Using LLMs to Learn to Code

Thougts for Academy mentors

Garrick Aden-Buie

2024-11-14

An LLM is just a…

1. black box
2. pattern matcher
3. text prediction engine
4. stochastic parrot
5. bullshit generator
6. fancy autocomplete

Mental models of LLMs

- The *stochastic parrot* mental model is…
  - true, reassuring, and **unhelpful**.
- The *machines that reason* mental model is…
  - false, terrifying, and **helpful**.

Fundamentals of LLMs

1. You write some text.
2. The LLM continues writing more text.
3. You think you're having a conversation.

Fundamentals of LLMs

1. You write some text.
2. The LLM continues **writing more text**.
3. You think you're having a conversation.

How does it write more text?

1. What else would someone who said what you said say?
2. Based on all written text and code that can be found.
3. Inherently biased toward things people have written about before the model was trained.

Fundamentals of LLMs

1. **You write some text.**
2. The LLM continues **writing more text**.
3. You think you're having a conversation.

Fundamentals of LLMs

1. You write some text.
2. The LLM continues **writing more text**.
3. **You think you're having a conversation.**

Talking with an LLM

Write R code to analyze the `mtcars` dataset.

````
I'll help you write R code to perform a comprehensive analysis
of the built-in `mtcars` dataset in R. Here's a detailed analysis
script with explanations:

```r
# Load required libraries
library(ggplot2)
library(dplyr)
library(corrplot)

# Basic exploration of the dataset
head(mtcars)
str(mtcars)
summary(mtcars)

# Convert relevant variables to factors
mtcars$cyl <- as.factor(mtcars$cyl)
mtcars$am <- as.factor(mtcars$am)
mtcars$vs <- as.factor(mtcars$vs)

# Calculate basic statistics
mean_mpg <- mean(mtcars$mpg)
sd_mpg <- sd(mtcars$mpg)
cat("Mean MPG:", mean_mpg, "\nSD MPG:", sd_mpg, "\n")

# Correlation analysis
correlation_matrix <- cor(mtcars[, c("mpg", "disp", "hp", "wt")])
corrplot(correlation_matrix, method = "color")

# Visualization
````
````

````
# 1. MPG by cylinder count
ggplot(mtcars, aes(x = cyl, y = mpg, fill = cyl)) +
  geom_boxplot() +
  labs(title = "MPG Distribution by Number of Cylinders",
       x = "Number of Cylinders",
       y = "Miles Per Gallon")

# 2. Scatter plot of weight vs. mpg
ggplot(mtcars, aes(x = wt, y = mpg, color = cyl)) +
  geom_point(size = 3) +
  geom_smooth(method = "lm", se = FALSE) +
  labs(title = "Weight vs. MPG",
       x = "Weight (1000 lbs)",
       y = "Miles Per Gallon")

# Statistical analysis

# 1. Linear regression
model <- lm(mpg ~ wt + hp, data = mtcars)
summary(model)

# 2. ANOVA test for MPG differences between cylinder groups
anova_result <- aov(mpg ~ cyl, data = mtcars)
summary(anova_result)

# Group statistics
mtcars %>%
  group_by(cyl) %>%
  summarise(
    mean_mpg = mean(mpg),
    sd_mpg = sd(mpg),
    count = n()
  )

# T-test comparing automatic vs manual transmission
t.test(mpg ~ am, data = mtcars)
```

This R code performs the following analyses:

1. **Data Exploration**:
   - Displays the structure and summary of the dataset
   - Converts categorical variables to factors

2. **Descriptive Statistics**:
   - Calculates mean and standard deviation of MPG
   - Creates group-wise summaries by cylinder count
````
````

````
3. **Correlation Analysis**:
   - Creates a correlation matrix for key numerical variables
   - Visualizes correlations using a correlation plot

4. **Visualizations**:
   - Box plot showing MPG distribution by cylinder count
   - Scatter plot of weight vs. MPG with trend lines

5. **Statistical Analysis**:
    - Linear regression analyzing the relationship between MPG, weight, and
horsepower
   - ANOVA test to compare MPG across different cylinder groups
   - T-test comparing MPG between automatic and manual transmissions

To run this code, you'll need to install the required packages first using:

```r
install.packages(c("ggplot2", "dplyr", "corrplot"))
```

This analysis will give you a comprehensive understanding of:
- The relationship between different variables in the dataset
- How fuel efficiency (MPG) varies with different car characteristics
- Statistical significance of various relationships
- Visual representations of key patterns in the data

The code includes comments explaining each section, making it easy to
understand and modify according to specific needs.
````

Talking with an LLM

> Write R code to analyze the `mtcars` dataset to determine the relationship between `mpg` and `wt`.

````
I'll help you write R code to analyze the relationship between
miles per gallon (mpg) and weight (wt) in the mtcars dataset.
Here's a comprehensive analysis:

```r
# Load the mtcars dataset (it's built into R)
data(mtcars)

# Basic summary statistics
summary(mtcars[c("mpg", "wt")])

# Create a scatter plot
````
````

```
plot(mtcars$wt, mtcars$mpg,
     xlab = "Weight (1000 lbs)",
     ylab = "Miles per Gallon",
     main = "Relationship between MPG and Weight",
     pch = 19)

# Add a regression line
abline(lm(mpg ~ wt, data = mtcars), col = "red")

# Perform linear regression
model <- lm(mpg ~ wt, data = mtcars)

# Display regression summary
summary(model)

# Calculate correlation coefficient
correlation <- cor(mtcars$mpg, mtcars$wt)
print(paste("Correlation coefficient:", round(correlation, 3)))

# Create diagnostic plots
par(mfrow = c(2, 2))
plot(model)
```
```
This code will:
1. Load the mtcars dataset
2. Show summary statistics for both variables
3. Create a scatter plot with a regression line
4. Perform linear regression analysis
5. Calculate the correlation coefficient
6. Generate diagnostic plots for the regression model

The output will show:
- The relationship between weight and MPG (negative correlation expected)
- The strength of the relationship through the correlation coefficient
- The statistical significance of the relationship
- Model diagnostics to check regression assumptions

You can run this code in R or RStudio to see the complete analysis
of the relationship between these variables.
```

Option 1 - Prompt

Write an outline for slides for a presentation titled “using llms as programming learning assistants: benefits and best practices”

Option 1 - Response

```
I'll help you create a presentation outline that covers using LLMs as programming
learning assistants. Here's a structured outline using markdown:

# Using LLMs as Programming Learning Assistants: Benefits and Best Practices

## I. Introduction
- Brief overview of LLMs (Large Language Models)
- The evolution of programming education
- The emergence of AI-powered learning assistance

## II. Benefits of LLMs in Programming Education
### A. Immediate Interactive Support
- 24/7 availability for problem-solving
- Real-time code explanations
- Personalized learning pace

### B. Diverse Learning Approaches
- Multiple explanations of complex concepts
- Code examples in various programming languages
- Step-by-step problem breakdown

### C. Error Detection and Debugging
- Code review assistance
- Common mistake identification
- Debugging strategy suggestions

## III. Best Practices for Implementation

### A. Setting Clear Learning Objectives
- Defining specific programming goals
- Structuring queries effectively
- Maintaining focus on learning outcomes

### B. Promoting Active Learning
- Using LLMs as guides, not solutions
- Encouraging problem-solving skills
- Combining LLM assistance with hands-on practice

### C. Verification and Understanding
- Cross-checking LLM suggestions
- Understanding the reasoning behind solutions
- Building critical thinking skills

## IV. Common Pitfalls and How to Avoid Them

### A. Over-reliance Issues
- Balancing LLM use with independent learning
- Avoiding copy-paste programming
```

```
- Developing self-reliance

### B. Quality Control
- Validating LLM-provided solutions
- Checking for outdated information
- Understanding limitations

## V. Implementation Strategies

### A. Educational Setting Integration
- Classroom implementation guidelines
- Self-study approaches
- Group learning scenarios

### B. Progress Tracking
- Setting measurable goals
- Monitoring learning outcomes
- Adjusting learning strategies

## VI. Future Perspectives
- Emerging trends in LLM-assisted learning
- Potential developments in educational AI
- Preparing for evolving technologies

## VII. Conclusion
- Key takeaways
- Action items for educators and learners
- Resources for further exploration

## VIII. Q&A Session
- Addressing common concerns
- Sharing success stories
- Discussion points

This outline provides a comprehensive structure for a presentation that covers
both the advantages and practical implementation of using LLMs in programming
education, while emphasizing responsible usage and best practices.
```

Option 2 - Prompt

> llms can be powerful, time-saving assistants when you're learning a programming language, but it's also very easy to fall into traps with their use that lead you astray. When you're learning how to write code, in particular in a new language that you've never used before, there are some things you should know and some tactics you can apply to get the best results possible.

Write slide content for a presentation explaining the above to a group of educators who teach programming skills to adult learners. Use short concise bullet points that will form the outline of the presentation

Option 2 - Response

```
Here's a slide outline for a presentation on effectively
using LLMs when learning programming:

## Using LLMs as Programming Learning Assistants:  Benefits & Best Practices

### The Power of LLMs in Learning
- Act as 24/7 programming tutors
- Provide instant code examples and explanations
- Help debug and troubleshoot code
- Explain complex concepts in simple terms

### Common Pitfalls to Avoid
- Over-reliance on generated code
- Accepting answers without verification
- Skipping fundamental concepts
- Missing language-specific best practices
- Copy-pasting without understanding

### Best Practices for Students
- Start with core language documentation
- Use LLMs to explain unclear concepts
- Break down problems before asking
- Verify solutions through testing
- Cross-reference with official resources
- Ask for explanations, not just code

### Effective Learning Strategies
- Combine traditional learning with LLM support
- Practice writing code from scratch first
- Use LLMs for code review and improvement
- Ask for multiple approaches to solutions
- Focus on understanding patterns, not memorizing code

### Teaching Tips for Educators
- Incorporate LLM usage guidelines in curriculum
- Teach students how to form effective prompts
- Encourage critical evaluation of LLM responses
- Design assignments that test understanding, not just functionality
- Show examples of good vs. poor LLM interactions

### Assessment Considerations
- Focus on problem-solving process
```

```
- Evaluate code understanding, not just output
- Include documentation requirements
- Test ability to explain code functionality
- Consider LLM-aware assignment design

### Future-Ready Skills
- Balance traditional learning with AI tools
- Develop prompt engineering skills
- Build strong debugging capabilities
- Foster independent problem-solving
- Maintain current tech awareness

This outline provides a comprehensive framework for discussing
both the benefits and potential issues with using LLMs
in programming education, while offering practical guidance
for implementation.
```

Explain this to me

> Explain this R code to me:
>
> ```
> dd[, Variable := Hmisc::makeNames(tolower(Variable), allow='_')]
> ```

Let me break down this R code:

This is using R's data.table syntax (indicated by the `[, :=]` notation) to modify a column in place. Let's analyze each part:

1. `dd` - This is the data.table object being modified
2. `Variable` - This is the name of the column being transformed
3. `:=` - This is data.table's assignment operator for modifying columns by reference (meaning it changes the original data without creating a copy)
4. `Hmisc::makeNames()` - This is a function from the Hmisc package that standardizes variable names
   - The `::` operator specifically calls the makeNames function from the Hmisc package
5. `tolower(Variable)` - First converts the values in the Variable column to lowercase
6. `allow='_'` - This parameter in makeNames specifies that underscores are allowed in the resulting names

The overall purpose of this code is to:

1. Take the values in the "Variable" column
2. Convert them to lowercase
3. Clean/standardize them using Hmisc's makeNames function, which:
   - Replaces spaces and invalid characters with underscores

  - Ensures names are valid R variable names
  - Maintains underscores (due to `allow='_'`)

For example, if your Variable column had values like:

- "First Name"
- "Last Name!"
- "Phone #"

After this transformation, they might become:

- "first_name"
- "last_name"
- "phone"

This is commonly used when cleaning data to ensure consistent, valid variable names that follow R naming conventions.

Ask follow up questions

> What is the `::` operator called?

The `::` operator in R is called the "namespace operator" or "double colon operator".

It's used to access functions or objects from a specific package namespace directly. When you write `package::function`, you're explicitly telling R to use the function from that specific package, which is useful to:

1. Avoid naming conflicts between packages
2. Make code more explicit about which package a function comes from
3. Access a specific package's function without having to load the entire package with `library()`

For example, in `Hmisc::makeNames()`, you're specifically calling the `makeNames` function from the Hmisc package, even if you haven't loaded Hmisc with `library(Hmisc)`.

There's also a related operator `:::` (triple colon) which can access internal (non-exported) functions from a package, but this is generally discouraged for regular use as it accesses non-public functions that may change without warning.

Ask about conventions

> When should I use `::`? When do R programmers generally use this operator? What are best practices?

Here are the best practices and common scenarios for using the `::` operator in R:

1. **Recommended Uses:**

   - **In package development:**

- When writing package code, always use `::` for functions from other packages
- This makes dependencies explicit and helps avoid namespace conflicts
- Example: `dplyr::filter()` instead of just `filter()`

- **For occasionally used functions:**
  - When you only need one or two functions from a package
  - Avoids loading entire packages with `library()`
  - Example: `lubridate::ymd()` for a single date conversion

- **To resolve ambiguity:**
  - When multiple loaded packages have functions with the same name
  - Common with functions like `filter`, `select`, or `summarize`
  - Example: `stats::filter()` vs `dplyr::filter()`

2. **When to Avoid:**

- **Frequently used functions:**
  - If you're using many functions from a package repeatedly
  - Better to load with `library()` at the top of the script
  - Example: Don't use `dplyr::` for every dplyr function in a data analysis

- **Base R functions:**
  - Generally unnecessary for base R functions
  - Exception: When you need to be explicit about using base R version
  - Example: `base::mean()` usually not needed, just `mean()`

3. **Style Guidelines:**

- **Script Headers:**

```r
# Good practice
library(dplyr)    # For data manipulation
library(ggplot2)  # For plotting

# Then use :: only for occasional functions from other packages
lubridate::ymd("2024-01-01")
```

- **Documentation:**
  - Use `::` in examples and documentation to show package origins
  - Helps users understand dependencies

4. **Common Real-World Examples:**

```r
# Good uses of ::
scales::percent(0.34)  # One-off formatting
jsonlite::fromJSON(file)  # Occasional JSON parsing
```

```r
utils::write.csv(data, "output.csv")  # Clear about source

# Instead of loading rarely-used packages:
# library(scales)
# library(jsonlite)
# library(utils)
```

5. **Performance Note:**

- Using `::` has a very small performance cost
- This is negligible for most applications
- Only matters in high-performance computing or loops

6. **Debugging Benefits:**

- Makes code more maintainable
- Easier to track down function sources
- Helps prevent issues when package updates change function behavior

7. **Package Development Context:**

```r
# In package development, always use :: for external functions
#' @export
my_function <- function(data) {
  data %>%
    dplyr::filter() %>%
    dplyr::select() %>%
    dplyr::mutate()
}
```

The general rule of thumb is: - Use `library()` for packages you use frequently throughout your code - Use `::` for occasional function calls or to resolve conflicts - Always use `::` in package development - Be consistent within your project/team's style guide

This balanced approach helps maintain code clarity while avoiding unnecessary verbosity.

Give the LLM more context

In the release notes for Shiny 1.9.0, I read the following:

> the Shiny object has a new property initializedPromise, which is a Promise-like object that can be awaited or chained with .then(). This Promise-like object corresponds to the shiny:sessioninitialized JavaScript event, but is easier to use because it can be used both before and after the events have occurred.

Can you update your example to take advantage of this new property?

Mental models I've found helpful

1. My personal intern
2. A very helpful rubber duck
3. A “write a blog post just for me” machine
4. A faster, tireless pair programmer