Why I abandoned online data courses for project-based learning

medium.freecodecamp.org/why-i-abandoned-online-data-courses-for-project-based-learning-17e350fd9347

There's no time like the present to teach yourself data science, analytics, or engineering. A quick search on Udemy shows over 2,000 results for courses about "data." People have even compiled their own Master's degree programs in data science comprised entirely of free online courses.

In my experience as a self-taught data engineer, taking dozens of massive open online courses (MOOCs) is not the best approach. It didn't work for me.

I didn't have hours every night and weekend to spend studying. The lectures didn't feel practical enough to launch me from a non-technical field to a job in data. The projects didn't usually align with my interests, and course-after-course, I quickly ran out of motivation to continue.

I eventually figured out that project-based, self-guided learning absolutely beats taking online courses — as long as you keep a few guiding principles in mind.

Choose a project that's interesting to you and requires skills you'd like to learn. As you build each unit of the project, learn the necessary skills to complete that unit. Project-based learning is more efficient, more practical, and more fun.

Today, I'm leading analytics for Milk Bar, a popular bakery and e-commerce desserts company. I acquired most of the skills I use today by working on projects that I cared about, not by slogging through another Coursera clone.

Here's why project-based learning is so effective.

Finding motivation is harder than finding information

The fundamental advantage of project-based learning is that motivation, not information, is usually the limiting factor when it comes to teaching yourself a new career.

The Internet is full of tutorials and MOOCs. But if you're a busy professional learning data on nights and weekends, you probably don't have the time or interest to sit through a 45-minute lecture filled with content that you may not remember or even need.

The hardest part of learning skills like programming, data warehouse modeling, machine learning, or statistics is finding the drive to keep going when you come up against confusing, technical material.

The key is to find a project that you're so excited to bring to reality that you don't mind pushing through the obstacles along the way.

Here are four attributes that I find make a project motivating. Maximize one or more of these and you'll have a much better chance of actually finishing your project.

- **Utility:** "This would make my life easier and save me time."
- **Passion:** "This would solve a problem I deeply care about."
- **Curiosity:** "This dataset really intrigues me and I want to explore it more."
- **Competition:** "I want to win this prize and beat other competitors."

Weaknesses of project-based learning vs. online courses

Project-based learning isn't all roses. There are a few important weaknesses to be aware of with this style of learning.

Online courses generally do some things better. However, I've highlighted a guiding principle that you can adopt to combat the weakness. I'll spend the rest of this guide describing those guiding principles in greater detail.

- It's harder to be structured. With project-based learning, you are both the student and the curriculum designer. It's easy to start in the middle of the material by accident, or choose a project that's too hard for your current skill level. **Guiding principle: Think before you type.**
- It's harder to be comprehensive.Project-based learning is inherently specific—you are learning only the skills needed to build your project. You might miss out on other must-learn skills for your field.

- It’s harder to understand what clean code looks like. It’s easy to pick up bad habits or style because you don’t have someone checking your work and you don’t have many reference points for clean code. **Guiding principle: Write code that reads like a novel.**
- It’s harder to get peer review.Self-directed learning can be a bit lonely and you miss out on the positive feedback loop that comes from having your work evaluated and praised. You also don’t have an organization or expert in the field vouching for your skills or your work unless you share your work directly. **Guiding principle: Build your portfolio and share your work.**

Think before you type

Most entry-level developers are not great at planning. The best data people are systems thinkers. They know that designing the architecture of a solution is much harder and more important than implementing it with code.

No matter how small your project, develop the discipline (and it is a discipline, it requires a lot of self restraint when you’re starting work on something fun) of planning it properly before you dive in and hack away.

This doesn’t have to be a time-consuming process, but it must be an intentional and thoughtful one.

Divide your project into units and sketch it

When I start work on a new project, I like to divide the work up into modular, conceptual units. Think about the **boundaries of functionality** in your project.

In the example of a typical machine learning model, you might choose data collection, data transformation, model training, model evaluation, and deployment as your project units.

If you’re just starting out in data, choose project units that are more granular.

For example, you might choose to divide up a web scraping project into downloading the file, extracting relevant information, and scheduling your scraper to run on an automated cadence.

Describing each unit of work and detailing its dependencies will help prepare you for working on engineering teams. Work is often divided up using user stories or GitHub issues.

Finally, sketch out each product unit using your preferred medium. This can be as simple as a bulleted list in a text file. Or, it could be as complex as a diagram with dependencies and attributes.

Whatever you do, make it visual, so you can see how everything fits together.

Map out the skills you'll develop by completing each unit

Since our goal is to learn new things, take some extra time and map out the skills you'll need to learn in order to build each unit.

It's okay if you don't get this perfectly right in the planning phase. The point is to have a directional sense of how challenging each unit will be to build and shape your learning process later.

Developing is inherently an iterative process. You'll probably find that you underestimate how much you need to learn. It's hard to know what you don't know.

Nevertheless, I find this step helps me as I start to learn and have questions like:

- "Where should I look for tutorials or examples?"
- "Am I even Googling the right thing? If I knew what this was *actually* called, I'd have a better idea of how to search for it…"
- "What packages are typically used for this kind of thing? What should I become familiar with and read documentation for?"

Using our example of building a web scraper, we might come up with the following:

- **Download the file:** read documentation for the `requests` package, learn how to use Chrome Developer Tools to inspect HTML code, learn about HTTP response codes
- **Extract the relevant information:** learn how to load a CSV into Python, learn about Pandas dataframes and how to filter them, learn how to create and save files in Python
- **Schedule the scraper to run on an automated cadence:** learn how to schedule a background process with cron

If you're like me, you may find that you get to the end of this process and realize that your project is too ambitious for your current skill level. If that happens, pick a single unit of the project and turn it into its own project.

Maybe building a neural network from scratch and deploying it on Google Cloud Platform is unrealistic. But, learning how to deploy a pre-built model might be doable.

Scope your project so that you're considerably challenged (you'll learn the fastest this way). But not so challenging that you're overwhelmed with difficulty and lose motivation.

I'm always impressed that self-taught developers can usually hack together a solution that works with very little instruction or guidance. I believe self-taught developers are some of the most resourceful workers you can hire.

What's less impressive is when their code is inefficient, inelegant, or impossible to read. I attribute this to two issues: an over-reliance on StackOverflow, and a lack of knowledge about best practices.

As a developer who is starting out and teaching yourself, you have a weakness. You simply haven't done enough work to know how a solution probably "should" be built. You may not even know how a variable "should" be named or how your code "should" be organized.

That’s not entirely a bad thing — it might make you more open to creative approaches that more seasoned developers wouldn’t consider. However, you need to be vigilant about learning how the industry approaches common problems so you can develop an intuition for good practices.

I think about Googling with a purpose (or more broadly, searching for help) as a funnel.

You start with high-level, descriptive content like blog articles or tutorials that help you understand preferred architecture and best practices. Next, you identify packages or tools you’ll need to use and devour the documentation.

Finally, as you build your project units, you consult sites like StackOverflow for solving tactical problems.

Start with articles from Medium, blogs, or tutorials

When I first started cooking, I was *super* dependent on recipes. When I made a dish, I followed each step blindly without understanding how that step fit into a larger understanding of cooking techniques and rationales.

Now that I’ve become a more experienced cook, I’ll read a handful of recipes for a dish from sources that I trust and note their similarities (i.e. best practices for making that dish) and differences. Then I’ll make the dish, generally adhering to the best practices but modifying the auxiliary steps or ingredients based on my own tastes.

High-level blog articles and tutorials are really similar to recipes in that sense.

The first thing you should do when trying to learn how to develop a data project is to scour the Internet for examples from reputable sources. Most of the problems data professionals are tackling these days are not that unique. It’s likely that your project isn’t that unique either (and that’s actually a good thing).

Read all the examples and note the similarities and differences in approach. Once you have a good sense of best practice for solving your problem, you can starting choosing your tools to build with.

Identify relevant packages and tools and read their documentation

After reading a few tutorials and blog posts, you should have a good sense of the tooling (packages, models, or algorithms) that people generally use to solve problems like yours.

Before you spend too much time implementing those tools, read their documentation in detail. Even the parts you think might be less relevant to your project. After all, the point of all this is to learn, right?

I'm a strong advocate for spending serious time reading documentation. It happens over and over again — I go to a documentation page trying to solve a specific issue and come away with a handful of useful classes, functions, or design patterns that I didn't know existed that dramatically improve other parts of my code.

Reading documentation helps us understand how to use packages as their author intended. Even if the documentation is lengthy or technical, it's worth your time to struggle through.

Why? Chances are, the author of the package is probably much more well-versed in the problem you're trying to solve than you are. I think of the hours I've spent assembling IKEA furniture. No one in their right mind would go to Quora to ask how to build their desk when they have the instructions from the manufacturer right in front of them!

Reading documentation will also help you understand how industry-grade code is written. Pay attention to the way functionality is distributed among classes, how arguments are structured, or the way information is loaded and stored. Don't be afraid to poke around in the package's source code as well.

Lean on Q&A sites to help you solve tactical problems

Finally, once you've exhausted high-level best practices and read relevant documentation cover-to-cover, it's time to ask for help. This is where

Q&A sites like StackOverflow shine, and where it’s totally okay to depend on them.

If you have a tactical issue (such as “I don’t understand why I’m getting this error”, or “I’m not getting the result I would expect from this”), there’s no better place to get answers.

Write code that reads like a novel

When you’re working on a project in isolation, all that matters is that you understand how it works. As soon as you join an engineering team, you’ll realize that perspective isn’t sustainable.

You’ll find that a lot of the actions you take and decisions you make when writing code, documenting it, and deploying it are actually to help *other* people understand what you’re doing.

Eventually, you’ll want to share your code with a potential employer, so write your code as if you were about to submit it with a job application.

Try to view your code as if you were reading it for the first time and take time to document or improve the aspects that aren’t immediately clear.

Don’t settle for “it runs”

I don’t care much for extensive comments in code — they’re often redundant and annoying to maintain as code changes. Instead, try to write code that is self-evident.

I have a piece of paper above my desk with giant letters that read **“Newlines are cheap, brain time is expensive.”**

Don’t worry about optimizing for perfect code. Instead, optimize for speedy comprehension of your code by others.

If that means adding another variable, breaking out some functionality into a separate function, or using more descriptive names, do it.

Every time you complete a development task, ask yourself if you could’ve done it in a more elegant or clear way. Your future colleagues will thank you.

Follow style principles for your language of choice

I often find that junior developers don't pay enough attention to how their code appears. It may not seem like it initially, but variable name casing, indentation, docstrings, and whitespace are all helpful when it comes to writing clean code.

Good style makes your code more natural to read for more senior developers who are accustomed to industry-standard style.

Take a look at the relevant style guide for your language (check out PEP 8 for Python or Google's R style guide). Put their main recommendations into practice.

You can even install linters for your text editor that enforce these style principles and help you clean up your code.

Build your portfolio and share your work

If a junior developer asked me for a job, and shared a GitHub portfolio and Medium profile with interesting projects that were thoroughly documented, I wouldn't care much about the contents of their resume. Analytics is a new field and the most in-demand ability is the ability to do the job.

My project portfolio got me my first data job, even though I flunked some of the technical interview questions pretty embarrassingly in person. At the end of the day, I was hired because I had evidence that I could write clean code and solve technical problems.

This is especially crucial if you don't come from a technical field. If you can prove that you've learned the skills required for the position you want, you'll have a much better chance of getting the job.

Write a blog post about your project

Writing is a crucial skill for any aspiring data professional to develop. As a data professional, you're constantly being asked to describe your process, diagram your architecture, explain your model, or interpret your results.

Writing a blog post on Medium or on a personal blog is a fantastic way to practice translating technical details into user-friendly language and build your personal brand.

In your post, describe your motivation for taking on the project, any best practices you identified along the way, and the details of your implementation.

Even if no one ever reads your post, writing has the added bonus of making your project feel truly *finished*. Write that blog post, hit publish, and move on with your next project.

We're only beginning to understand the addictive psychological properties of social media. The dopamine hit we experience when we receive positive feedback from our friends for a picture or tweet we've shared is a powerful force.

Why not harness this cycle to help you stay motivated with your projects?

Share your project wherever you can — on LinkedIn, with friends in the industry, submit it to newsletters or blogs — it's all good. You never know when something you learned and wrote about might be useful to someone else.

Spread the love! If you're inspired to build anything cool after reading this article, send me a link . I'd love to take a look. Good luck and happy building!

If you enjoyed this article, please give it some claps (on a scale of 1–50) below! Follow me here for more content about data engineering and analytics in small to mid-size companies.