As a data scientist, you will likely be asked one day to automate your analysis and port your models to production environments. When that happens you cross the blurry line between data science and software engineering, and become a machine learning engineer. I'd like to share a few tips on how to make that transition as successful as possible.

Let's first discuss testing, and let's assume without loss of generality that you develop your machine learning application in R. Just like any other software system, your application needs to be thoroughly tested before being deployed. But how do you ensure that your application will perform as expected when dealing with real data? If you've structured your application as an R package you already know that you can write unit tests that will run automatically when you check your package; but how do you test the application as a whole?

Testing an application as whole locally, known as integration testing, should normally not be part of your suite of unit tests: integration tests are frequently much slower than unit tests, and would break your regular build-test workflow. (Besides, if you forgive some pedantry, an integration test is not a unit test by definition since it tests the whole system.)

Nevertheless, integration tests serve a useful purpose in your workflow. They're often the last test you'll run locally before releasing your application to more 'serious' testing. But that doesn't mean you should write them last; in this article I'm going to argue that writing them first, before even writing a line of code, tends to produce a better, clearer, and more testable design.

Let's illustrate with an example. You work for a travel agency whose CEO would like a system that predicts the survival probability of its customers, should their cruise ship hit an iceberg. You are told that we have extensive data on shipwreck survivors since 1912, when the Titanic sank. You are told that your application should apply machine learning on a dataset that has the same structure as the famous Titanic survivor data set, but you cannot access the entire set directly.

You are relatively free to choose the output format of your application, because it will be picked up downstream by other teams. You decide to start with a CSV file for now.

Well, where to begin? From an earlier analysis (ideally documented in a vignette, a notebook, or another report) you know that a random forest is the best model for this problem. So it would make sense to verify whether indeed your application trains a random forest from the training data. How would we know that? The simplest way to save the trained model along with the other output, and check that it is indeed a random forest, trained with the right number of training samples. And this is definitely something which we know how to test.

I'm going to assume you have the basic structure of an R package in place, such as the one created by default by RStudio when you start a new project. We'll call it `app`:

```
app
├── DESCRIPTION
├── NAMESPACE
├── R
│   └── hello.R
├── app.Rproj
└── man
    └── hello.Rd
```

Besides the `testthat` package, which has become the standard library for writing unit tests in R, I recommend using `testhis`, which facilitates creating and running integration tests:

```
usethis::use_testthat()
testthis::use_integration_tests()
```

This sets up the basic structure for running unit tests and other integration tests:

```
app
├── DESCRIPTION
```

```
├── NAMESPACE
├── R
│   └── hello.R
├── app.Rproj
├── man
│   └── hello.Rd
└── tests
    ├── testthat
    │   └── integration_tests
    └── testthat.R
```

Any test you create under `tests/testhat/integration_tests` will be run when you call `testthis::test_integration()`, but not when running your regular unit tests.

I like to begin with a failing test to make sure that everything is setup correctly. Let's go ahead and write our end-to-end test, calling it `tests/testthat/integrations_test-end_to_end.R`:

```
# integrations_test-end_to_end.R
test_that("this test fails", {
  expect_true(FALSE)
})
```

As expected, we can now run the "integration" test and see it fail:

```
> testthis::test_integration()
Loading app
✓ |  OK F W S | Context
✗ |   0 1     | test_end-to-end
──────────────────────────────────────────────────

test-test_end-to-end.R:2: failure: this test fails
FALSE isn't true.

──────────────────────────────────────────────────

══ Results ═════════════════════════════════════
OK:       0
Failed:   1
Warnings: 0
Skipped:  0
```

Good. Let's now write test code as if a model had been serialized, and check its number of training samples.

```
# integrations_test-end_to_end.R
test_that("a model is output with the right number of training samples", {
  app::main()
})
```

Hold it right there. This test will, of course, not even run because the `main()` function doesn't exist. Let's create `R/main.R` with the following contents before we continue:

```
# main.R
main <- function() {}
```

The integration test now runs, so we can complete it. But before we can continue, we will need some training data, which will be the Titanic survival dataset. We don't want `main()` to read the data from file, but from an interface that is as similar as possible to the actual environment.

We've been told that the application will read from a PostgreSQL relational database. We could certainly have our test code set up a local PostgreSQL instance, and load it with the training dataset; this will sometimes be the right strategy, but in this case I think it is overkill. Instead, we will exploit the fact that most database systems can be abstracted away through ODBC drivers. R code that talks to PostgreSQL through

its ODBC driver should work just as well against any other database system. In particular, we'll use the in-memory SQLite database for integration testing.

So our integration test code will look like this, in pseudo-code:

```
# integrations_test-end_to_end.R
test_that("a model is output with the right number of training samples", {


  app::main()


})
```

Well, let's do it. Create the `tests/testthat/testdata-raw` folder (or just call `testthis::use_testdata_raw()`), and copy there the `train.csv` dataset downloaded from https://www.kaggle.com/c/titanic/data, renamed to `titanic.csv`.

We'll proceed with turning our pseudo-code into real code, refactoring as we go along:

```
# integrations_test-end_to_end.R
test_that("a model is output with the right number of training samples", {
  # Load Titanic training set
  titanic <- readr::read_csv("../testdata-raw/titanic.csv")
  # Copy it to SQLite
  con <- odbc::dbConnect(
    odbc::odbc(),
    driver = "SQLite",
    database = "file::memory:?cache=shared"
  )
  dplyr::copy_to(con, titanic, temporary = FALSE)
  # Call main()
  app::main()
  # Read serialized model
  model <- readRDS("output/model.rds")
})
```

Notice the `file::memory:?cache=shared` URI used to instantiate the in-memory SQLite database. If you simply pass the usual `:memory:` filename you won't be able to connect to it from more than one place in your program; but we want the test code to populate the same database that will be accessed by the main application, so we must pass this special `file::memory:?chache=shared` URI.

Notice also the `temporary = FALSE` argument to `dplyr::copy_to()`. This is required for the table to be accessible from another connection than the current one.

Calling `testthis::test_integration()` should now fail:

```
> testthis::test_integration()
Loading app
✓ |  OK F W S | Context
✗ |   0 1 1   | test_end-to-end
──────────────────────────────────────────────────

test-test_end-to-end.R:14: warning: a model is output with the right number of
training samples
cannot open compressed file 'output/model.rds', probable reason 'No such file or
directory'

test-test_end-to-end.R:14: error: a model is output with the right number of
training samples
```

```
cannot open the connection
1: readRDS("output/model.rds") at /Users/dlindelof/Work/app/tests/testthat
/integration_tests/test-test_end-to-end.R:14
2: gzfile(file, "rb")
──────────────────────────────────────────────────

══ Results ═════════════════════════════════════
OK:       0
Failed:   1
Warnings: 1
Skipped:  0
```

It fails indeed, so we stop writing test code at this point and fix it. It fails because no model is serialized yet; we don't want to overengineer at this point and fix that by instantiating a fake model and serializing it:

```
# main.R
main <- function() {
  model <- lm(~ 0)
  if (!dir.exists("output")) dir.create("output")
  saveRDS(model, "output/model.rds")
}
```

The tests pass now, so we keep on going:

```
# integrations_test-end_to_end.R
test_that("a model is output with the right number of training samples", {
  # Load Titanic training set
  titanic <- readr::read_csv("../testdata-raw/titanic.csv")
  # Copy it to SQLite
  con <- odbc::dbConnect(
    odbc::odbc(),
    driver = "SQLite",
    database = "file::memory:?cache=shared"
  )
  dplyr::copy_to(con, titanic, temporary = FALSE)
  # Call main()
  main()
  # Read serialized model
  model <- readRDS("output/model.rds")
  # Verify number of training samples
  expect_equal(length(model$fitted.values), nrow(titanic)) # (1)
})
# (1) check if the model has been trained with the right number of samples
```

There are several issues with that test, but it is complete (for now) and, as expected, fails:

```
> testthis::test_integration()
Loading app
✓ |  OK F W S | Context
✗ |   0 1     | test_end-to-end
──────────────────────────────────────────────────

test-test_end-to-end.R:16: failure: a model is output with the right number of
training samples
length(model$fitted.values) not equal to nrow(titanic).
1/1 mismatches
[1] 0 - 891 == -891
──────────────────────────────────────────────────

══ Results ═════════════════════════════════════
```

```
OK:       0
Failed:   1
Warnings: 0
Skipped:  0
```

To pass the test, the `main()` function needs to open an ODBC connection to SQLite, read the training dataset, train (some) model with it, and serialize the model. But we run into a problem: how does our application know where to find the database it should connect to? And how does it know which driver to load? A reasonable solution is to use the `config` package with a configuration file. When we run integration tests we'll use the `test` settings, and when we run in production we'll use the `default` settings.

So here's a first attempt at `config.yml`:

```
# config.yml
test:
  driver: "SQLite"
  database: "file::memory:?cache=shared"

default:
  driver: "PostgreSQL"
  database: "TBD"
```

We should now be able to read the `titanic` dataset from `main()`:

```
# main.R
main <- function(env = "default") {
  connection_params <- config::get(config = env)
  con <- do.call(odbc::dbConnect, c(odbc::odbc(), connection_params))
  train <- dplyr::tbl(con, "titanic") %>% dplyr::collect()
  model <- lm(~ 0, data = train)
  if (!dir.exists("output")) dir.create("output")
  saveRDS(model, "output/model.rds")
}
```

Running the tests, we see that the dataset gets read without errors by `main()`, but the number of training samples in the model remains 0:

```
> testthis::test_integration()
Loading app
✓ |  OK F W S | Context
✗ |   0 1     | test_end-to-end [0.1 s]
──────────────────────────────────────────────────

test-test_end-to-end.R:16: failure: a model is output with the right number of
training samples
length(model$fitted.values) not equal to nrow(titanic).
1/1 mismatches
[1] 0 - 891 == -891
──────────────────────────────────────────────────

══ Results ═══════════════════════════════════════
Duration: 0.1 s

OK:       0
Failed:   1
Warnings: 0
Skipped:  0
```

That is simply because we have not specified any target variable in our call to `lm()`. Let's fix that:

```
# main.R
```

```
main <- function(env = "default") {
  connection_params <- config::get(config = env)
  con <- do.call(odbc::dbConnect, c(odbc::odbc(), connection_params))
  train <- dplyr::tbl(con, "titanic") %>% dplyr::collect()
  model <- lm(Survived ~ 0, data = train)
  if (!dir.exists("output")) dir.create("output")
  saveRDS(model, "output/model.rds")
}
```

And the tests now pass:

```
> testthis::test_integration()
Loading app
✓ |  OK F W S | Context
✓ |   1       | test_end-to-end

== Results =====================================
OK:       1
Failed:   0
Warnings: 0
Skipped:  0
```

Let's now recap what we have achieved. We have an integration test case that verifies that a linear model (or **any** model, really, provided it provides a `fitted.values` attribute) gets trained by our application, using a dataset read using an ODBC connection we provide. But more importantly, we have written an automated test that exercises *the whole application*, and will keep on doing so as we implement the missing pieces.

It doesn't feel like we have accomplished much; after all, our model is certainly not the best one for the classification task we have at hand, and we don't even train it in a sensible manner. But consider this: before you began the project, chances are the customer (or the boss) didn't really know what they wanted. Now you are in a position to hold the following conversation:

you: We have decided that the first user story should prove that the application, when run, trains a model on the Titanic dataset. This is what this first integration test demonstrates.
boss: How good is that model?
you: Its accuracy is 62%, if that's what you mean.
boss: Really? That's pretty impressive for a first shot. How did you achieve that?
you: It predicts that everybody dies. And 62% of passengers in the training dataset did indeed die.
boss: (shifts uncomfortably) hm I see well I guess we should think of alternative metrics. We'd rather avoid false positives, even at the cost of some accuracy. Can you do something about it?

So by getting a woefully incomplete, but working, application out so soon you've been able to help your stakeholders understand what really matters to them. Better, yet, you are even in a position to write the next item on your backlog as a precise user story:

```
As a customer, I want to know if I'm going to die on this cruise
but a death prediction had better be right 90% of the time
```

Time to write a new integration test and our first unit tests.