Image created with ChatGPT

Using Machine Learning to Predict the Next Command in Revit _ Part 6: Revit Plugin Using C#, ML.NET & Revit API

Let's BIM Together
728 followers

September 25, 2024

*(Part 6 of a 7 series blog. **Click here to go to Part 1**)*

In previous parts of this blog, I mentioned that I created a console application first because building the Revit plugin was the "easy" part since I was familiar with it. Well, that wasn't entirely true. There was one challenge I wasn't sure about: retrieving previously executed commands and determining whether a command could be executed via the *Revit API*. Before diving into the console application, I started working on the Revit plugin. Unfortunately, I hit a wall early on, but I kept pushing forward. After all, this is a learning journey. I was committed to this idea and this dataset, and even though I had to make some compromises, I was content once I finished the console application.

The good news is that executing a command with the *Revit API* using a command ID is simple. Below is an example of how to post the *Door* command:

```
using Autodesk.Revit.Attributes;
using Autodesk.Revit.DB;
using Autodesk.Revit.UI;

namespace BlogCodeSample
{
    [Transaction(TransactionMode.Manual)]
    [Regeneration(RegenerationOption.Manual)]
    public class Command : IExternalCommand
    {
        public Result Execute(
            ExternalCommandData commandData,
            ref string message,
            ElementSet elements)
        {
            var uiapp = commandData.Application;
            uiapp.PostCommand(
                RevitCommandId.LookupCommandId("ID_OBJECTS_DOOR"));
            return Result.Succeeded;
        }
    }
}
```

Now for the bad news: I couldn't figure out how to get the most recent executed command through the *Revit API*. The workaround I came up with involved reading journal files. In Revit, everything: user interactions, under-the-hood operations, etc., are logged in journal files stored in this location:

C:\Users\UserName\AppData\Local\Autodesk\Revit\Autodesk Revit 20XX\Journals

These journal files are named like *journal.0001.txt*, with the number incrementing each time Revit is launched. Inside the journal files, you can find command IDs along with a lot more information. However, logging one command doesn't always result in a single entry of the command ID. I noticed that even though I posted the command once, there were multiple entries in the journal. With some more analysis of the journal structure, it's possible to find a pattern to pinpoint the actual command execution. Despite the duplication, I was fine with it because my model was trained on the last four unique commands, so I just needed to ensure at least four commands were captured.

However, I came across a big flaw: while the journal captures previous commands, it doesn't log THE MOST RECENT command. So, AG *Feeling Lucky* Revit plugin uses the second-most recent command pretending its the most recent, then the third most recent pretending its the second most recent, and so on. It's a notable flaw in the logic, but we are going to be trucking through for reasons mentioned earlier.

3_GET-LAST-COMMANDS.cs

Link to source code: **https://pastebin.com/0n7Ugksg**

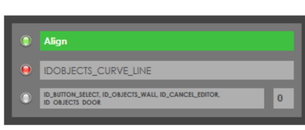
The next step was finding the journal for the active Revit session. The changing variables here are the Revit version (which is part of the folder name) and then the specific journal file for that session. Once found, I extracted the sequence of command IDs. Below is a snippet from *3_GET-LAST-COMMANDS.cs* showing how this is done.

- *journalPath*: Stores the current journal's file location.
- *GetLastJournalLines()*: Retrieves the last 400 lines from the journal.
- *ExtractFilteredCommands()*: Extracts unique command IDs from the last 400 lines, using the *CMD_U-ID_RAW.txt* file from *Part 3* of the blog as the keyword filter.

Finally, I set up an event handler to monitor changes to the active Revit project. This is done by subscribing to the *DocumentChanged* event which triggers the command extraction each time a change is detected. Example:

```
public Result Execute(UIApplication a)
{
    // Subscribe to the DocumentChanged event.
    a.Application.DocumentChanged += new
EventHandler<DocumentChangedEventArgs>(OnDocumentChanged);
    return Result.Succeeded;
}

// Event handler for document changes.
private void OnDocumentChanged(object sender, DocumentChangedEventArgs e)
{
    TaskDialog.Show("Message", "A change was made to the Revit project!");
}
```



AG Feeling Lucky Revit plugin user interface window

Now that the plugin can fetch the "last four" commands used, it can make a prediction based on the same logic as the console application in *Part 4*. I added some fun user interaction pizzazz to the plugin by introducing *Progressive Predictions*: a term I coined in also in *Part 4*. Here's how it works:

1. The plugin displays two predictions: the most likely command and the second most likely.
2. If the first prediction matches the previous one, instead of repeating it, the plugin shows the third and fourth most likely predictions.
3. If it repeats again, it shows the fifth and sixth most likely.
4. At any point, if the top prediction changes, the index resets, and the first and second most likely predictions are displayed again.

A small textbox in the bottom-right corner displays this prediction index. Another textbox in the bottom-left corner displays the previous command IDs retrieved from the journal.

The first-place prediction shows a command description, *Align*, and the corresponding button is highlighted in green. This allows the user to click and execute the command directly. For the second-place prediction, things work a little differently. Instead of showing the description, it displays the *Revit API* command ID. The button for this prediction is greyed out if it turns out to be a non-postable command (i.e., one that cannot be executed via the *Revit API*). This situation arises when the command ID isn't listed in the *ID+DESCRIPTION-FORMATTED.txt* file we discussed back in *Part 2* of this blog.

Next to each command button are LED-looking lights to provide visual feedback:

- A green light for the first prediction, indicating it's good to go.
- A red light for the second prediction, indicating it's a non-postable command.
- A grey, unlit light for previously executed commands. This light briefly flashes when the active Revit project changes and new command IDs are fed into the prediction engine.

These lights add a touch of fun to the user interface and are quite simple to set up. For anyone interested in adding this feature to their own tools, I used a great tutorial that explains how to create these LED effects in WPF, which you can find here:

https://www.codeproject.com/Articles/1248078/WPF-LED-User-Control

You might be wondering: why even bother including non-postable commands in the predictions? Wouldn't it make sense to filter them out completely? Well yes, but there are a couple of reasons why I did this. First, I really wanted to see that red LED light in action! Second, removing all non-postable commands would have significantly reduced my dataset, which I was keen to avoid. In *Part 3*, *Chart 2*, I showed how I reviewed the top 80 most frequent predictions and filtered out non-postable and undesirable commands. As a result, non-postable commands rarely slip through, so most of the time you'll have two executable commands to choose from which is what I wanted.

Go to the next part

Go to the previous part

Part 1 - Introduction & Index

Part 2 - Data Preprocessing Using PowerShell & Python with PyRevit

Part 3 - Data Analysis Using Power BI

Part 4 - Model Trainer Application Using C# ML.NET PowerShell Notepad++

Part 5 - Parameter Optimization Algorithms Using Python with Optuna

Part 6 - Revit Plugin Using C# ML.NET & Revit API

Part 7 - Outro (Canva flow chart)

Download AG Feeling Lucky Plugin for Revit & Trainer/Analysis Console Application here:

https://letsbimtogether.com/blog.html