AutoML and Domain Driven Design

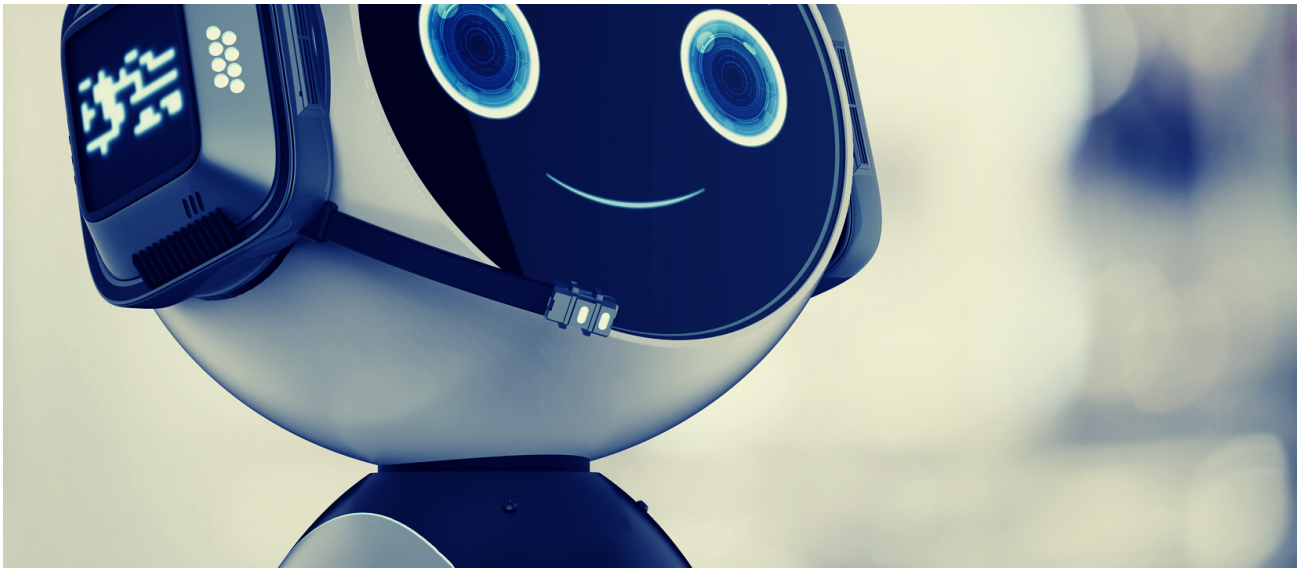
By Shawn Deggans

Can Domain-Driven Design (DDD) improve AutoML implementations? I believe it can, as many Machine Learning experiments involve the same problem-solving approaches used in software development.

This article provides a brief overview of AutoML and no-code development. It then discusses the most common approach to DDD for software development. With a specific use case in mind, I'll walk through a scenario with AutoML as the tactical architecture. I'll explain how DDD should be used to make strategic and tactical decisions regarding AutoML development.

By the end, you'll have a basic understanding of AutoML and DDD. You'll also understand how to apply DDD as a framework to build the right ML solution for the domain problem with organizational stakeholders.

Introduction to AutoML



AutoML is the process of automating the tasks of applying machine learning to real-world problems, according to Wikipedia. So, what is the use case for no-code AutoML?

Many organizations struggle to move beyond the proof-of-concept stage. This can be due to a lack of staff or data estate to support the efforts, the technical complexity of building out the infrastructure to support machine learning in production, or an unclear definition of the business objectives they wish to meet in the problem space.

AutoML helps reduce the risk of failure by providing cloud-native, low- or no-code tools to guide users through the process of curating a dataset and deploying a model. No-code development has enabled organizations to reach their goals without the need for experts. Popular platforms like Microsoft's Power Platform, Zoho Creator, Airtable, BettyBlocks, and Salesforce have made no-code development a regular part of an organization's IT toolset. This puts development tools closer to domain experts, allowing organizations to meet their objectives without the usual IT project overhead.

Critics of the no-code movement point to limited capabilities compared to traditional software development, dependency on vendor-specific systems, lack of control, poor scalability, and potential security risks. However, some organizations may find these risks worth the opportunities and solutions that no-code provides.

AutoML has both critics and champions. Organizations should be aware that AutoML comes with tradeoffs alongside its benefits. Champions of AutoML will point to the following advantages over traditional machine learning development processes:

- Accessibility: AutoML requires minimal knowledge of machine learning concepts and techniques, so you don't need a data scientist or data engineer to guide you through the process.
- Collaboration: Platforms like AutoML, Databricks, and Amazon SafeMaker Studio enable data collaboration in one platform, allowing teams to share data, models, and results with each other.
- Consistency: Automating the optimization of models reduces the chances of human error, improving the consistency of machine learning model results.
- Customization: Platforms like Azure ML and Amazon SageMaker Studio make it easy to customize machine learning environments and set specific requirements for models and parameters.
- Efficiency: AutoML addresses faster ways to preprocess data, select the correct model, and tune hyperparameters, reducing tedious and time-consuming tasks.
- Scalability: AutoML platforms are typically built on cloud architecture, making it easier to handle large datasets and complex problems.

Critics of AutoML warn that using it instead of traditional machine learning could lead to dependence on data quality, ethical concerns, lack of control, lack of interpretability, and lack of transparency.

Data quality is essential: many AutoML platforms require clean data with no issues. Without data engineers or a data quality process, it's unlikely to have clean data. Poor data quality or noisy data can result in inaccurate models.

Ethical considerations must also be taken into account. Algorithms may perpetuate existing biases and discrimination if the data used to train them is unbalanced or biased.

AutoML's abstraction of the complexities of model creation is beneficial, but it also means users can't control what happens during the pipeline process. If the algorithms developed from AutoML are difficult to understand, organizations may not have insight into how decisions are being made, and may unknowingly release models with flawed biases.

Without understanding how the model is making decisions, it's hard to fully grasp the strengths and weaknesses of a model, leading to a lack of transparency.

Additionally, the models generated from AutoML may not be able to handle specialized problems or reach the performance expected of modern ML models.

AutoML is a process of automating the tasks of applying machine learning to real-world problems. It offers cloud native, low to no-code tools that help guide users from a curated dataset to a deployed model. There are benefits and tradeoffs to using AutoML, such as accessibility, collaboration, customization, scalability, and efficiency, but also potential ethical concerns, lack of control, interpretability, transparency, and limited capabilities. Is there a way that we can apply a common, and well-established framework that helps us better exploit the positive elements of AutoML while reducing the negative side-effects brought up by its critiques? I believe we can, and I think Eric Evans' approach to creating a ubiquitous language for the software development team and the domain experts within an organization is the best place to start.

A Quick Overview of Domain-Driven Design for Software Development

DDD is a software development practice that focuses on understanding and modeling the complex domains that systems operate in. It emphasizes the importance of gaining a deep understanding of the problem domain and using this knowledge to guide system design. DDD is a flexible practice based on principles and concepts rather than rigid rules. I use DDD because, as a developer, it encourages me to think more about the domain problem and desired business outcomes than on the technical approach to creating software and infrastructure. It's a lightweight way of building a shared language with someone using a common language. The best example of this I found was in the book *Architecture Patterns with Python* by O'Reilly Media.

Imagine that you, our unfortunate reader, were suddenly transported light years away from Earth aboard an alien spaceship with your friends and family and had to figure out, from first principles, how

to navigate home.

In your first few days, you might just push buttons randomly, but soon you'd learn which buttons did what, so that you could give one another instructions. "Press the red button near the flashing doohickey and then throw that big lever over by the radar gizmo," you might say.

Within a couple of weeks, you'd become more precise as you adopted words to describe the ship's functions: "Increase oxygen levels in cargo bay three" or "turn on the little thrusters." After a few months, you'd have adopted language for entire complex processes: "Start landing sequence" or "prepare for warp." This process would happen quite naturally, without any formal effort to build a shared glossary.

https://learning.oreilly.com/library/view/architecture-patterns-with/9781492052197/ch01.html#:-:text=Imagine that you,a shared glossary.

I love that this example shows the natural process of discovery and how it creates a shared understanding of the spaceship's behavior. DDD is a big topic, so I won't try to cover it all here. The important thing to understand is that DDD is meant to be practiced. It's a process based on discussions and drives towards building deep, shared knowledge about a specific problem.

Why do I believe that someone wishing to learn machine learning, even as an AutoML user, should begin their own DDD practice? I would point to these key concepts of DDD:

1. Bounded context: Isolate and well-define a specific part of the problem domain to manage complexity and prevent misunderstandings between different parts of the domain model. Avoid "boiling the ocean" and taking on more work than can be managed. Bounded context can represent a team, line of business, department, set of related services, data elements, or parameters.
2. Domain expert: Someone with a deep understanding of the problem domain who can provide valuable insights and guidance to the development team. Without access to domain expert, it's difficult to build a solution of real value.
3. Domain model: Representation of the key concepts and relationships in the problem domain, expressed in a shared language. Not an exact replica of reality, but captures the essence of what makes the organization's model unique.
4. Event storming: Collaborative technique to identify and model key events and processes in the problem domain. Uncovers hidden complexity and ensures the domain model reflects the needs of the business.
5. Ubiquitous language: Shared language used by all members of the development team to communicate about the problem domain. Ensures everyone is using the same terminology and concepts.

Why would an AutoML developer want to know DDD?

DDD encourages the use of domain-specific language and concepts in modeling, which can make it easier for domain experts to understand and interpret the results of the models. It also emphasizes the importance of understanding the business context and domain-specific knowledge when solving problems. This can help AutoML developers to build more accurate and effective models.

DDD provides a common language and set of concepts that can help data scientists communicate more effectively with domain experts and other stakeholders. It also emphasizes the importance of designing solutions that are maintainable and adaptable over time, helping AutoML developers to build models that are more robust and resilient to change.

Finally, DDD encourages collaboration between domain experts and technical experts, which can help AutoML developers to better understand the problem they are trying to solve and the impact their solutions will have on the business.

A Use Case: A Machine Learning Model to Diagnose the Flu

I have explored how AutoML and Domain-Driven Design can be used together as a framework to help AutoML developers. Our aim is to take advantage of AutoML's positive aspects while minimizing its negative tradeoffs. I have discussed why an AutoML developer might choose to use DDD as a framework, so in this section I will explain how to implement the process.

I have chosen a relatively simple use case, one that has been extensively studied in terms of building classifiers for the problem domain. Therefore, I will take a basic approach to a non-novel problem, focusing on the DDD process rather than the complexity of the problem domain.

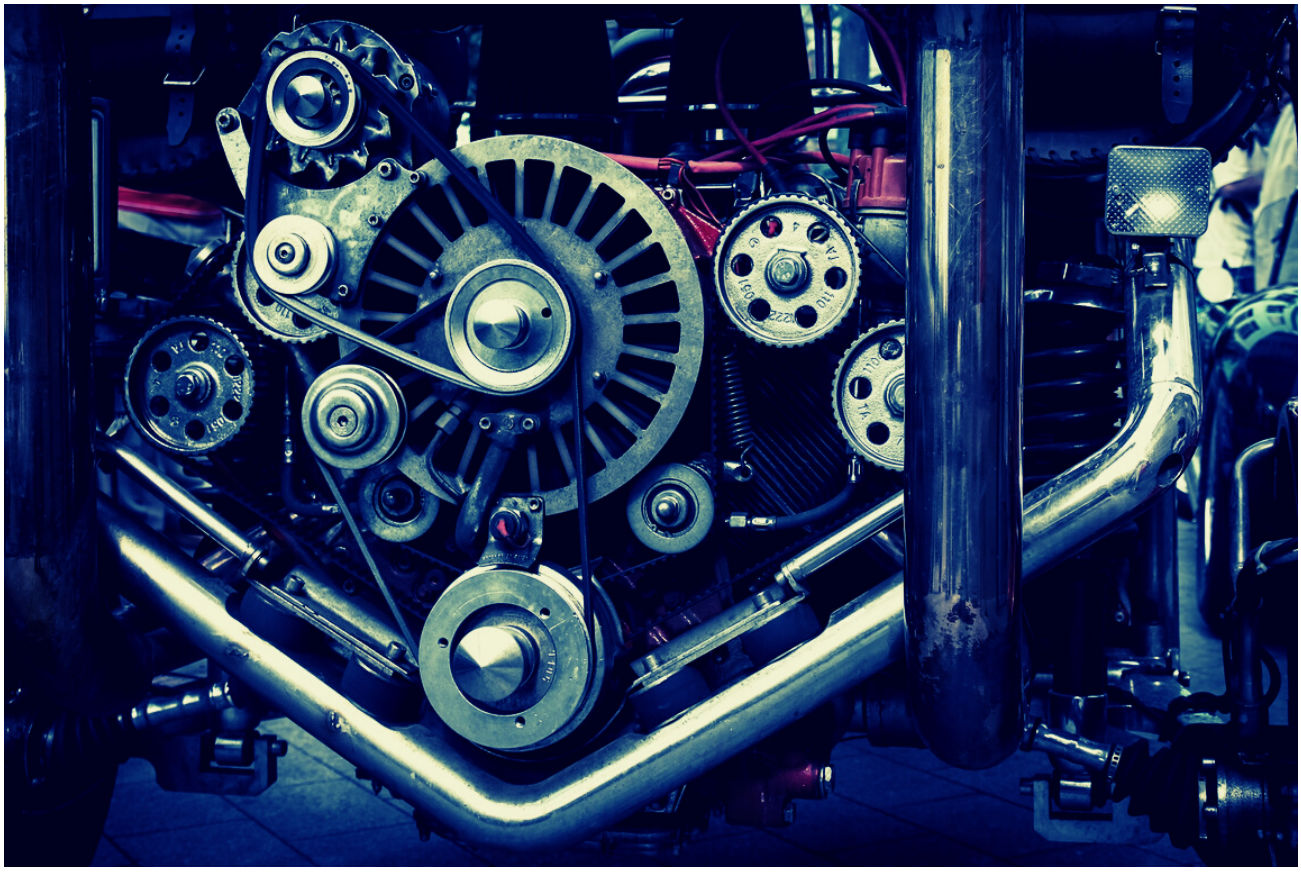
Using domain-driven design (DDD) to build a machine learning model to help diagnose patients with the flu could involve the following steps:

1. Identify the bounded context: The first step would be to identify the bounded context, or specific part of the problem domain, that the model will operate in. In this case, the bounded context might be the process of diagnosing patients with the flu. A conversation with the people making the request for a solution can help establish a scope and general goals. Additionally, tools like Simon Wardley's Wardley Mapping and Teresa Torres' Opportunity Solution Trees can uncover what type of business outcomes the organization is attempting to address and the service or supply chain associated with meeting customer needs. Questions such as what the requesters expected outcomes are from a solution that can diagnose patients who have the flu, if it will lessen time in a waiting room, or if it will make patient intake faster should be asked.
2. Identify the domain experts: The development team should then identify domain experts who have a deep understanding of the problem domain and can provide valuable insights and guidance. These domain experts might include medical professionals who are experienced in diagnosing and treating patients with the flu. The goal is to establish clarity around vocabulary, expected behaviors, and begin to build a vision for the existing strategic, business systems.
3. Define the ubiquitous language: The development team should work with the domain experts to define a shared language, or ubiquitous language, that everyone can use to communicate about the problem domain. This might include defining key terms and concepts related to the flu and its symptoms.
4. Conduct event storming: The development team should use a collaborative technique called event storming to identify and model the key events and processes involved in diagnosing patients with the flu. This might include identifying the symptoms that are most indicative of the flu and the tests that are typically used to diagnose it. A large whiteboard or a shared online collaboration tool can be used to define domain events, commands, policies, and other important elements that make up a working system.
5. Build the domain model: Using the insights and knowledge gained through event storming, the development team should build a domain model that represents the key concepts and relationships in the problem domain. This might include building a model that predicts the likelihood of a patient having the flu based on their symptoms and test results.
6. Use AutoML to build and tune the machine learning model: The development team should then use automated machine learning (AutoML) to build and tune the machine learning model. This might involve selecting an appropriate model type, preprocessing the data, and optimizing the hyperparameters. The AutoML user should build a better understanding of the type of settings they want to establish for their model. The type of model should address the primary problem

type uncovered during the DDD process. If it wasn’t, the AutoML developer should return to additional sessions with the domain experts to tune and solidify the design of the solution.

Overall, by applying DDD principles to the development of the machine learning model, the development team can create a model that is closely aligned with the business needs and can evolve and adapt over time. DDD is not just a process to follow when planning the project, but one to continue throughout the development and deployment of the solution. Involve domain experts in the ML model's lifecycle as long as it creates value.

DDD and AutoML



AutoML is a process of automating machine learning tasks to solve real-world problems. It offers cloud-native, low- to no-code tools to guide users from a curated dataset to a deployed model. Benefits include accessibility, collaboration, customization, scalability, and efficiency. However, there are potential ethical concerns, lack of control, interpretability, transparency, and limited capabilities.

Domain-Driven Design (DDD) is a software development practice that focuses on understanding and modeling complex organization domains. It encourages developers to think more about the domain problem and desired business outcomes than the tactical approach. DDD is a flexible practice built on principles and concepts, not hard rules. It is a lightweight method of building a common language with domain experts.

Does this mean DDD is right for every AutoML endeavor? Not necessarily. When experimenting with data and working with light predictions, bringing the framework of DDD is likely overkill. But when working with complex domains, like healthcare or modern manufacturing, involving domain experts is common. DDD is useful for conducting valuable discussions, capturing important vocabulary, and uncovering unique domain behaviors. It is a tool to include in the professional's toolbox, even when working with low- or no-code solutions. Understanding how the organization will use the solution to meet desired outcomes is essential for success. DDD helps bridge the gap between desired outcomes and machine learning models.