

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

Måns Magnusson

14 februari 2016

## Innehåll

<b>1 Förutsättningar</b>	<b>2</b>
1.1 Mjukvara . . . . .	2
1.2 GitHub account . . . . .	2
<b>2 Grundläggande git</b>	<b>2</b>
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
<b>3 Remote repositories</b>	<b>15</b>
3.1 Skapa ett globalt (remote) repository . . . . .	15
3.2 Push . . . . .	17
3.3 Pull . . . . .	18
3.4 Conflicts . . . . .	20
<b>4 Branch och merge</b>	<b>23</b>
4.1 Branch . . . . .	23
4.2 Merge . . . . .	26
<b>5 Korrigera fel i ett repo - Revert och Reset</b>	<b>28</b>

# 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. Git och SourceTree fungerar oavsett operativsystem. Git fungerar för all typ av programkod som SAS-, SPSS- och R-kod liksom för data i csv-format och textfiler. Git och SourceTree är gratis och bygger på öppen källkod.

1. Git: Det program som används för att versionhantera 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 konto på github.com för att arbeta med git remote repositories. Skapa ett gratis konto på [github.com](https://github.com).

# 2 Grundläggande git

Git är ett system för att versionhantera filer och förenkla samarbete när det gäller kod och filer i projekt. Det är gjort för att vara snabbt och stabilt. För att börja versionhantera ett projekt med git behöver vi först skapa ett repository.

## 2.1 Skapa ett repository

Det mest grundläggande enheten i git är ett "repository", eller "repo". Det enklast är att se ett repo som en vanlig mapp på datorn och allt vi lägger i mappen versionhanteras av git. Mappen kan innehålla vilka filer vi vill, som kod, data och rapporter. Således är det bara att arbeta vidare som vanligt.

Git är smidigast för att hantera textfiler som programkod, data i csv-format och rapporter i text eller markdownformat. Exakt vilken omfattning ett repo ska ha är inte helt uppenbart, men ofta utgörs ett repo av ett "projekt" i abstract mening.

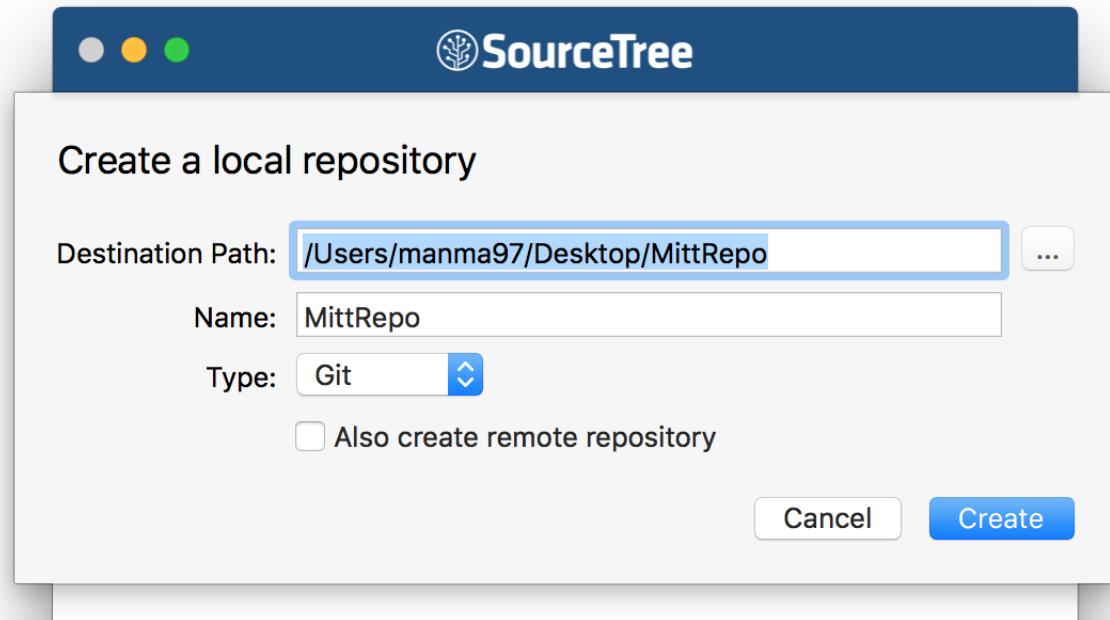
För att skapa ett nytt repo:

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

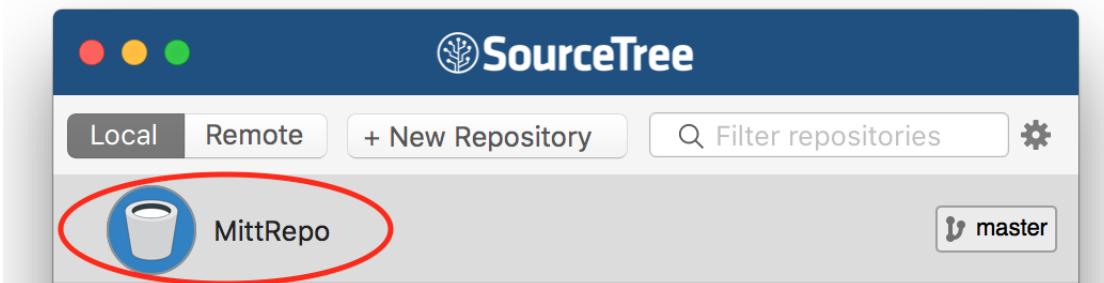


Vi har nu skapat ett lokalt repo, d.v.s. repot kommer bara finnas på din egen dator (vi kommer gå in på s.k. remote repositories senare).

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



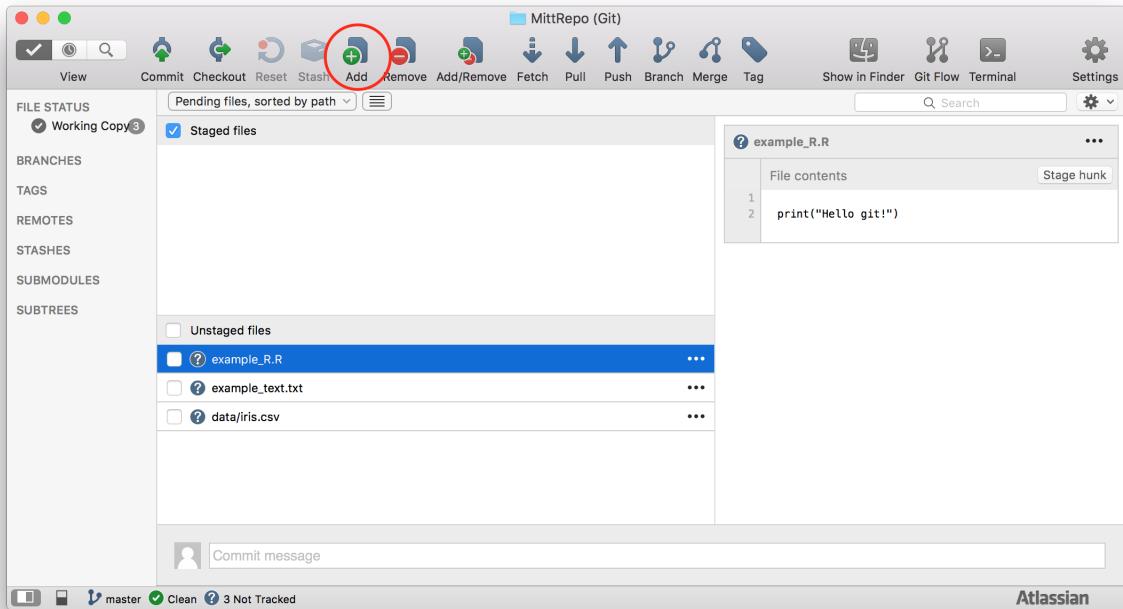
- Nu har vi skapat ett första repo som dyker upp på SourceTrees förstasida. För att börja arbeta med filerna i repot, dubbelklicka på MittRepo (eller det namn du angav).



## 2.2 Lägga till filer i repot (Add)

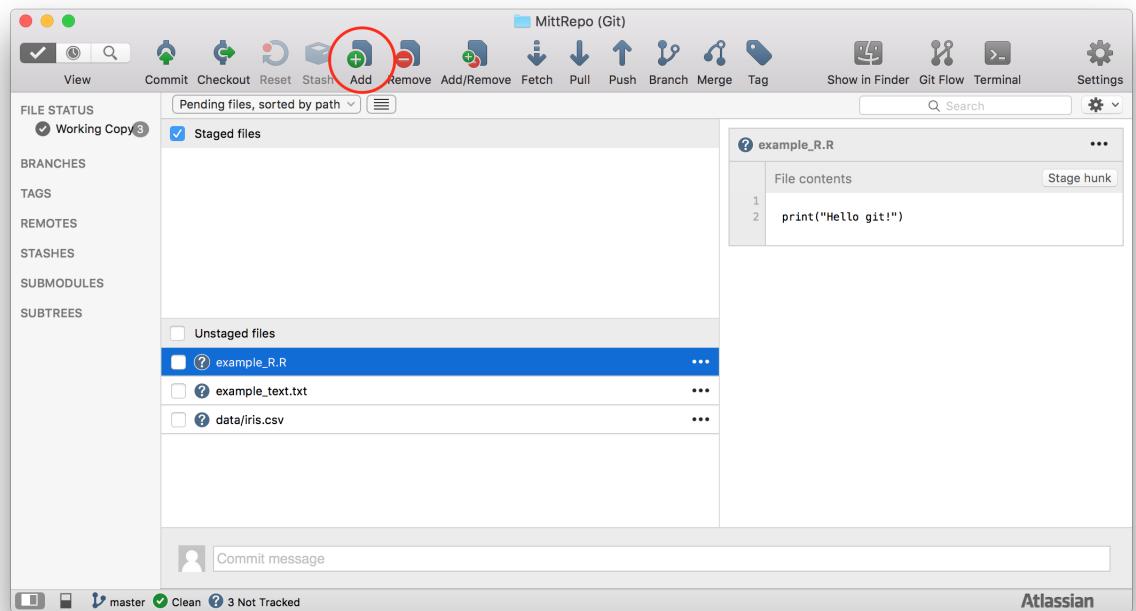
Nu har vi skapat ett helt nytt repo, lokalt på din dator. Men än har vi inte börjat versionhantera några filer. Nu måste vi lägga till de filer vi vill versionshantera. Det finns tre stycken exempelfiler som vi kommer använda i följande exempel. Dessa filer går att ladda ned från github [[här](#)].

Kopiera dessa tre filer till den mapp som utgör det nyligen skapade repot. Filen `iris.csv` ligger i en undermapp "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 börja versionhantera. Markera de filer du vill lägga till och klicka på **Add**.

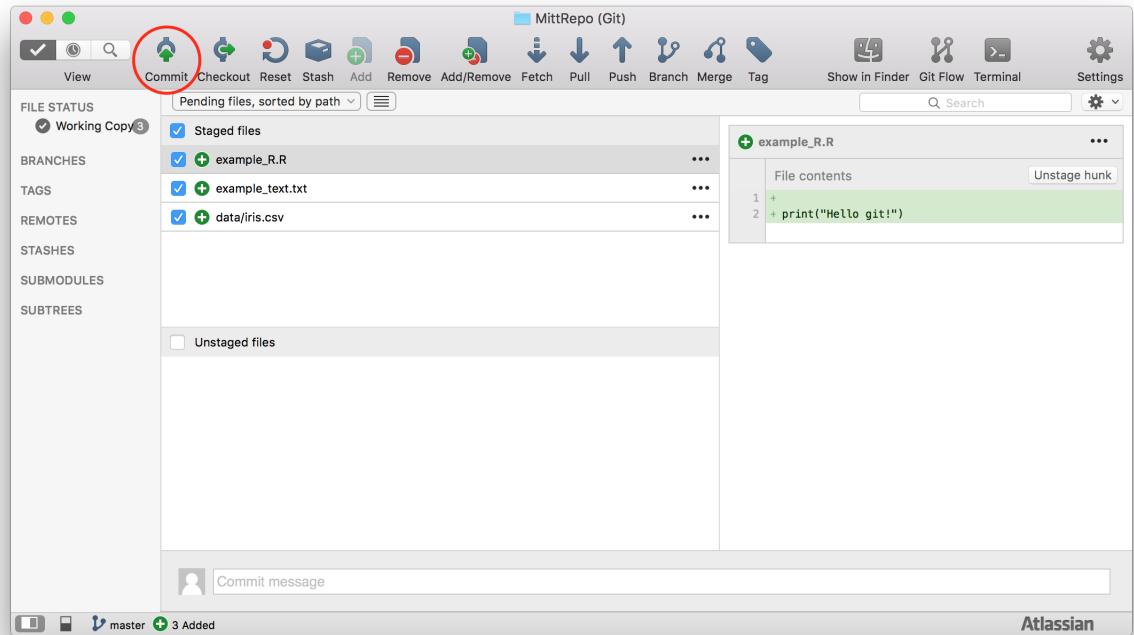


Detta innebär att vi har markerat de filer vi vill lägga till i vårt repo och börja versionhantera. Det vi gör är att vi flyttar filerna till "Stage", vilket kan ses som att vi markerar dessa filer för att sedan, i nästa steg, lägga till dem i vårt repo som en commit.

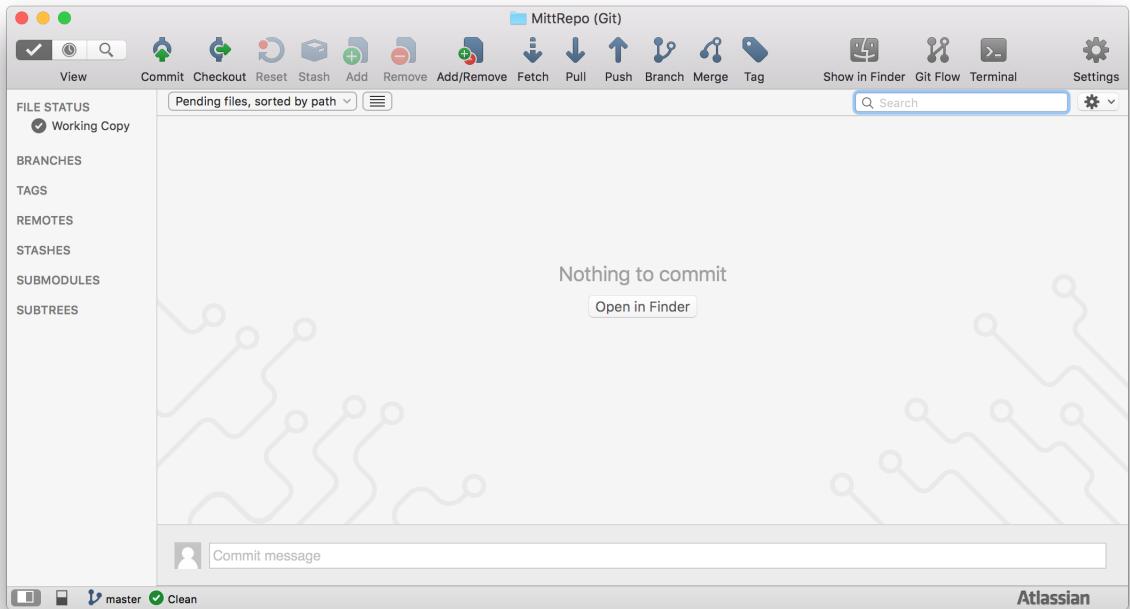
2. Nästa steg är att vi ska lägga till de filer vi markerat. Detta gör vi med genom att göra en **Commit**. När vi gör en commit "fryser" vi de förändringar vi gjort i en fil i vårt repo. I detta fall har vi lagt

till nya filer så vi fryser filerna som de såg ut när vi lade till dem i vårt repo.

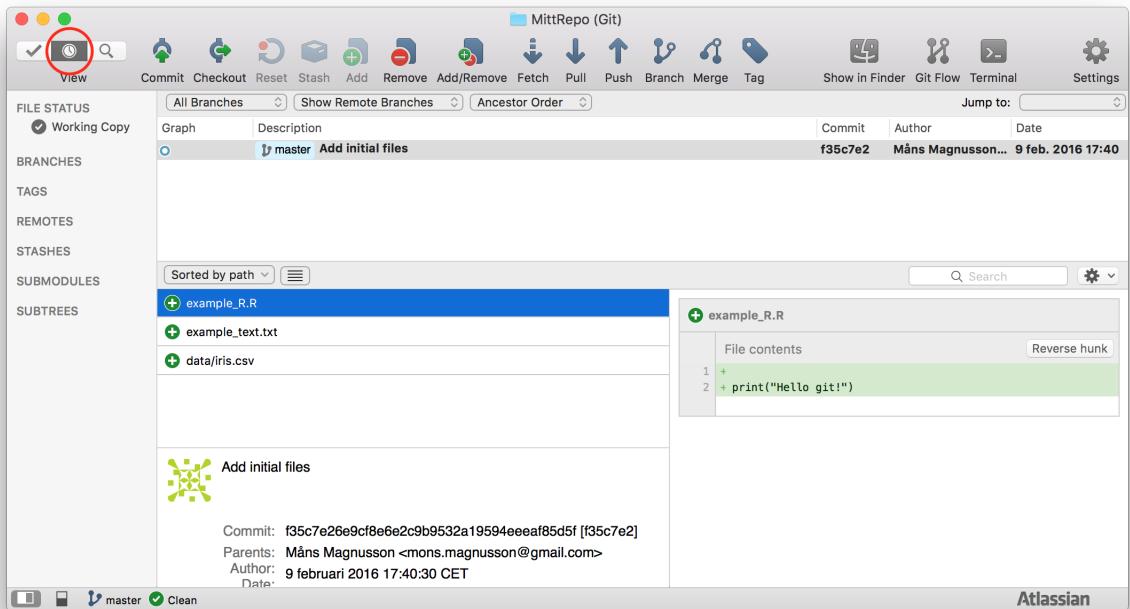
När vi klickar på **Commit** får då upp att vi ska ange ett ”commit message”. Detta är ett meddelande vi kommer gå in på mer i detalj senare. Nu anger vi ”Add initial files” som meddelande. Klicka sedan på knappen **Commit**.



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



4. Vi kan vilja se vilka förändringar vi gjort över tid i vårt repo (för spårbarhet). 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 commits.



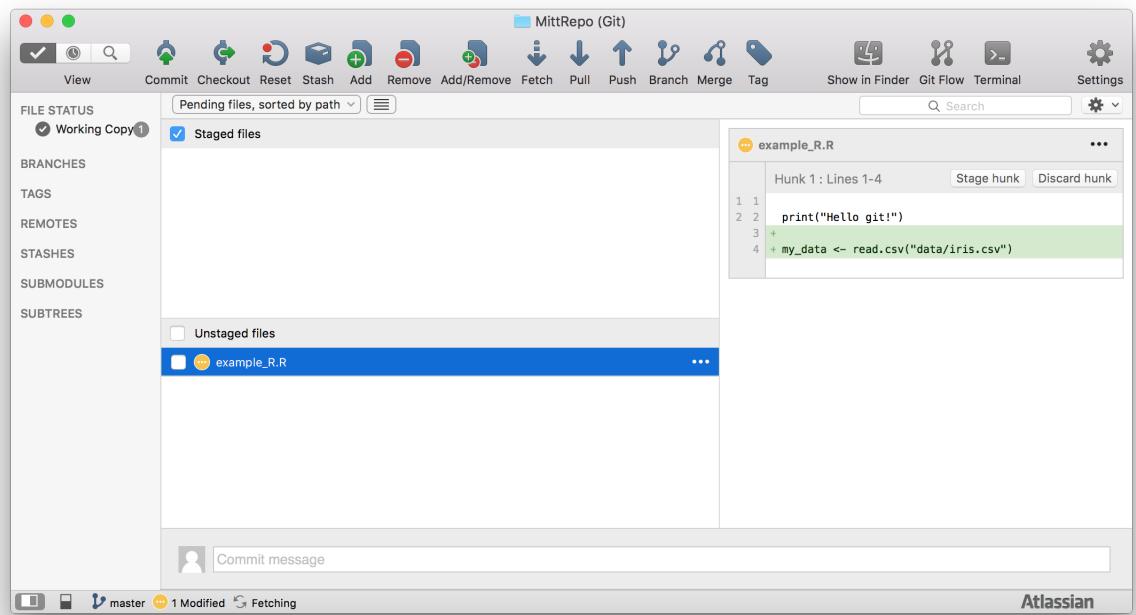
## 2.3 Stage och Commit

Versionshantering med git handlar om att vi gör förändringar i våra filer och sparar filerna. Sedan vill vi lägga till dessa förändringar i git som en eller flera commits. Som ett första steg ska vi ändra i en av våra nu versionshanterade filer. Vi ska lägga till en rad i `example.R.R` där vi läser in csv-filen `iris.csv`.

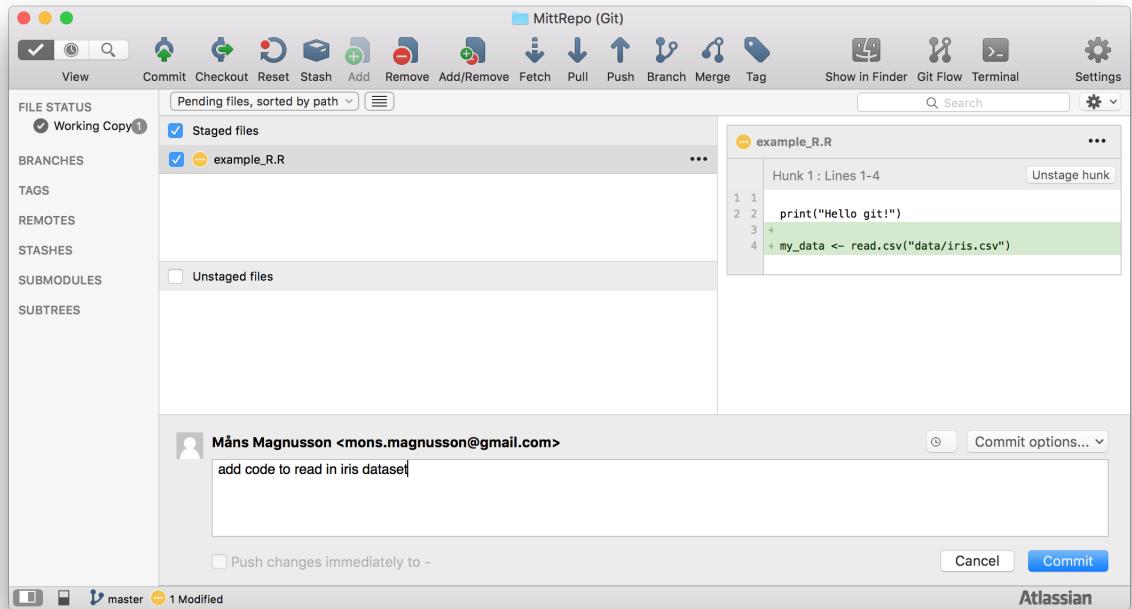
1. Öppna filen `example_R.R` (om du inte har R kan du göra det med en vanlig texteditor) och lägg till följande rad kod:

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

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



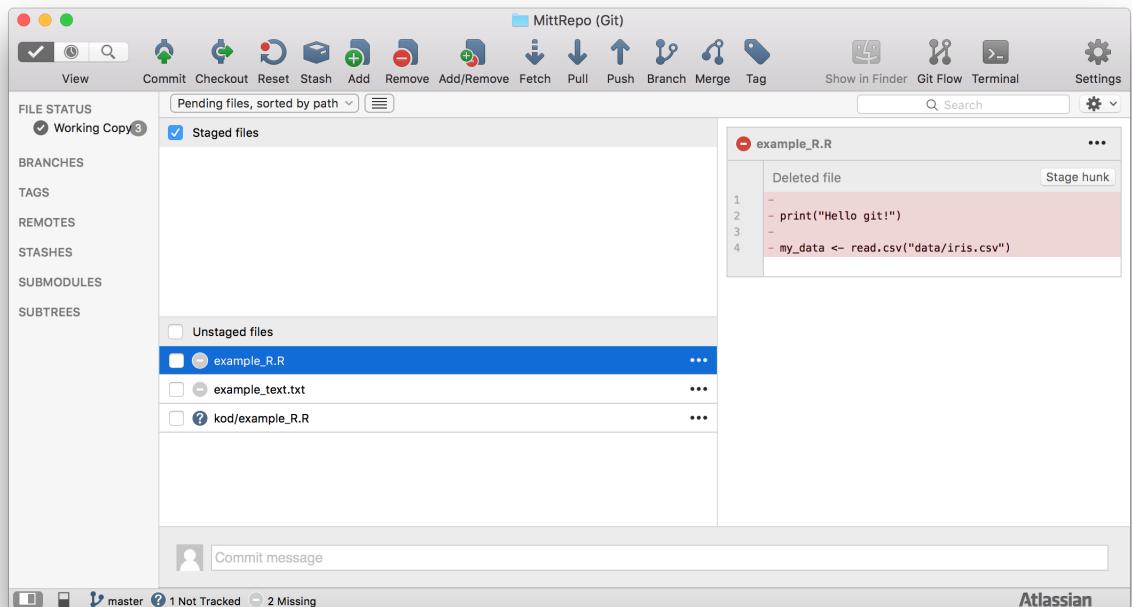
2. Filen har nu dykt upp i git och vi ser i grönt exakt vilket tillägg som gjorts i filen. Detta beror på att git jämför filerna med den senaste commiten och ser nu en skillnad.
3. Precis som tidigare ska vi nu lägga till denna förändring, först markerar vi denna förändring med **Add**. Då kommer vi 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 som en ny commit.
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.



- Nu har vi gjort en förändring i en av våra filer och commitat in denna förändring i vårt repo.

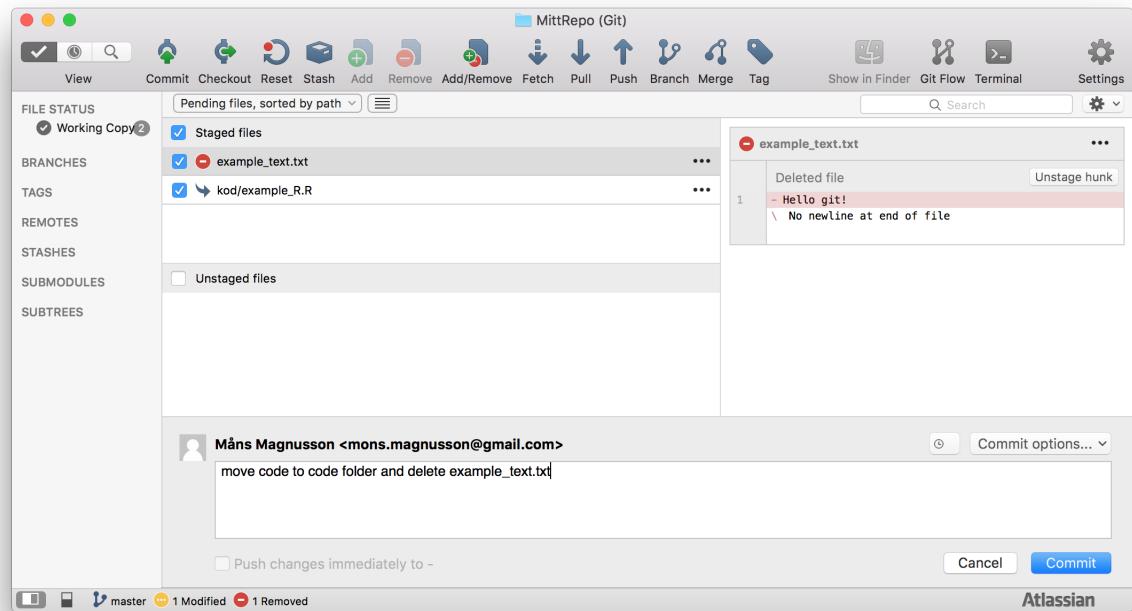
Detta är det vanligaste sättet att hantera förändringar av filer. Vi kan självklart också lägga till nya filer (som tidigare), flytta filer och ta bort filer. Nu ska vi se hur det fungerar.

- 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:



- 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 förändringar till vårt repo med **Commit**.



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

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 committar in ett par rader i taget. Vi gör det genom att markera de rader vi vill committa i den högra vyn. Därefter klickar vi på "Stage lines". På detta sätt kan vi välja ut vilka rader vi vill markera för att committa in i repot.

The screenshot shows the MittRepo (Git) application window. The left sidebar includes 'FILE STATUS' (Working Copy), 'BRANCHES' (master), and other repository navigation options. The main area has tabs for 'Pending files, sorted by path' and 'Staged files'. The 'Staged files' tab is active, showing the file 'kod/example\_R.R' with its content:

```

Hunk 1: Lines 1-8 Stage lines Discard lines
1 1
2 2
3 3
4 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

```

Below the staged file list is a 'Commit message' input field. The status bar at the bottom indicates 'master' and '1 Modified'.

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

This screenshot shows the same MittRepo (Git) interface after the commit. The 'Staged files' tab is still active, and the content of 'kod/example\_R.R' now includes the last two lines of the commit message:

```

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

```

The commit message field remains empty. The status bar at the bottom still shows 'master' and '1 Modified'.

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 när den del vi arbetar med 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 kod som inte fungerar.

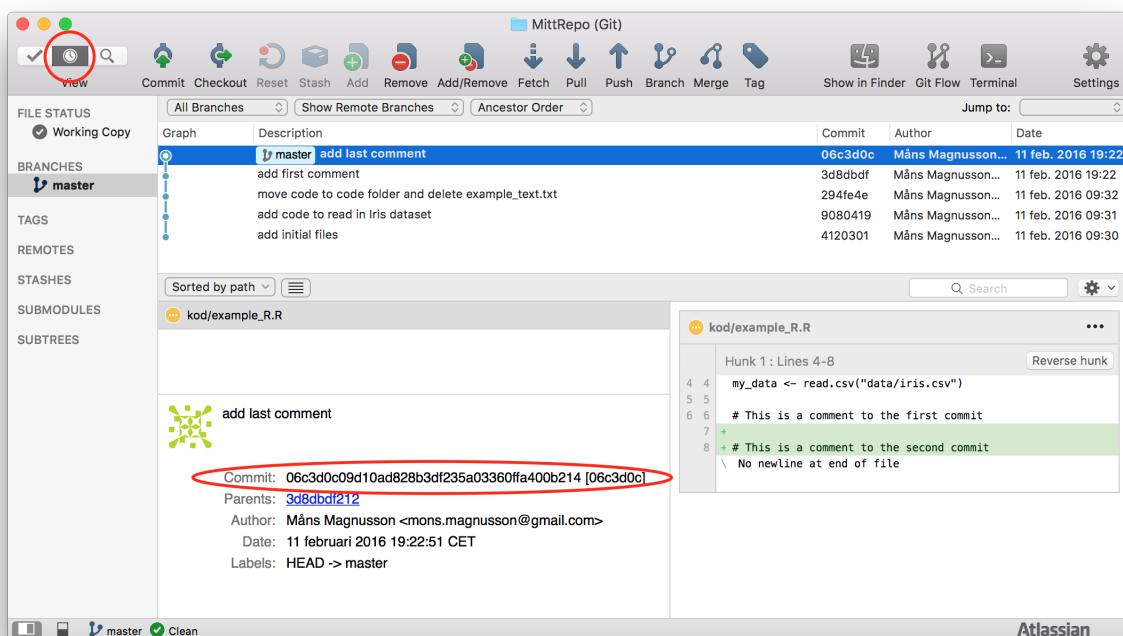
De meddelanden som anges till repektive commit är viktiga. Det är enkelt att efter tag bara skriva ”uppdateringar” som commit message, vilket är intetsägande både för sig själv eller andra. En best practice är att tänka att ett commitmeddelande ska kunna fylla i följande mening:

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 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 eller koden kördes.

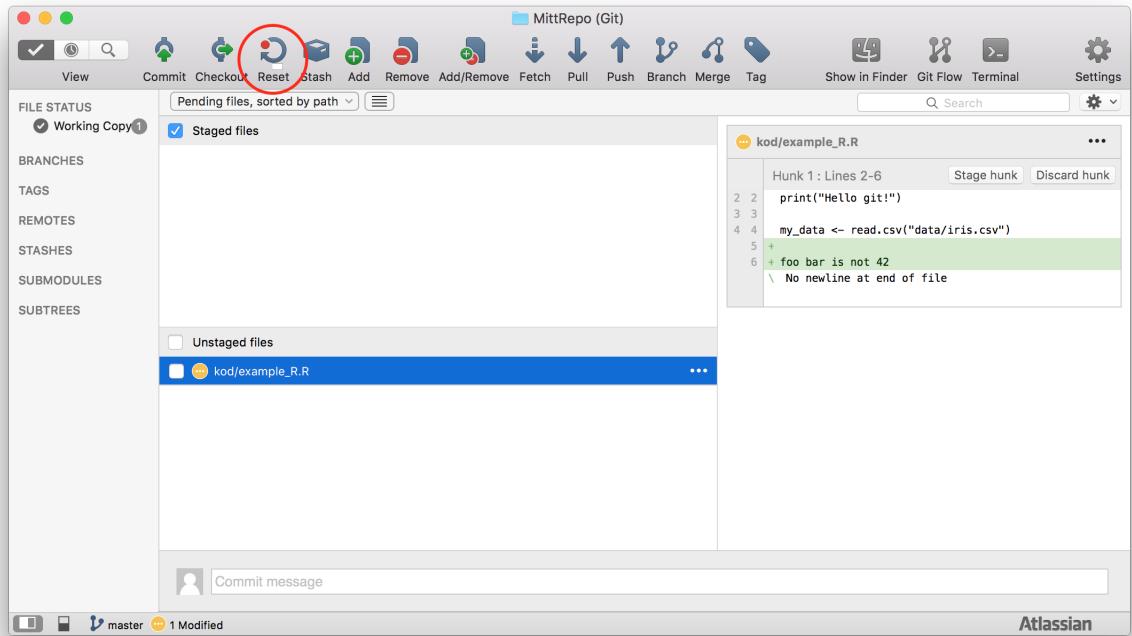
För att se historiken över vilka commits som finns i vårt repo, klicka på klockan i övre vänstra hörnet i SourceTree. Då kan vi se de commits vi gjort och deras commit hash (markerad nedan).



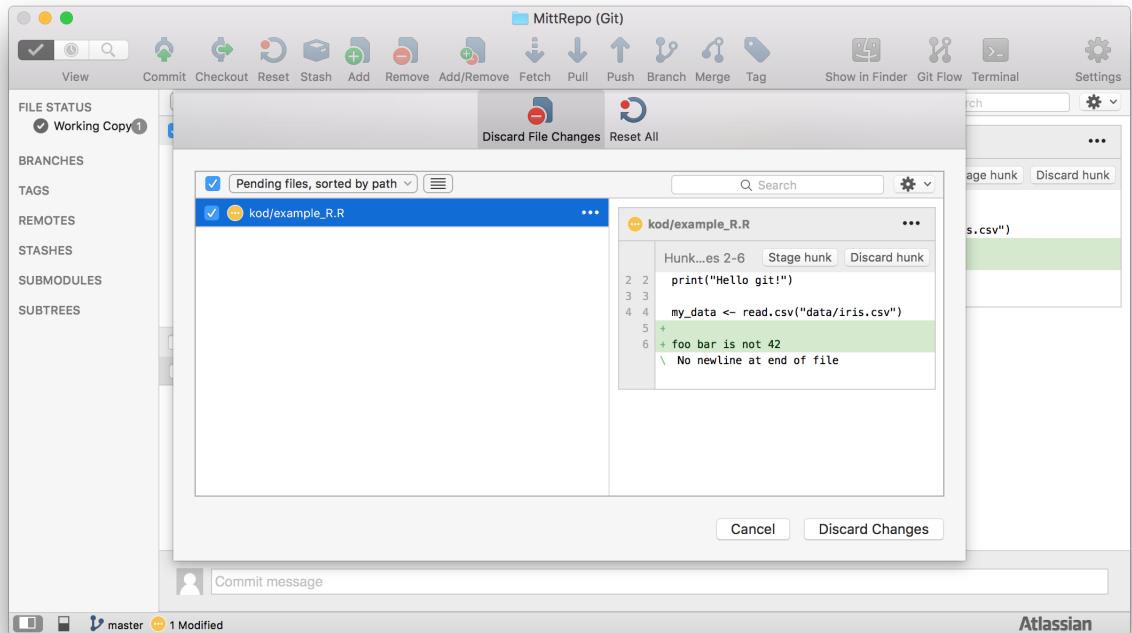
## 2.4 Reset unstaged changes

Att kunna versionshantera kod är bra, men vi vill också kunna ångra förändringar vi gjort men inte commitat.

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 gjorde men inte committade. Titta i filen att den återställts till den senaste commiten.



## 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, 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.

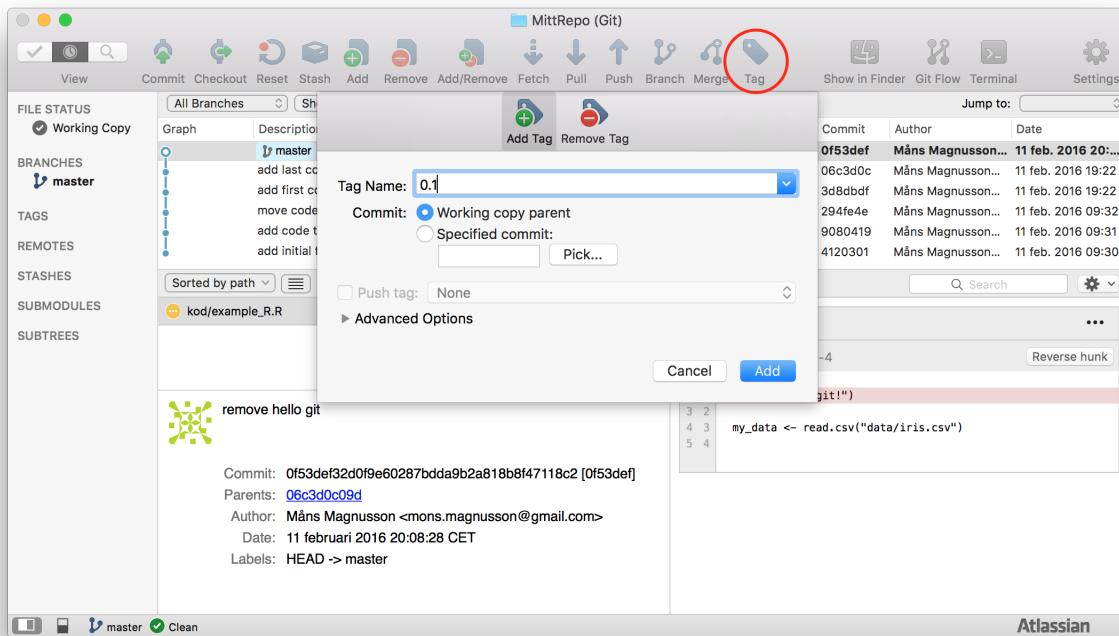
The screenshot shows the Atlassian Git tool interface. On the left, there's a sidebar with sections for FILE STATUS (Working Copy), BRANCHES (master), TAGS, REMOTES, STASHES, SUBMODULES, and SUBTREES. The main area displays a commit graph titled "All Branches". A commit from "master" is selected, showing its description: "remove hello git", followed by several log entries: "add last comment", "add first comment", "move code to code folder and delete example\_text.txt", "add code to read in Iris dataset", and "add initial files". To the right of the graph, a table lists commits with columns for Commit, Author, and Date. Below the graph, a message says "Displaying all changes between 294fe4e79c73b63b57555ccb6f9c1421f07783a6 and 0f53def32d0f9e60287bdda9b2a818b8f47118c2". On the far right, there's a detailed diff view for the file "kod/example\_R.R", specifically for Hunk 1: Lines 1-7. The diff shows the removal of the `print("Hello git!")` line and the addition of code to read an Iris dataset. The code is color-coded in red and green.

Commit	Author	Date
0f53def	Måns Magnusson...	11 feb. 2016 20:...
06c3d0c	Måns Magnusson...	11 feb. 2016 19:22
3d8abdf	Måns Magnusson...	11 feb. 2016 19:22
294fe4e	Måns Magnusson...	11 feb. 2016 09:32
9080419	Måns Magnusson...	11 feb. 2016 09:31
4120301	Måns Magnusson...	11 feb. 2016 09:30

## 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 eller vi gjort en preliminär analys). Då kan vi lägga till en tag till en enskild commit. Detta är ett sätt för att märka upp de större stegeten som gjorts i ett projekt.

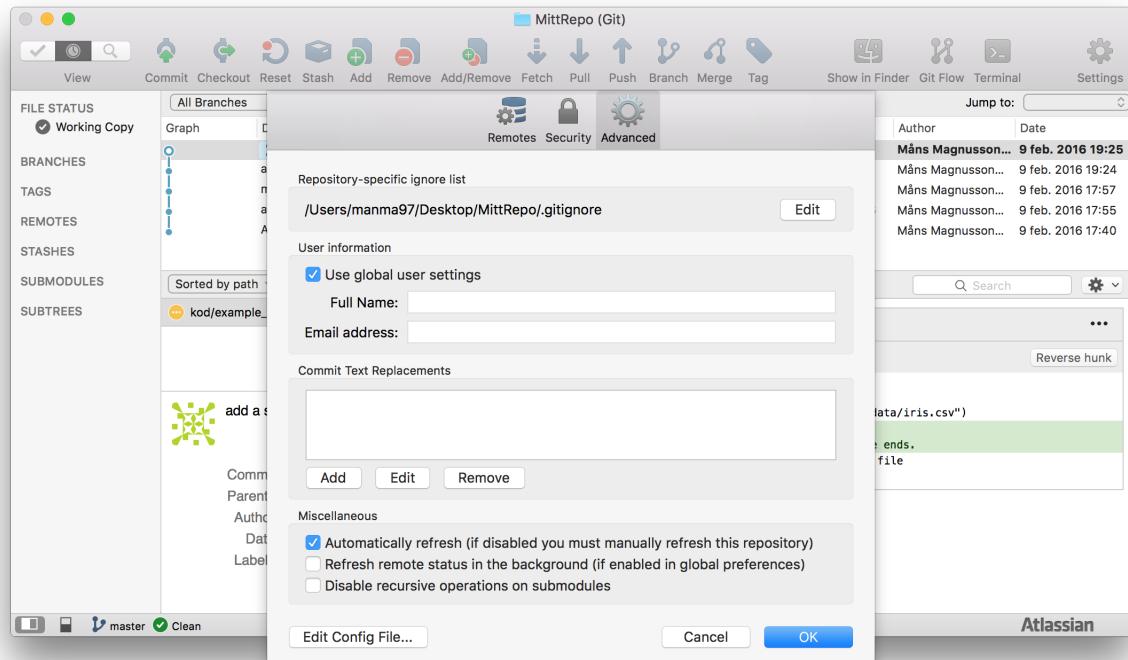
För att lägga till en tag klickar vi på **Tag**. Skriv namnet på taggen och klicka på **Add**.



## 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 i detta repo. Pröva att lägg till en pdf-fil i repot. Git kommer inte visa denna fil.

3. 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 versionhantera/bevaka.
4. Ändrar vi i `.gitignore` kommer den 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

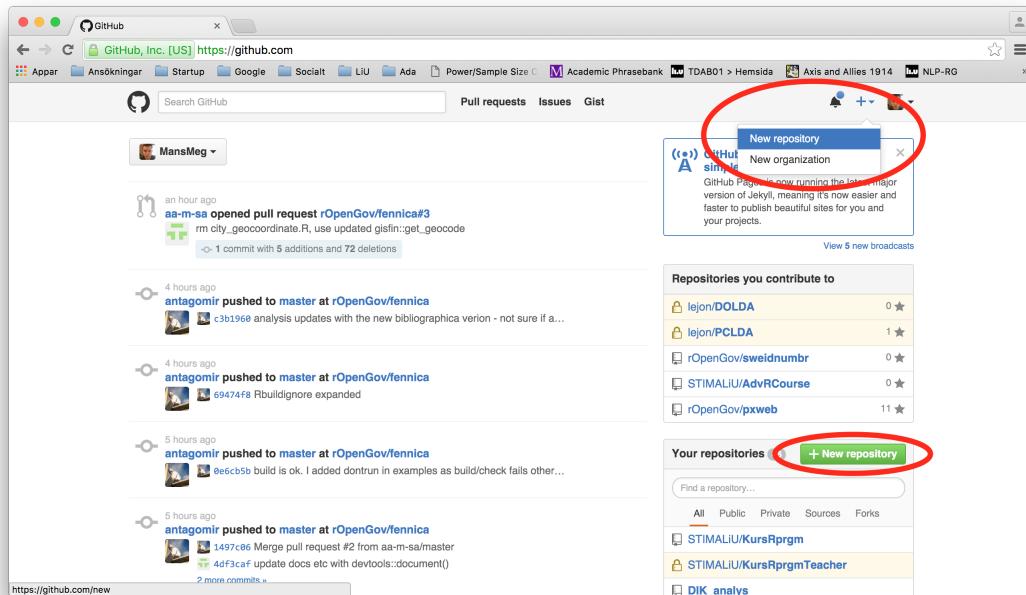
Fram till nu har vi fokuserat på lokal versionshantering. Det fungerar bra om vi jobbar med ett eget projekt och ingen annan har behov av att delta i vårt arbete. Vill vi dock arbeta tillsammans med andra behöver vi ett remote repository (också kallat globalt repo eller centralt repo). Ett remote repo lagrar våra commit på ett centralt plats och medlemmarna i det centrala repot kan lägga till egna commits och hämta ned andras commits. I denna labb kommer vi använda github.com som exempel, men det finns andra tjänster som BitBucket eller GitLab. I praktiken är det ofta bara url-adressen till det globala repot som skiljer sig.

Varje centralt repo har ett antal medlemmar som kan delta och arbeta med repot.

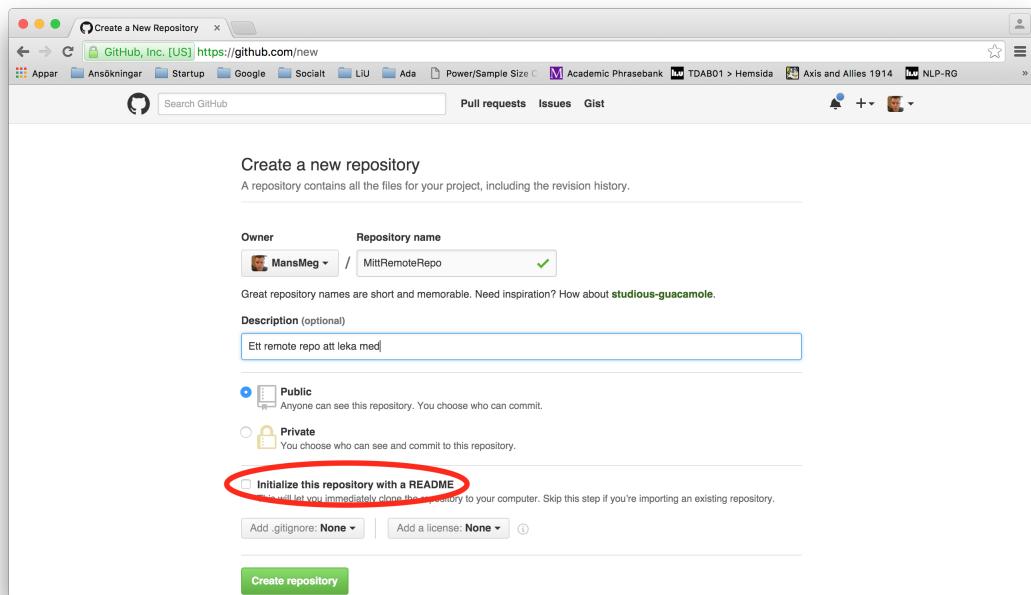
### 3.1 Skapa ett globalt (remote) repository

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

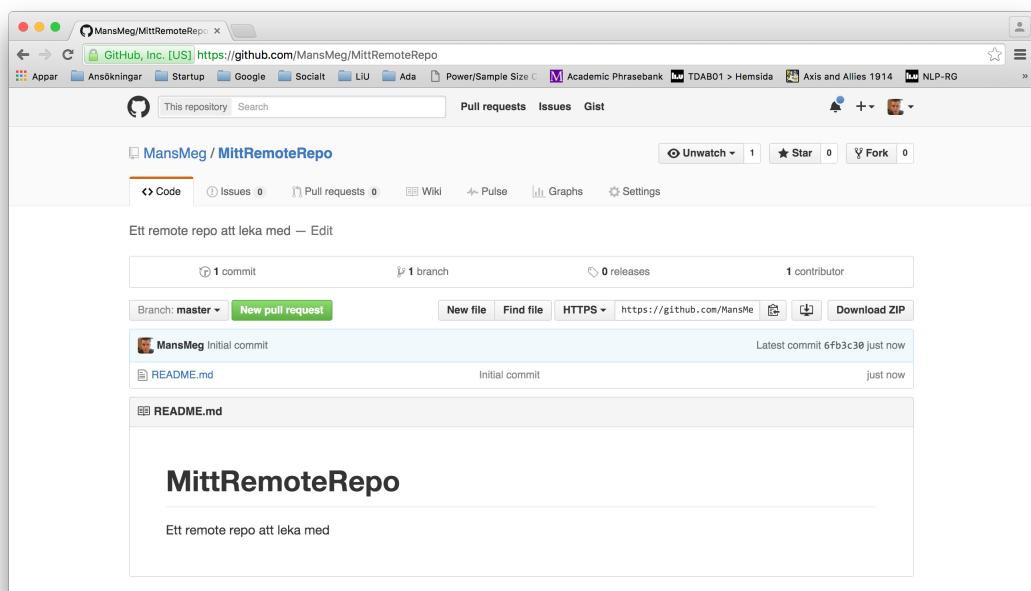
1. Logga in på github.com
2. Skapa ett nytt repo



3. Nästa steg är att ge repot ett namn, beskriva det kort och ange om det ska vara publik (öppet för alla att läsa) eller privat (bara repots medlemmar kan läsa innehållet). Skapa nu ett öppet repo och ange att repot ska skapas med en README. Klicka på **Create repository**.



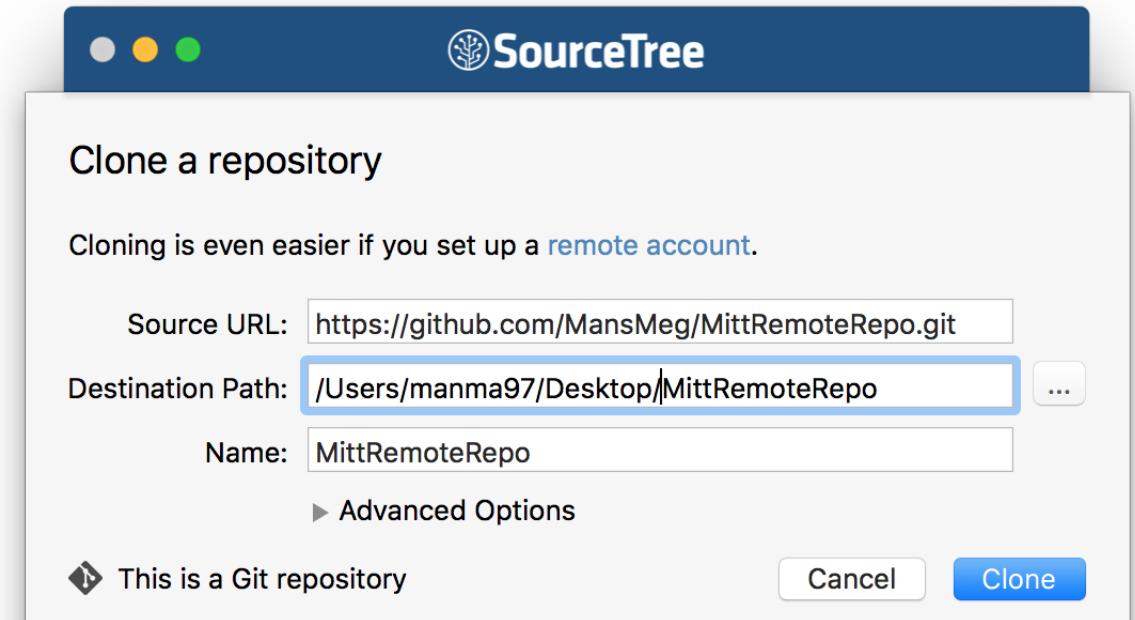
4. Nu har du skapat ett remote repository på github.com. Det borde se ut ungefär såhär:



5. Vi har nu skapat ett remote repo på github. Men då git alltid arbetar lokalt behöver vi klonna detta repo till vår egen dator. Det gör vi i SourceTree genom att välja **+New repository** ⇒ **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 (det lokala) namnet på vårt repo.

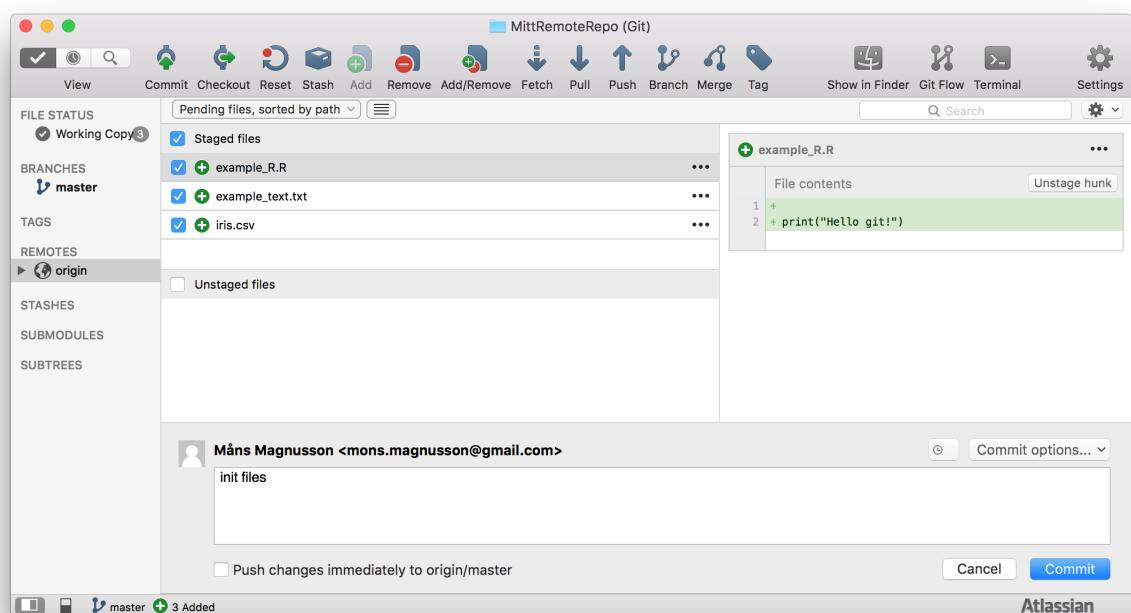


- Nu har vi klonat vårt remote repo till vår egen dator.

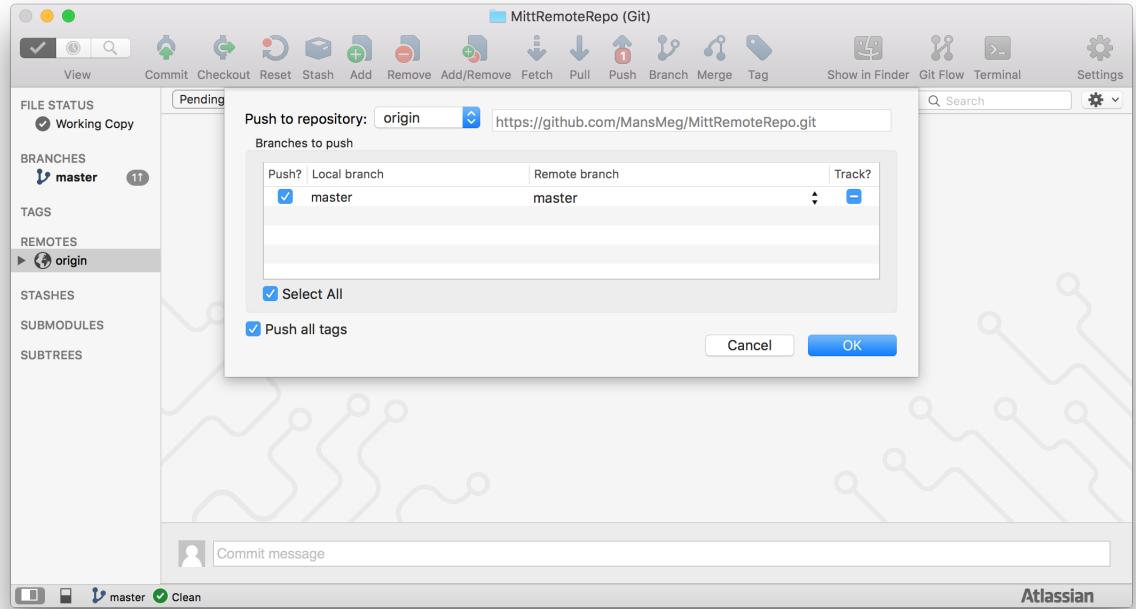
## 3.2 Push

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 sedan av funktionen **Push**.

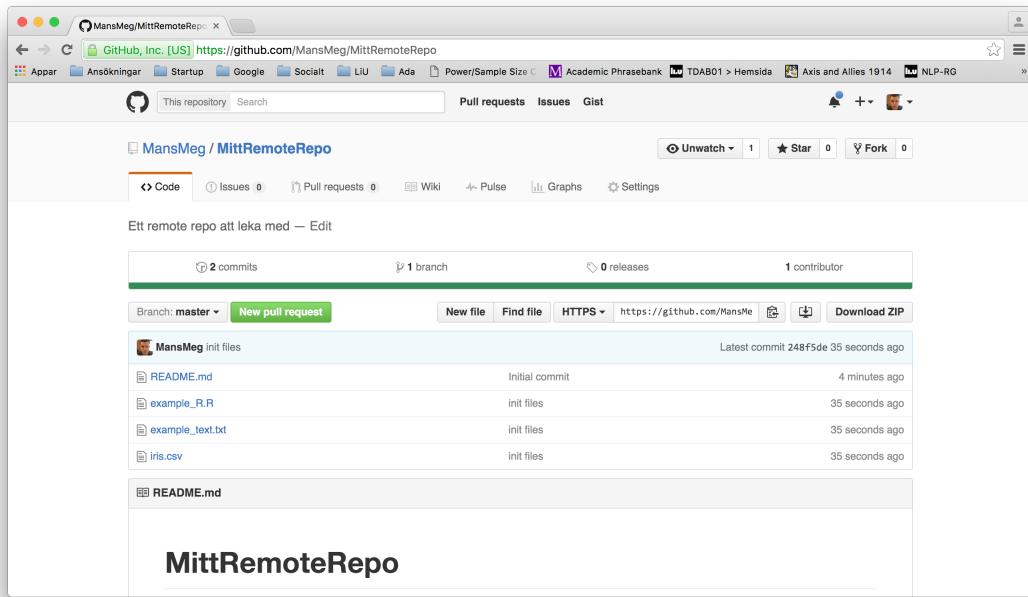
- Markera och committa in samma tre filer som du lade till i det lokala repot vi skapade tidigare. Filerna går att ladda ned från github [[här](#)].



- 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**.



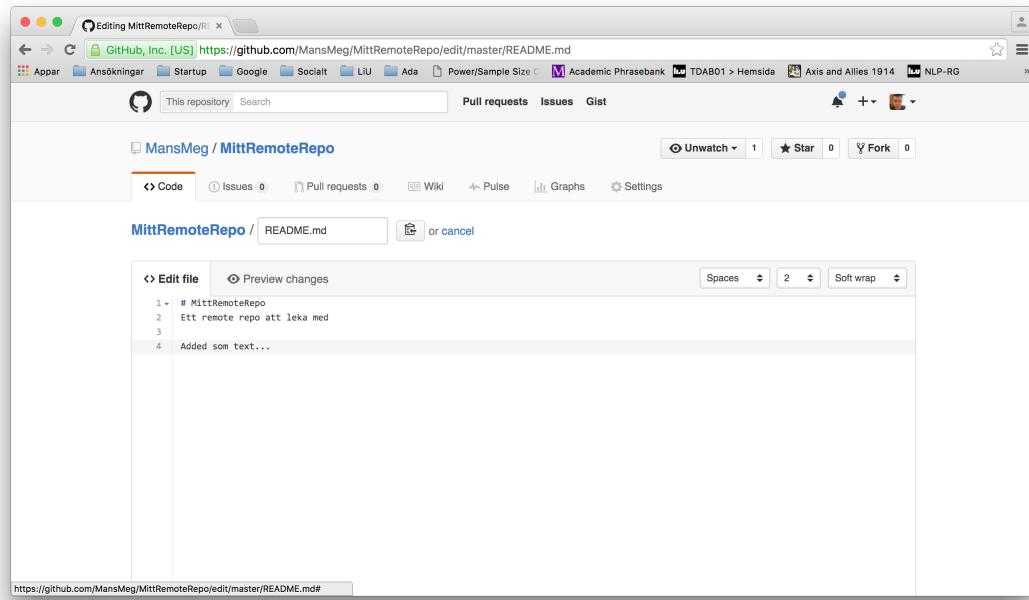
- Nu kan vi titta på github igen. De nya filerna ligger nu i vårt centrala repo på github.



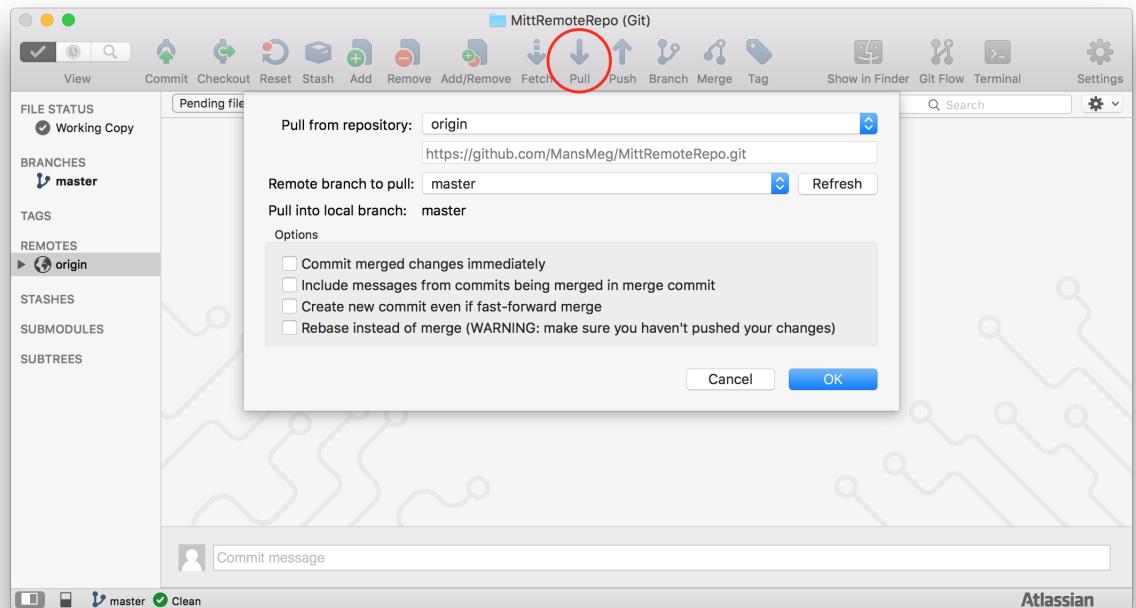
### 3.3 Pull

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

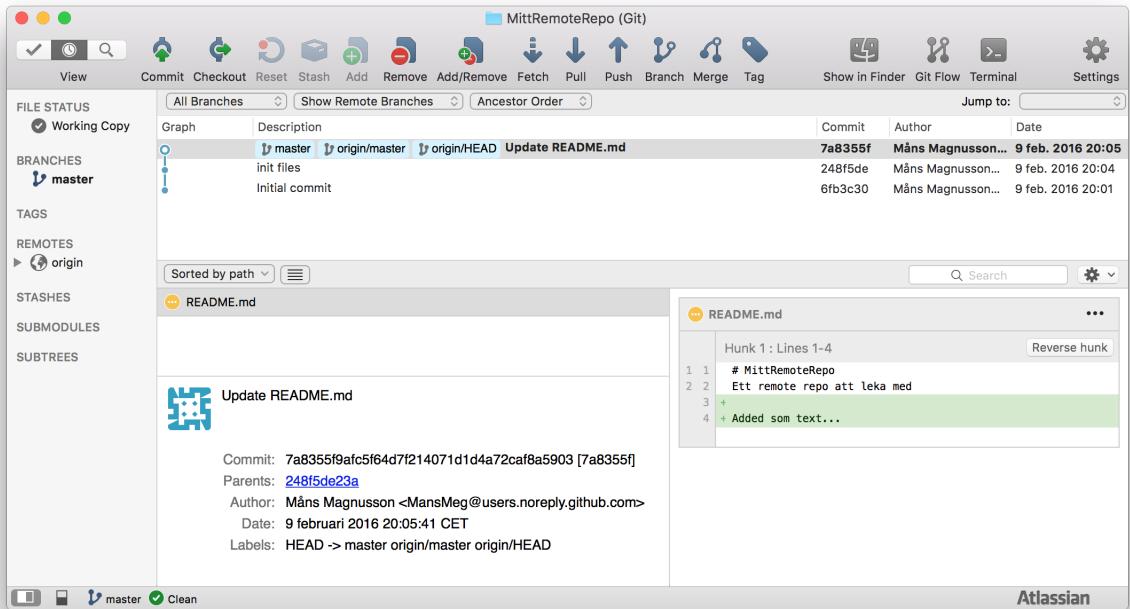
- Låt oss börja med att ändra vår README direkt på github.



- Nu när vi har gjort en förändring direkt på github kan vi låtsas som att detta är en förändring vår kollega gjort och pushat upp till github. För att lägga till denna förändring i vårt lokala repo klickar vi på **Pull**.



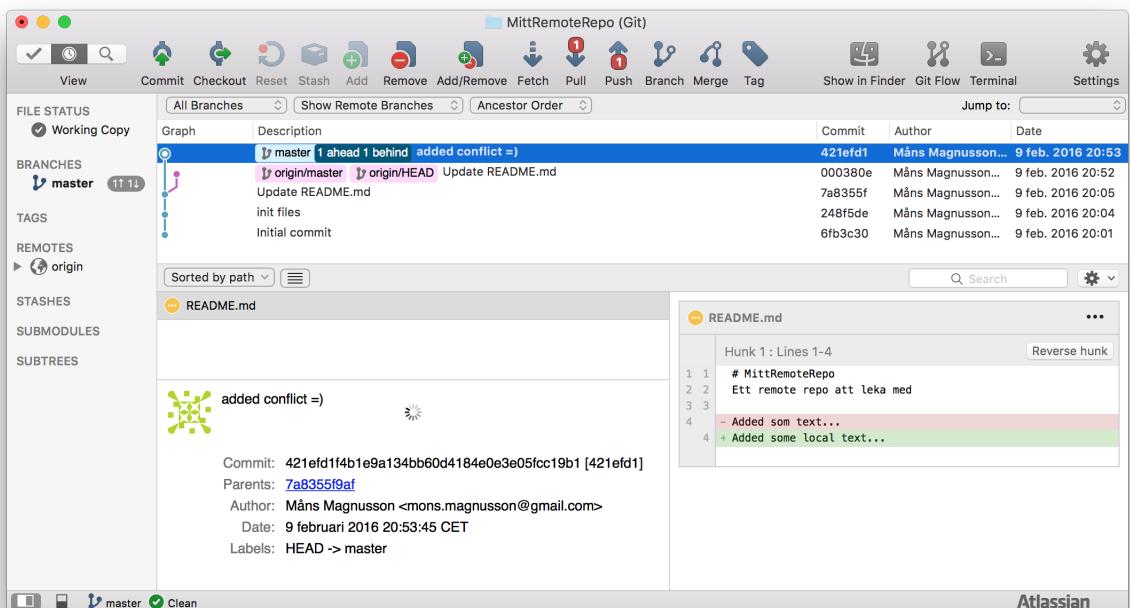
- Nu har vi uppdaterat vårt lokala git repo med de förändringar som låg i det globala repot.



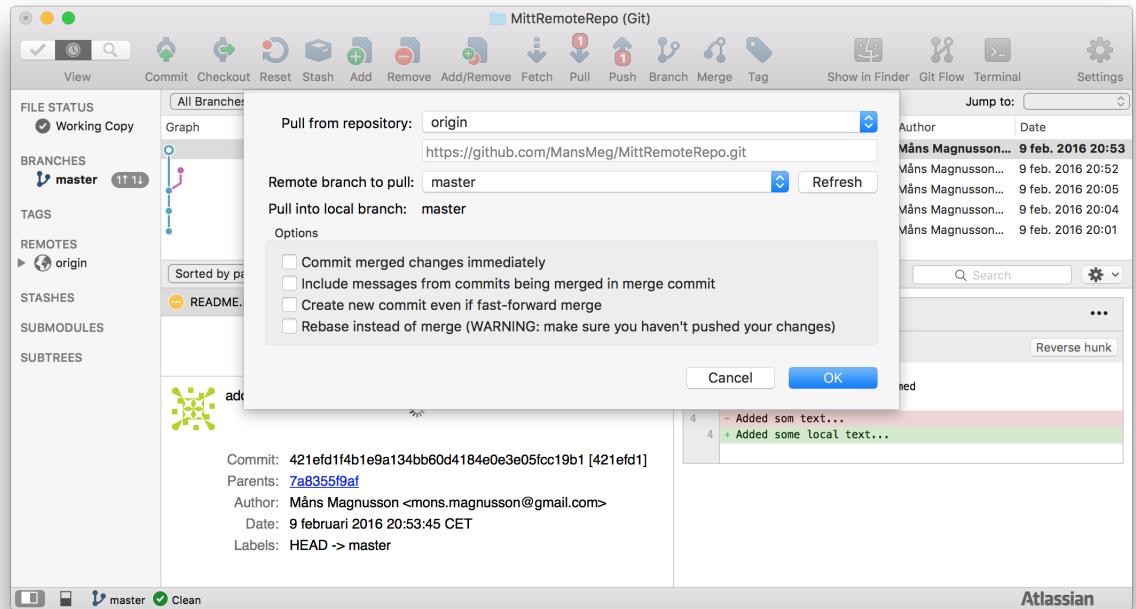
### 3.4 Conflicts

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" i filen 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 har skapat två olika grenar.



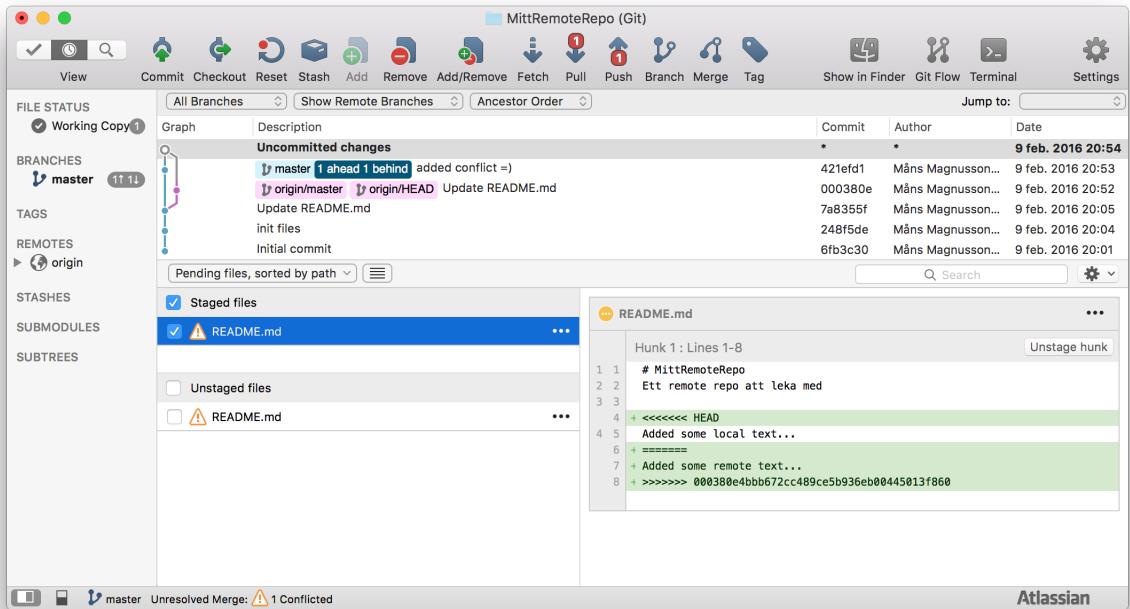
- För att lösa vår konflikt måste vi först använda **Pull** för att dra ned de commits i vårt remote repo som vi saknar lokalt. Vi löser alltid konflikter lokalt. Klicka **OK**.



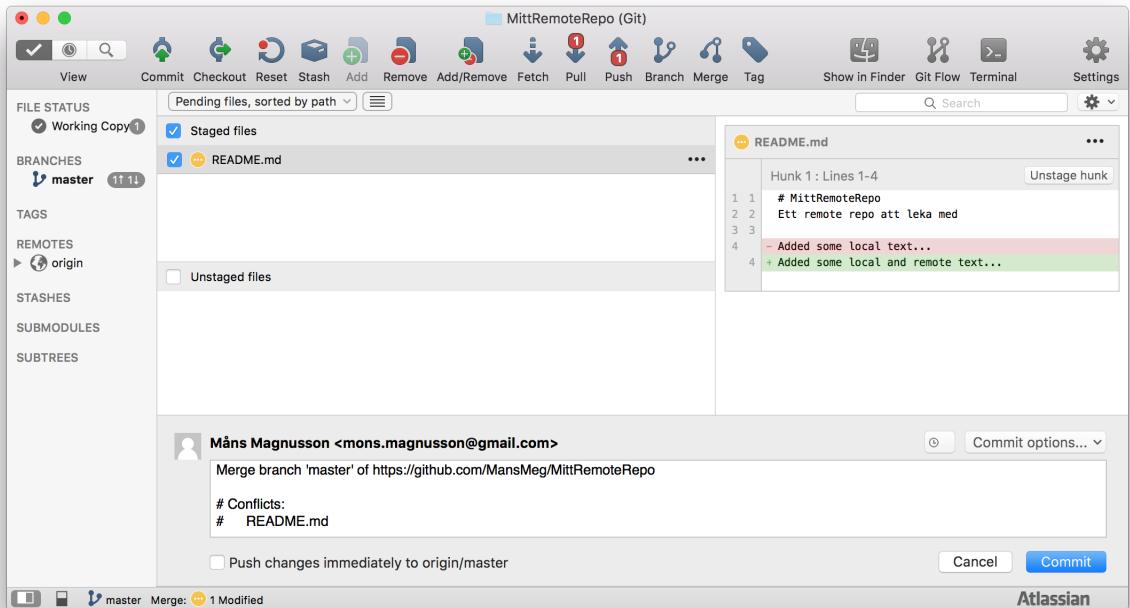
- Nu får vi en varning om att vi har en konflikt.



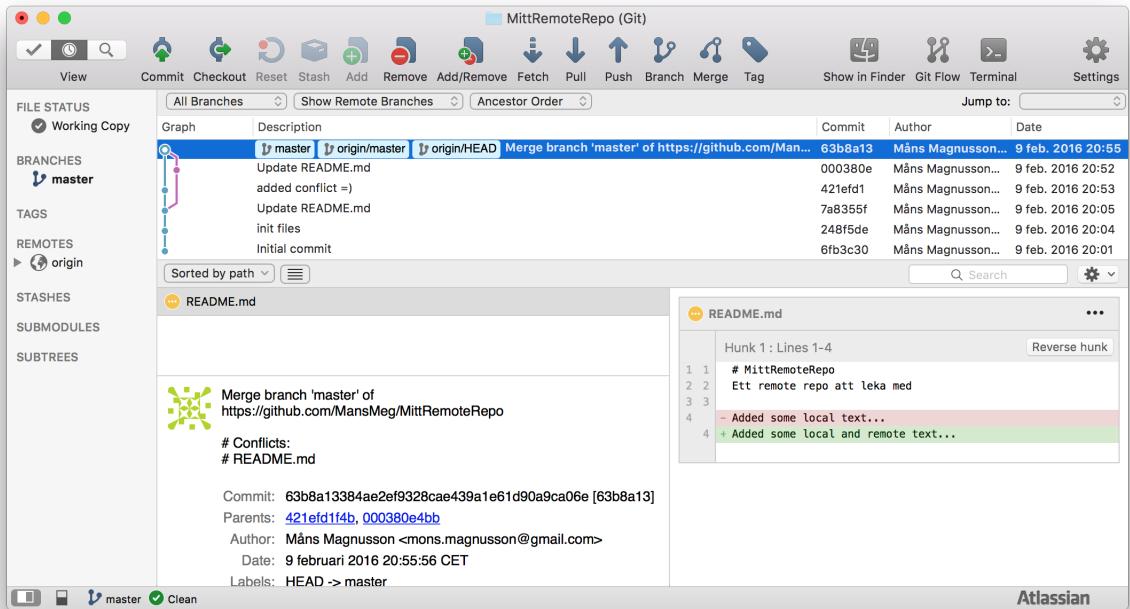
- När det skett en konflikt och vi använder Pull kommer vi få nya filer med skillnaderna mellan filerna som konflikten gäller. För att lösa vår konflikt går vi nu in i vår lokala fil där skillnaden har markerats ut. **HEAD** anger hur vår fil såg ut lokalt och **====** avgränsar den konflikt vi ha med 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 löpande projekt med flera medarbetare.

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**.

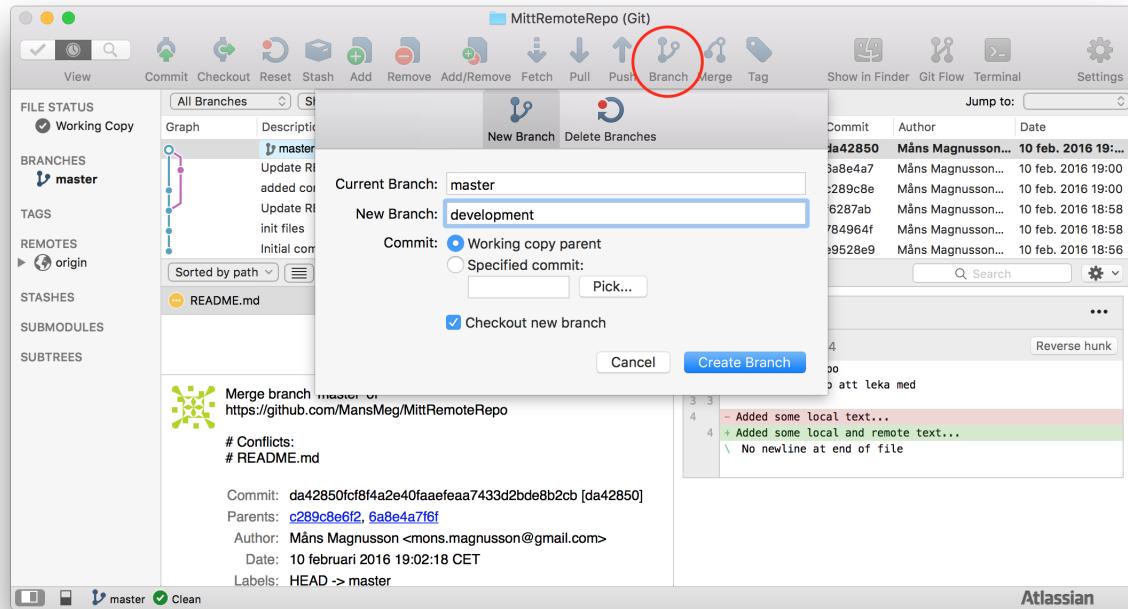
Om vi skapar en ny branch kan vi lägga till nya delar, när vi sedan är nöjda med vårt tillägg kan vi sedan kombinera ihop vår utvecklingsgren med vår huvudgren i repot, vår master branch.

### 4.1 Branch

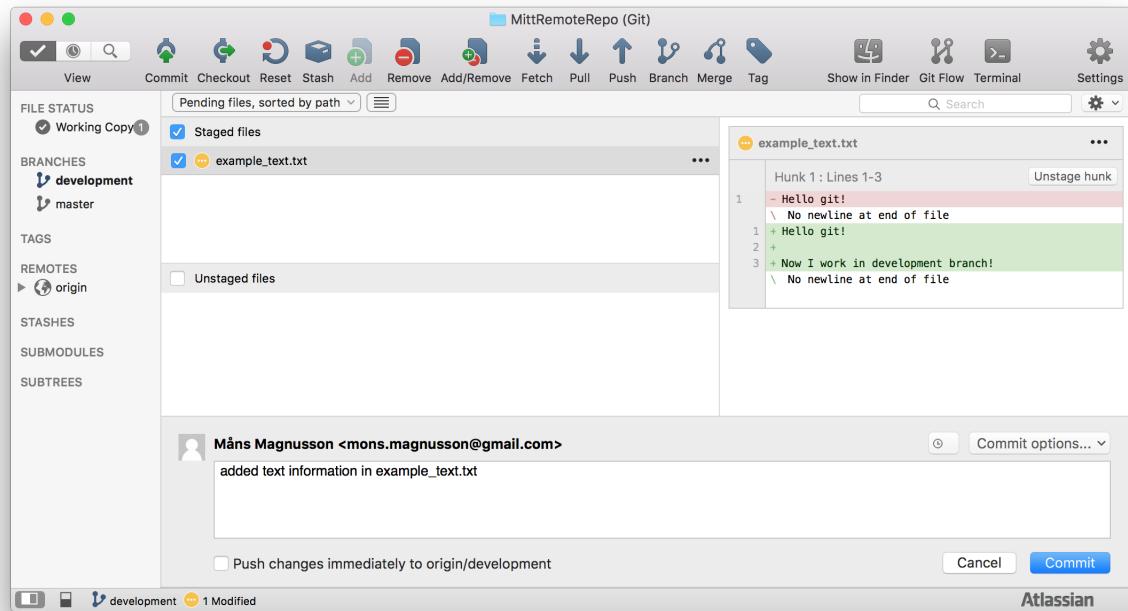
Initialt i vårt repo har vi alltid vår huvudgren master, eller master branch. Det är bra att se master som huvudgrenen i repot. En person som bara ska använda vår kod eller läsa vår rapport ska bara behöva titta i repots master branch. Alla andra grenar är till för de som arbetar i projektet och gör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

För att förstå vad en branch är kan vi tänka oss att vi gör en kopia av vårt repo och arbetar vidare med denna kopia. Denna kopia är en branch.

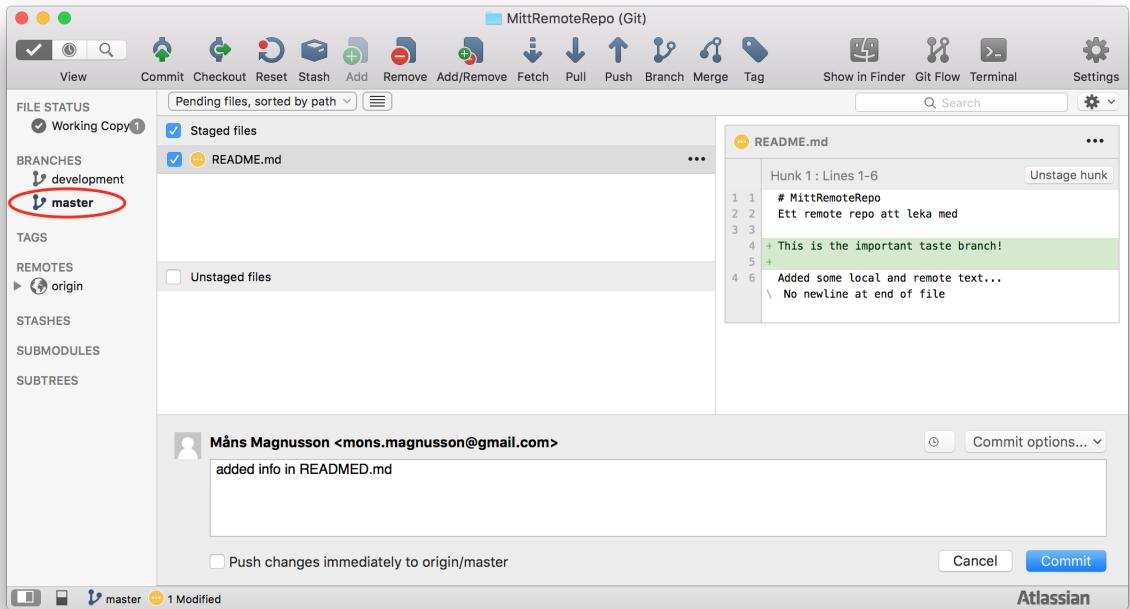
1. För att skapa en ny gren (branch) i vårt repo använder vi funktionen **Branch**. Skapa en branch du kallar "development". För att se vilka branches vi har i vårt repo kan vi använda den vänstra vyn.



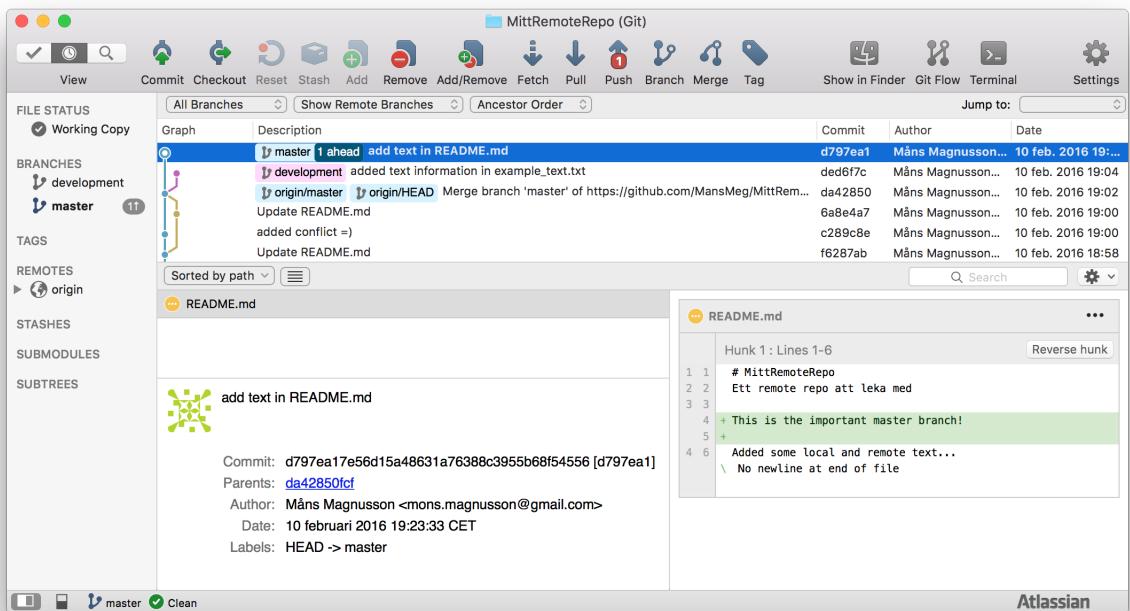
2. Nu har vi skapat en ny gren vi kallar ”development” och den syns i den vänstra vyn. Dubbelklicka på **development** för att byta branch. Ändra nu i vår R-fil och lägg till kommentaren som framgår nedan. Committa in förändringen i development branch. Det borde då se ut på följande sätt.



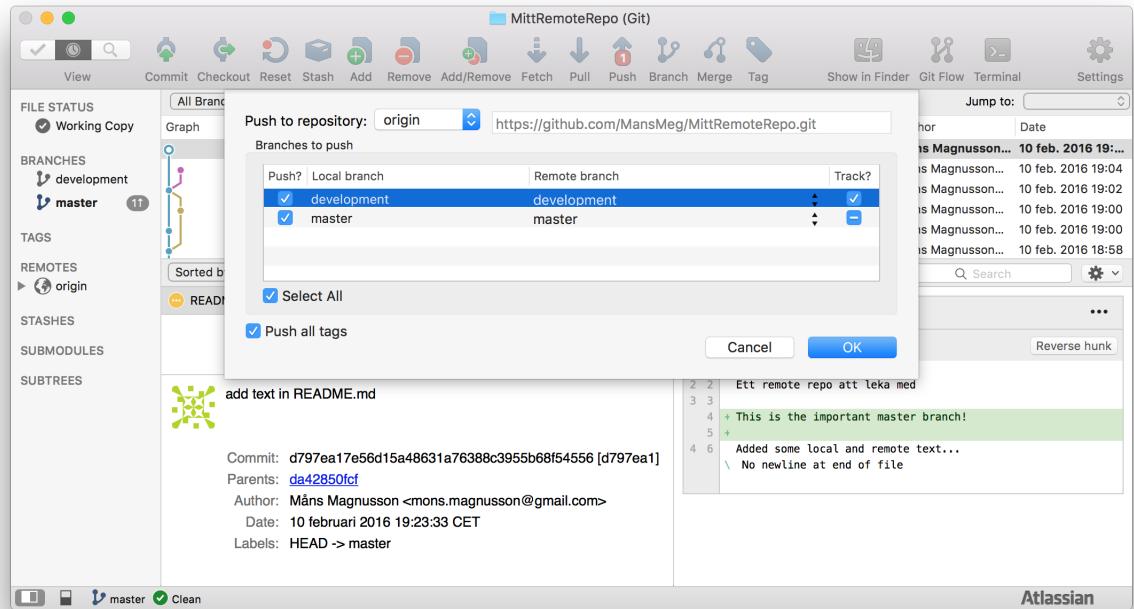
3. Byt nu tillbaka till vår master branch genom att klicka på master i vänstervyn. Committa sedan in följande text i README.md.



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



- Pröva att byta branch från development till master och vice versa. Vad händer med innehållet i filerna README.md och example\_text.txt i vår versionhanterade mapp 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.

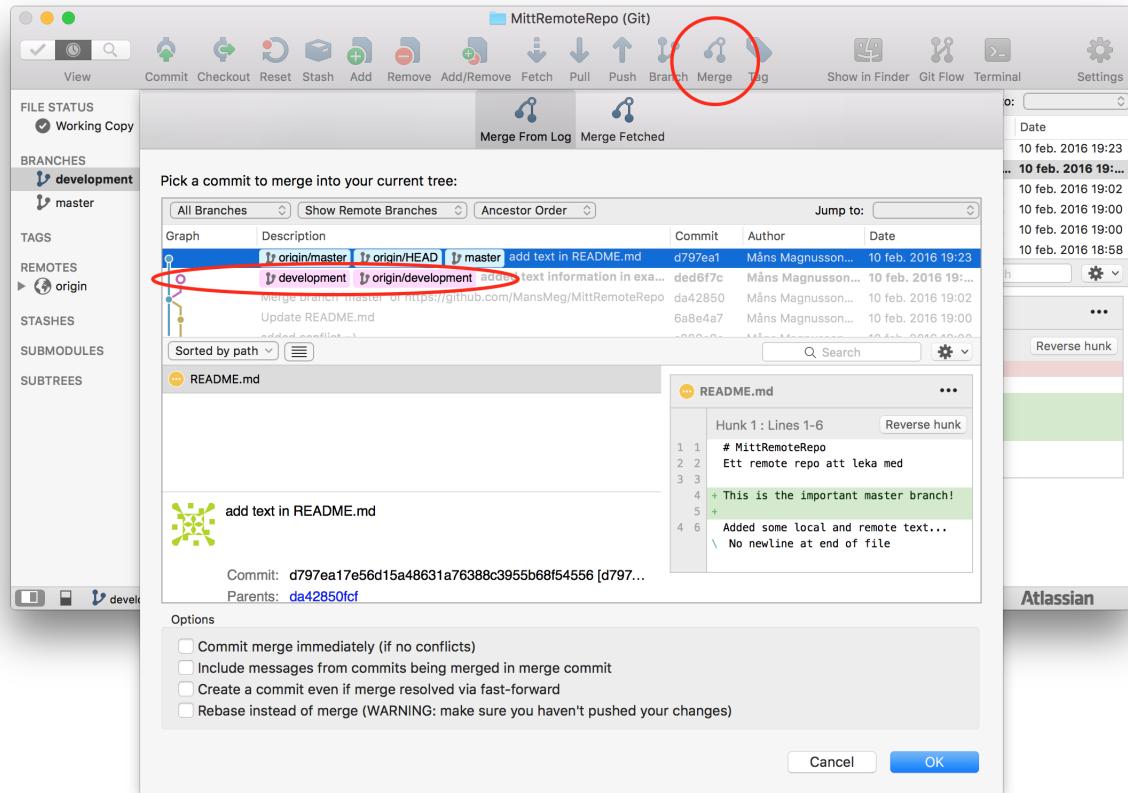


- Nu har vi pushat både master branch och development branch till vårt remote repo. Vi ser det genom att det finns en origin/development- och origin/master-tagg i vår "History".

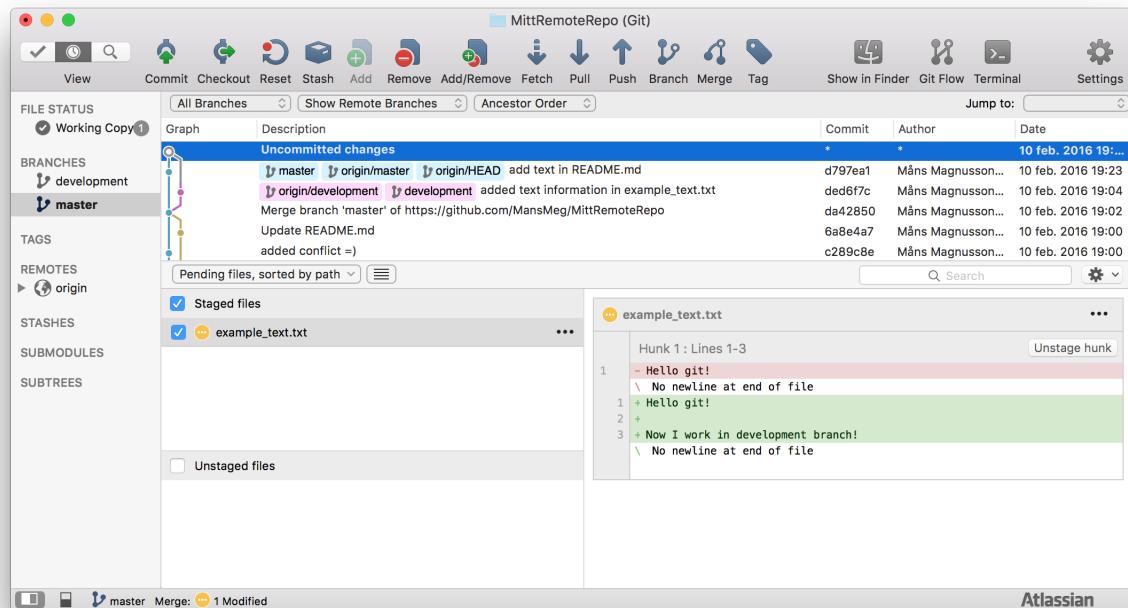
## 4.2 Merge

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

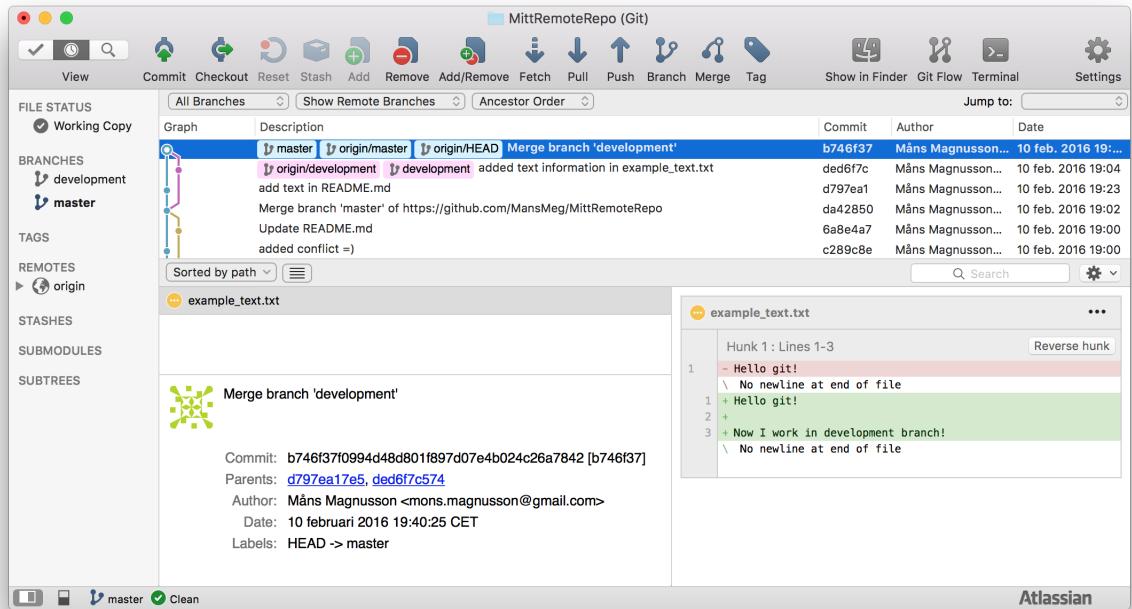
- För att slå ihop en gren behöver vi först aktivera den gren vi vill lägga till en annan gren till. Har vi arbetat i en developmentgren och vill lägga till detta arbete till master ska vi aktivera master.
- Markera grenen master. Klicka nu på **Merge**. Markera development branch (sista commiten) och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



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



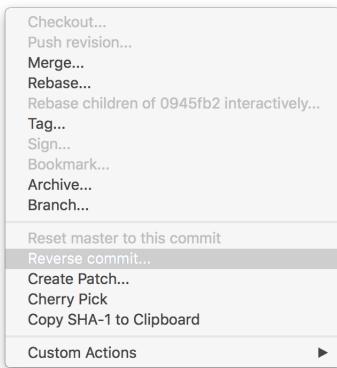
- Commita in dessa förändringar och pusha sedan allt till vårt remote repo. Då borde det se ut som nedan. Vi ser att de två grenarna nu slagits ihop i vår master branch.



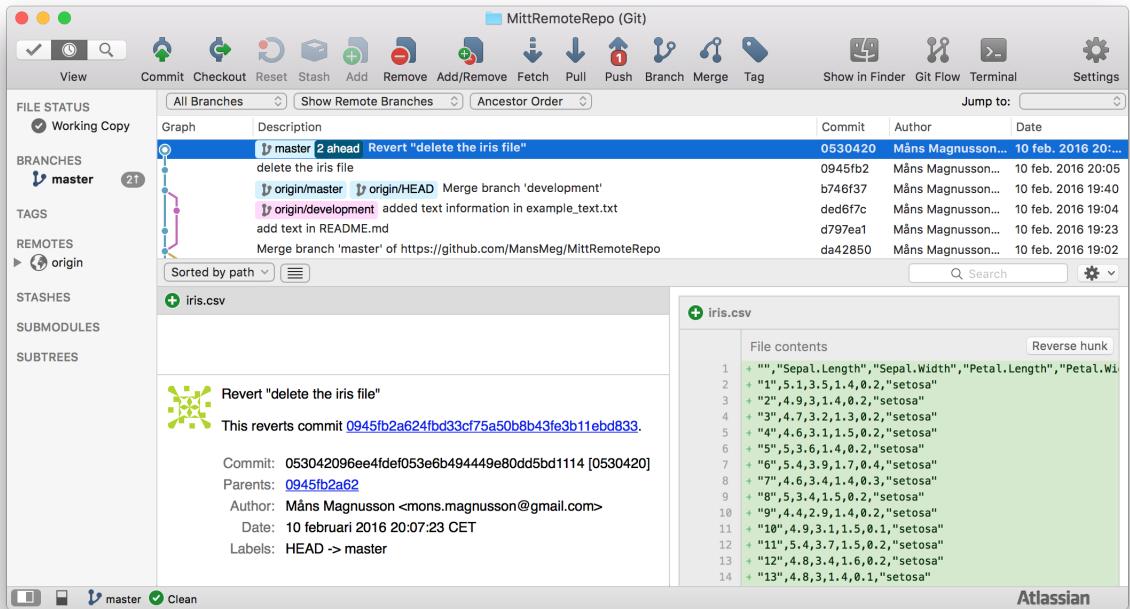
## 5 Korrigera fel i ett repo - Revert och Reset

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 commits (av spårbarhetsskäl) 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. Reset dock tar bort enskilda commits.

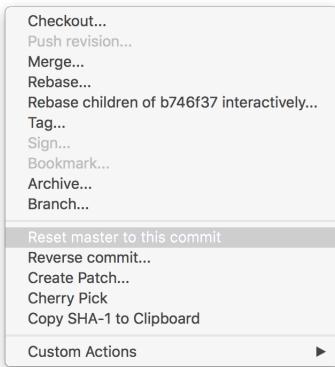
1. Börja med att radera filen `iris.csv` från MittRemoteRepo och committa denna förändring.
2. Som ett första steg ska vi nu återställa detta fel med Revert. Högerklicka på det commit vi vill ta bort/återställa och välj **Reverse commit...**



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. Var det några förändringar i filerna som gjors kommer dessa förändringar dyka upp som ”uncommitted changes”.

