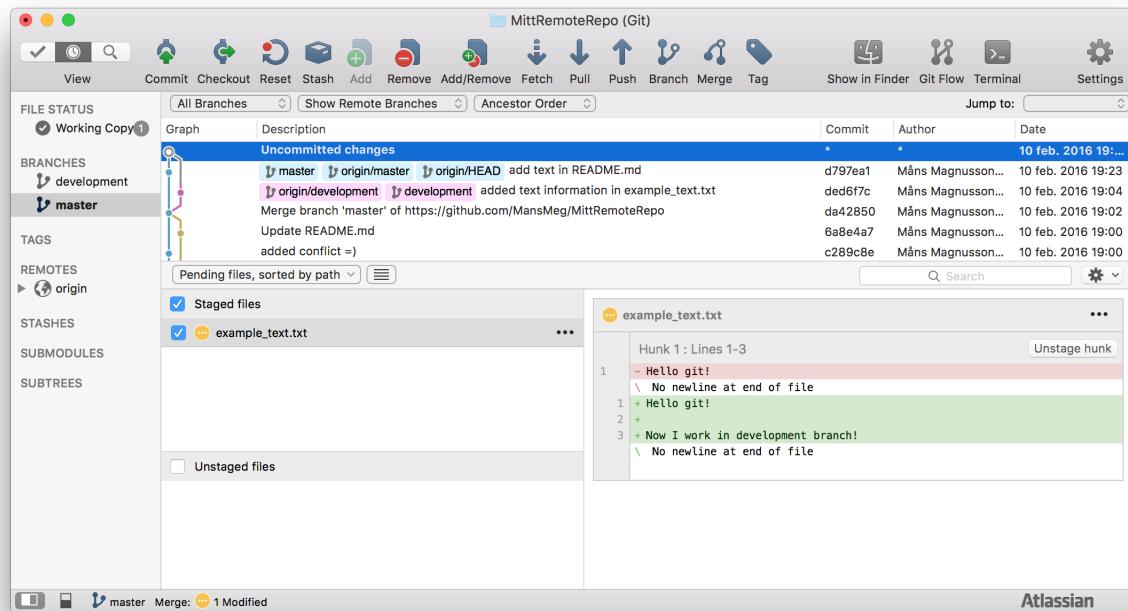
4.2 Merge

När vi arbetat klart med en enskild gren och vi vill lägga till denna gren till vår huvudgren, master, använder vi Merge för att slå ihop grenar.

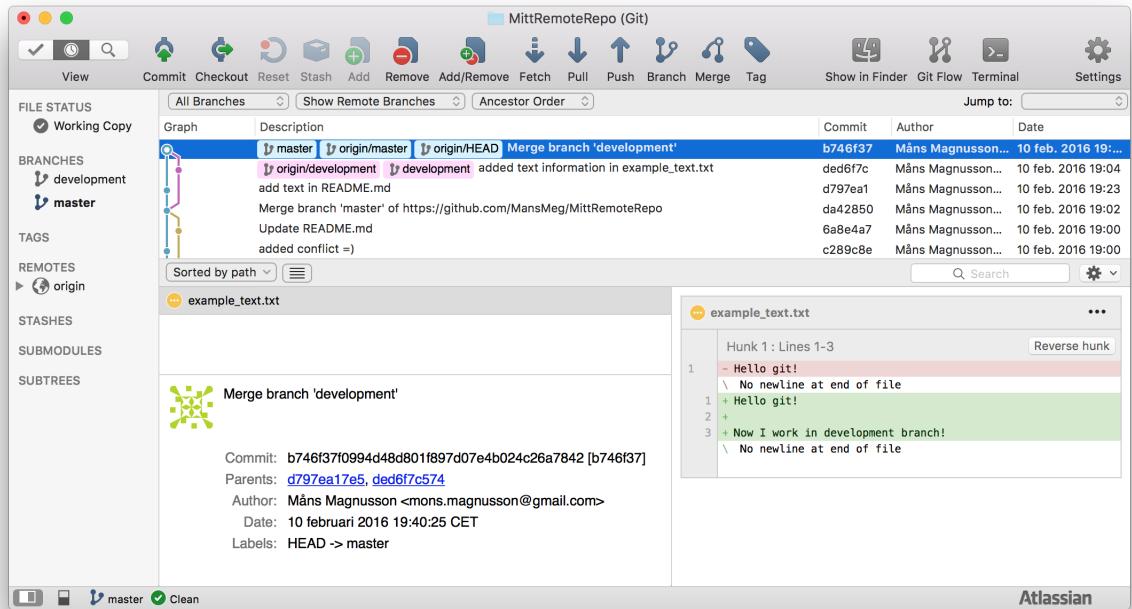
- För att slå ihop en gren behöver vi först aktivera den gren vi vill lägga till en annan gren till. Har vi arbetat i en developmentgren och vill lägga till detta arbete till master ska vi aktivera huvudgrenen master.
- Klicka nu på **Merge**. Markera development branch (sista commiten) och klicka **Ok**.



- Nu har skillnaden mellan master och development grenarna lagts till i filerna i master branch. För att slutligen slå ihop grenarna behöver vi därför committa in dessa skillnader i vår master branch.



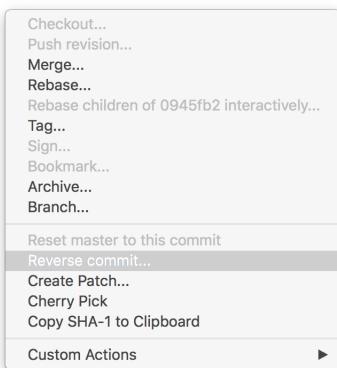
- Committa in dessa förändringar och pusha sedan allt till vårt remote repo. Då borde det se ut som nedan. Vi ser att de två grenarna nu slagits ihop i vår master branch.



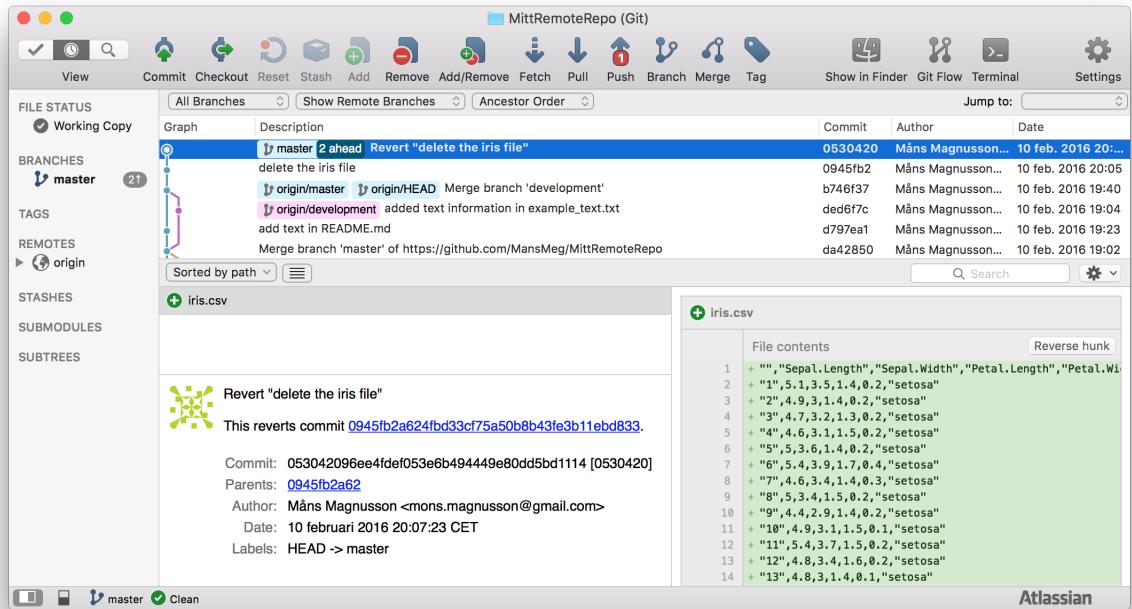
5 Korrigera fel i repot - Revert och Reset

Om vi upptäcker felaktigheter vi har committat in i vår kod kan det vara så att vi vill ”ta bort” enskilda commits. Eftersom git är gjort för att det ska vara svårt att ta bort kod kan det vara lite krångligt. Vill vi ta bort det vi gjort i en eller flera commits finns det två vägar att gå. Har vi inte pushat upp det vi commit kan vi använda reset. Har vi dock pushat upp våra commits till vårt remote repo bör vi istället använda revert. Revert skapar en ny commit som återställer tidigare commits i en ny commit. Reset dock tar bort enskilda commits.

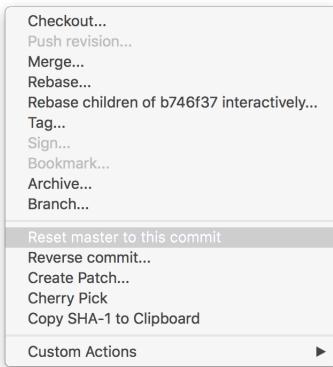
1. Börja med att ta bort filen `iris.csv` från MittRemoteRepo och committa denna förändring.
2. Som ett första steg ska vi nu återställa detta fel med Revert. Högerklicka på det commit vi vill ta bort/återställa och välj **Reverse commit...**



3. Nu har en ny commit skapats som återställer den eller de commits vi valt att återställa.



4. Vi kan också ta bort enskilda commits helt med reset. Klicka på den commit *du vill återställa till*. Exempelvis ”Merge branch ‘development’”. Klicka på **Reset master to this commit**



5. Nu har du tagit bort dessa commits. Kvar kommer de förändringar som gjorts i dessa två commits finnas kvar.

