Making optimal use of ChatGPT and other chatbots for data science

AI-powered data science workflows

2025-01-18

Table of contents

1 Introduction

When it comes to data science, having access to the right tools and resources can make a significant difference in the quality and efficiency of your work. Chatbots, such as ChatGPT, have emerged as powerful tools that can assist data scientists in various tasks, from data analysis and visualization to model building and evaluation. In this guide, we'll explore how you can make optimal use of ChatGPT and other chatbots for data science tasks, providing tips and examples to help you leverage these tools effectively.

1.1 Whats in a name?

GPT in ChatGPT stands for Generative Pre-trained Transformer. • Generative: It generates text rather than just analyzing or classifying it. • Pre-trained: It is trained on a large dataset before being fine-tuned for specific tasks. • Transformer: It uses the Transformer architecture, a neural network model designed for natural language processing (NLP).

Essentially, ChatGPT is a chatbot built on a GPT model, which is designed to understand and generate human-like text based on prompts

From a recent article in Nature:

"ChatGPT stunned the world on its launch in November 2022. Powered by a large language model (LLM) and trained on much of the text published on the Internet, the artificial intelligence (AI) chatbot, created by OpenAI in San Francisco, California, makes the latest advances in natural-language processing broadly accessible by providing a dialogue-based interface capable of answering complex questions, composing sophisticated essays and generating source code. One obvious question was: how could this tool improve science?"

Nature: Chatbots in science: What can ChatGPT do for you?

Lets start with a couple examples to get an idea of how to use ChatGPT for data science tasks.

1.2 Example work up of a regression analysis for the iris data set.

1. Prompt: "I want to do a trial data analysis in R of the Fisher iris data set that is distributed with R. I want to start with logistic regression analysis and want to use K-fold cross-validation. I want to use an ROC curve to visualize the results."

comment: this is a good example of a prompt that is clear and specific. It provides many of the necessary context and requirements for the task. However, it is not completely unambiguous. It does not specify the number of folds for cross-validatikn. It also does not specify the R package to use for the aanalysis. There ae several to choose from.

```
install.packages("caret")  # Install if not already installed
install.packages("pROC")   # Install if not already installed
install.packages("glmnet") # Install if not already installed
library(caret)
library(pROC)
library(glmnet)
```

```r
# Prepare binary dataset
data(iris)
iris_binary <- iris[iris$Species != "setosa", ]
iris_binary$Species <- factor(iris_binary$Species)

# Split the data into training and test sets
set.seed(42)
trainIndex <- createDataPartition(iris_binary$Species, p = 0.8, list = FALSE)
train_data <- iris_binary[trainIndex, ]
test_data <- iris_binary[-trainIndex, ]

# Convert the data into matrix format as required by glmnet
x_train <- model.matrix(Species ~ ., train_data)[,-1]
y_train <- train_data$Species

x_test <- model.matrix(Species ~ ., test_data)[,-1]
y_test <- test_data$Species

# Set up cross-validation for Lasso regression on the training data
train_control <- trainControl(method = "cv", number = 10, classProbs = TRUE, summaryFunction =

# Define the lasso model using glmnet (alpha=1 for Lasso)
lasso_model <- train(x = x_train, y = y_train, method = "glmnet", trControl = train_control, tu

# Make predictions on the test set
test_predictions <- predict(lasso_model, newdata = x_test, type = "prob")[,2]

# Plot ROC curve for the test set
roc_curve_test <- roc(y_test, test_predictions, levels = rev(levels(y_test)))
plot(roc_curve_test, col = "blue", lwd = 2, main = "ROC Curve for Lasso Regression on Test Data
v
# Calculate AUC and its 95% confidence interval
auc_value <- auc(roc_curve_test)
ci <- ci.auc(roc_curve_test, conf.level = 0.95)

# Add AUC with 95% confidence interval to the plot
legend_text <- paste("AUC =", round(auc_value, 2), "\n95% CI:", round(ci[1], 2), "-", round(ci
legend("bottomright", legend = legend_text, col = "blue", lwd = 2)
```

On completion of the analysis, the user may have additional questions or requests. For example, they may ask why use 10 folds specifically? Or why split the 80:20? Why use the lasso regression model?

They can just ask the chatbot directly.

Pronpt: why split the data set if we're doing 10-fold cross validation?

ChatGPT answer: ### **Why Not Split? - 10-fold CV inherently splits the data multiple times**
- Each fold acts as a validation set once, while the remaining 9 folds serve as the training set. - Every observation is used for both training and testing across different iterations.

- **An additional 80:20 split is redundant**
  - It would unnecessarily reduce the amount of data available for training.
  - K-fold CV already provides a better estimate of model performance.

1.2.1 When Would We Still Use an 80:20 Split?

- If we want **one final test set** to evaluate model performance **after tuning** hyperparameters using K-fold CV.
  - Example: Use **80% for training**, perform **10-fold CV on this training set**, and then evaluate on the held-out **20% test set**.
  - This prevents **data leakage** and ensures the final model is evaluated on completely unseen data.

1.2.2 Best Practice for Model Selection

1. **Cross-validation alone (no 80:20 split)**
   - When optimizing model parameters and assessing performance.
   - Avoids unnecessary data loss.
   - Every observation contributes to both training and validation.
2. **80:20 Split + Cross-validation on training set**
   - If we want to **train the final model on the full dataset**,
   - Then evaluate **generalization** on a completely independent holdout test set.

For your Fisher Iris dataset analysis, **just using 10-fold CV is sufficient** unless you plan to tune hyperparameters before a final evaluation. for a 95% confidence interval calculationgM "should I split my data set into train and test parts if I'm using k-fold cross-validation?" "add a 95% confidence interval calculation to the AUC annotation in the ROC plot" "do the same analysis but use lasso regression instead of logistic regression"

2 R code

```r
# Load necessary packages
library(tidyverse)   # Data manipulation
library(caret)       # K-fold cross-validation
library(pROC)        # ROC curve analysis

# Load the Iris dataset
data(iris)

# Filter to create a binary classification problem
iris_bin <- iris %>%
  filter(Species != "virginica") %>%  # Use only "setosa" and "versicolor"
  mutate(Species = factor(Species, levels = c("setosa", "versicolor")))

# Set up 10-fold cross-validation
set.seed(123)  # For reproducibility
cv_folds <- createFolds(iris_bin$Species, k = 10, list = TRUE, returnTrain = TRUE)

# Store results
roc_curves <- list()
predictions <- data.frame(actual = iris_bin$Species, predicted_prob = NA)

# Perform K-fold cross-validation
for (i in seq_along(cv_folds)) {
  # Split into training and test sets
  train_index <- cv_folds[[i]]
  train_data <- iris_bin[train_index, ]
  test_data <- iris_bin[-train_index, ]

  # Fit logistic regression model
  model <- glm(Species ~ ., data = train_data, family = binomial)

  # Make predictions (probability of being "versicolor")
  test_probs <- predict(model, newdata = test_data, type = "response")

  # Store predictions
  predictions$predicted_prob[-train_index] <- test_probs

  # Compute ROC curve
  roc_curves[[i]] <- roc(test_data$Species, test_probs, levels = rev(levels(iris_bin$Species)))
}

# Compute mean ROC curve
roc_avg <- roc(predictions$actual, predictions$predicted_prob, levels = rev(levels(iris_bin$Sp

# Plot ROC curve
plot(roc_avg, col = "blue", lwd = 2, main = "ROC Curve for Logistic Regression (Iris Data)")
abline(a = 0, b = 1, lty = 2, col = "gray")  # Diagonal reference line
```

3 Complete R Code for Lasso Regression with LOOCV on the Entire Dataset

```
# Prepare binary dataset
data(iris)
iris_binary <- iris[iris$Species != "setosa", ]
iris_binary$Species <- factor(iris_binary$Species)

# Convert the data into matrix format as required by glmnet
x <- model.matrix(Species ~ ., iris_binary)[,-1]
y <- iris_binary$Species

# Set up LOOCV for Lasso regression on the entire dataset
train_control <- trainControl(method = "LOOCV", classProbs = TRUE, summaryFunction = twoClassSu

# Define the lasso model using glmnet (alpha=1 for Lasso)
lasso_model <- train(x = x, y = y, method = "glmnet", trControl = train_control, tuneGrid = exp

# Make predictions using LOOCV
predictions <- predict(lasso_model, newdata = x, type = "prob")[,2]

# Plot ROC curve for the entire dataset
roc_curve <- roc(y, predictions, levels = rev(levels(y)))
plot(roc_curve, col = "blue", lwd = 2, main = "ROC Curve for Lasso Regression with LOOCV")

# Calculate AUC and its 95% confidence interval
auc_value <- auc(roc_curve)
ci <- ci.auc(roc_curve, conf.level = 0.95)

# Add AUC with 95% confidence interval to the plot
legend_text <- paste("AUC =", round(auc_value, 2), "\n95% CI:", round(ci[1], 2), "-", round(ci
legend("bottomright", legend = legend_text, col = "blue", lwd = 2)
```

4 today's date is 2024-11-10

Trying to get the vim plugin for openai to work. having trouble vim can't find the python module OpenAI

5 useful prompts:

Have chatGPT provide the README.md file as a download. “provide the README.md file as a download.”

6 Guide to Building, Submitting, and Managing the `zzlongplot` R Package

6.1 Step 1: Setting Up the Package Structure

1. **Create a New Package Directory**:
   - Use `usethis` to create a package directory:

     ```
     usethis::create_package("path/to/zzlongplot")
     ```

   - This sets up the necessary directory structure with folders like `R/` and files like `DESCRIPTION`.
2. **Add the Core Script**:
   - Place the `zzlongplot.R` file in the `R/` directory.
3. **Set Up the `DESCRIPTION` File**:
   - Edit the `DESCRIPTION` file to include metadata about the package. Use `usethis::use_description()` to create and fill this file:

     ```
     usethis::use_description(fields = list(
       Title = "Flexible Longitudinal Plotting in R",
       Description = "Provides tools for generating observed and change plots in longitudi
       Version = "0.1.0",
       Author = "Your Name [aut, cre]",
       Maintainer = "Your Name <your_email@example.com>",
       License = "MIT",
       Encoding = "UTF-8"
     ))
     ```

4. **Add Dependencies**:
   - List package dependencies in the `DESCRIPTION` file under `Imports`. For example:

     ```
     Imports:
       dplyr,
       ggplot2,
       patchwork
     ```

6.2 Step 2: Document the Package

1. **Add Roxygen2 Comments**:
   - Ensure all functions in `zzlongplot.R` have Roxygen2 comments for documentation.
2. **Generate Documentation**:
   - Run:

     ```
     devtools::document()
     ```

   - This creates help files in the `man/` directory and updates the `NAMESPACE` file.
3. **Create a Vignette**:
   - Add the vignette to introduce the package:

     ```
     usethis::use_vignette("Introduction_to_zzlongplot")
     ```

   - Place the provided `zzlongplot-vignette.Rmd` file in the `vignettes/` directory and build it:

     ```
     devtools::build_vignettes()
     ```

6.3 Step 3: Test the Package

1. **Add Unit Tests**:
   - Use `usethis` to set up a testing framework:

     ```
     usethis::use_testthat()
     ```

   - Place the `test-zzlongplot.R` file in `tests/testthat/`.
2. **Run Tests**:
   - Run all tests:

     ```
     devtools::test()
     ```

6.4 Step 4: Check the Package

1. **Build and Check**:
   - Build the package:

```
devtools::build()
```

- Check the package for CRAN compliance:

```
devtools::check()
```

2. **Fix Issues**:
   - Address any warnings or errors reported by `devtools::check()`.

6.5 Step 5: Submit to CRAN

1. **Prepare for Submission**:
   - Ensure the package passes `R CMD check` with no warnings, errors, or notes.
   - Compress the package into a `.tar.gz` file using:

```
devtools::build()
```

2. **Submit to CRAN**:
   - Go to the CRAN submission page.
   - Upload the `.tar.gz` file and fill out the required metadata.
3. **Respond to Feedback**:
   - CRAN maintainers might request changes. Address them promptly and resubmit if needed.

6.6 Step 6: Set Up a GitHub Repository

1. **Initialize a Git Repository**:
   - In the package directory, run:

```
git init
git add .
git commit -m "Initial commit"
```

2. **Create a Repository on GitHub**:
   - Use the GitHub website or the `gh` CLI tool:

```
gh repo create yourusername/zzlongplot --public --source=.
```

3. **Push the Code**:

- Push the code to GitHub:

```
git branch -M main
git push -u origin main
```

6.7 Step 7: Manage the Development Repository

1. **Add Version Control**:
   - Use Git for version control. For example, create a branch for new features:

```
git checkout -b feature-new-plot
```

2. **Tag Releases**:
   - Tag versions for releases:

```
git tag -a v0.1.0 -m "First release"
git push origin v0.1.0
```

3. **Add Continuous Integration**:
   - Set up GitHub Actions for testing:

```
usethis::use_github_action_check_standard()
```

4. **Publish Development Versions**:
   - Use GitHub to manage development versions and issues.

6.8 Step 8: Maintain the Package

1. **Address Issues**:
   - Monitor and address issues reported by users.
2. **Update the Package**:
   - For updates, increment the version number in `DESCRIPTION` and tag the new version.

6.9 Prerequisites

In development

6.10 Step-by-Step Implementation

In development

6.11 Key Takeaways

In development

6.12 Further Reading

In development