- data science
- training and education
- research
- strategic planning

For each, we present the unaltered transcripts generated through our interactions with ChatGPT. We then comment on the modes of interaction with ChatGPT and note its strengths and limitations in the context of the specific case study.

We found that ChatGPT contributed to the quality of products generated and expedited their development. However, ChatGPT did not eliminate the need for human involvement: knowledgeable people were needed to decompose complex tasks into simpler ones that ChatGPT could accomplish, and they needed to verify its outputs. More nuanced findings we report pertain specifically to ChatGPT—they may not hold for other existing LLMs or ones under development. Yet the finding that ChatGPT can enhance productivity, but not replace human involvement, holds for all LLMs.

Case Study 1: Data Science

In today's data-driven business landscape, the adoption of artificial intelligence (AI) has become a critical factor for organizations seeking to gain a competitive edge. In a 2022 survey of global executives, 50 percent of respondents reported their organizations had embraced AI in at least one business unit [McKinsey 2022]. Of those, approximately one-third reported reduced operational costs, while the remainder reported increased revenue. This highlights the compelling bottom-line value proposition associated with data science capabilities, including AI.

Despite the apparent benefits, many companies encounter challenges when integrating data science capabilities into operations and product lines. Data science products are fundamentally software products. Hence, data science teams must contend with software engineering obstacles. However, additional challenges arise due to the unique nature of machine learning (ML-) and AI-driven products. Some of the greatest challenges faced by data scientists include the following:

- Quality: Data science products are software-based and require adherence to quality assurance standards. The code they use must be syntactically and functionally correct. Additionally, the code must be optimized and secure.
- Efficiency: Even in companies with established data science teams, demand for data science products often surpasses capacity. Consequently, data science teams are challenged to develop reliable products efficiently, striking a delicate balance between speed and quality.
- Sustainment: Once a data science product is developed, it must be maintained and supported. The ongoing costs of these lifecycle management activities may significant.

- Collaboration: Data science teams typically comprise individuals with overlapping yet distinct skill sets. Successful development and delivery of new capabilities require seamless collaboration among team members.

Using LLMs to Enhance Data Science Processes

Table 1 outlines the main challenges data science teams face and how LLMs can address them. The cells in orange marked "Medium" denote medium alignment for using an LLM to address that challenge, and the cells in green marked "High" denote high alignment.

For example, an LLM can retrieve information about an existing Python library to accomplish a programming goal. The LLM can generate syntax using the library, review code generated by human developers, and recommend changes based on runtime errors. All these activities improve the quality of the code and increase developer efficiency. Besides generating and reviewing code, the LLM can automatically produce documentation to explain what the various objects do. This makes it easier to sustain the code. Finally, the LLM can refactor existing code into different languages, making it possible for data science teams to integrate products created in R, Python, and other languages.

Table 1: Using LLMs to Address Challenges in Data Science

	Challenges			
LLM Capabilities	**Quality**	**Efficiency**	**Sustainment**	**Collaboration**
Information retrieval	High	High	Medium	Medium
Code review	High	High	Medium	High
Code completion	Medium	High	Medium	Medium
Debugging	Medium	High	Medium	Medium
Documentation	Medium	High	High	High
Automated test construction	High	High	High	High
Code translation	Medium	High	High	High

Data Science Case Study

The following case study displays a sequence of interactions intended to construct an intrusion detection system. The system learns a predictive model to distinguish between "malicious" connections (i.e., intrusions) and "benign" ones (i.e., normal connections). The system is trained using data collected as part of the 1998 DARPA Intrusion Detection Evaluation Program.[1] The dataset is used for

[1] https://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html

illustrative purposes—the sequence of interactions in the example could be repeated for current cyber-security problems.

PROMPT: *Write me a Python script to do the following:*

1. *Load the kddcup99 data using sklearn.datasets.*
2. *Assign predictors to the variable X and outcomes to the variable Y.*
3. *Convert the variable X to a Pandas dataframe and add column names.*
4. *Convert all elements of Y that contain the substring 'normal' to 0 and convert all other elements to 1.*

ChatGPT: Certainly! Here's a Python script that performs the steps you've outlined:

```
import pandas as pd
from sklearn.datasets import fetch_kddcup99

# Load the KDD Cup 1999 dataset
kddcup99 = fetch_kddcup99()

# Assign predictors to variable X and outcomes to variable Y
X = kddcup99.data
Y = kddcup99.target

# Convert predictors to a pandas DataFrame and add column names
df = pd.DataFrame(X, columns=kddcup99.feature_names)

# Convert elements of Y containing 'normal' to 0 and others to 1
Y = [0 if 'normal' in outcome else 1 for outcome in Y]

# Print the first few rows of the DataFrame and the modified Y
print("Predictors (X):")
print(df.head())
print("\nOutcomes (Y):")
print(Y)
```

PROMPT*: The script returns the error: TypeError: a bytes-like object is required, not 'str'.*

ChatGPT: Apologies for the error. It seems that the fetch_kddcup99() function returns byte strings instead of regular strings. To resolve this issue, we can convert the byte strings to regular strings before performing any operations. Here's the modified script:

```
# Convert byte strings to regular strings
Y = [outcome.decode() for outcome in Y]
```

PROMPT: *Convert all columns of X to float and retain non-NA columns.*

ChatGPT: To convert all columns of X to float and retain only the non-NA columns, you can modify the script as follows:

```
# Convert all columns to float and retain non-NA columns
df = df.apply(pd.to_numeric, errors='coerce')
df = df.loc[:, ~df.isna().any()]
```

PROMPT*: Should I normalize predictors before or after creating the train-test split?*

ChatGPT: The general recommendation is to perform normalization (or standardization) of predictors after creating the training-test split. This approach helps ensure that the normalization process is applied independently to the training and test sets, mimicking real-world scenarios where the model encounters new, unseen data during deployment.

PROMPT*: Okay. Create an 80/20 train-test split.*

RESPONSE: To create an 80/20 training-test split using the DataFrame df and the outcome variable Y, you can utilize scikit-learn's train_test_split function. Here's the updated script:

```
from sklearn.model_selection import train_test_split

# Create an 80/20 training-test split
X_train, X_test, Y_train, Y_test = train_test_split(df, Y, test_size=0.2,
ran-dom_state=42)
```

PROMPT*: Now, standardize the predictors and fit a logistic regression model using SGDClassifier. Report the model's accuracy on the test set.*

ChatGPT: To standardize the predictors and fit a logistic regression model using SGDClassifier in scikit-learn, and then report the model's accuracy on the test set, you can use the following script:

```
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import SGDClassifier
from sklearn.metrics import accuracy_score

# Standardize the predictors
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# Fit a logistic regression model using SGDClassifier
logistic_regression = SGDClassifier(loss='log', random_state=42)
logistic_regression.fit(X_train_scaled, Y_train)

# Make predictions on the test set
Y_pred = logistic_regression.predict(X_test_scaled)
```

```
# Calculate the accuracy of the model on the test set
accuracy = accuracy_score(Y_test, Y_pred)
print("Accuracy on the test set:", accuracy)
```

PROMPT: *Can you do that with Random Forest instead?*

ChatGPT: Certainly! Here's the updated script that uses Random Forest classifier instead of logistic regression:

```
from sklearn.ensemble import RandomForestClassifier

# Fit a Random Forest classifier
random_forest = RandomForestClassifier(random_state=42)
random_forest.fit(X_train_scaled, Y_train)

# Make predictions on the test set
Y_pred = random_forest.predict(X_test_scaled)

# Calculate the accuracy of the model on the test set
accuracy = accuracy_score(Y_test, Y_pred)
print("Accuracy on the test set:", accuracy)
```

PROMPT: *Can you convert the complete script to R?*

ChatGPT: Certainly! Here's the updated script that uses R:

```
library(mlbench)
library(caret)
library(randomForest)

# Fetch the KDD Cup 1999 dataset
data("kddcup99")

# Assign predictors to variable X and outcomes to variable Y
X <- kddcup99$data
Y <- kddcup99$target

# Convert byte strings to regular strings
Y <- sapply(Y, function(outcome) as.character(outcome))

# Convert predictors to a data frame
df <- as.data.frame(X)

# Convert all columns to numeric and retain non-NA columns
df <- df[ , sapply(df, is.numeric)]
```

```
# Convert elements of Y containing 'normal' to 0 and others to 1
Y <- ifelse(grepl("normal", Y), 0, 1)

# Create an 80/20 training-test split
set.seed(42)
train_indices <- createDataPartition(Y, p = 0.8, list = FALSE)
X_train <- df[train_indices, ]
X_test <- df[-train_indices, ]
Y_train <- Y[train_indices]
Y_test <- Y[-train_indices]

# Standardize the predictors
preprocessor <- preProcess(X_train, method = c("center", "scale"))
X_train_scaled <- predict(preprocessor, X_train)
X_test_scaled <- predict(preprocessor, X_test)

# Fit a Random Forest classifier
random_forest <- randomForest(X_train_scaled, Y_train, ntree = 100)

# Make predictions on the test set
Y_pred <- predict(random_forest, X_test_scaled)

# Calculate the accuracy of the model on the test set
accuracy <- sum(Y_pred == Y_test) / length(Y_test)
cat("Accuracy on the test set:", accuracy, "\n")
```

PROMPT*: Okay. Now, create a Python function that takes a new record as input and uses the pre-trained Random Forest to classify it as benign or malicious.*

ChatGPT: To create a function for prediction, you'll need to ensure that the new record matches the same format and preprocessing steps as the training data. Here's how you can do it:

```
def predict_new_record(new_record, model, scaler, feature_names):
    '''
    Predicts if a new record is benign or malicious using a pre-trained
model.

    Parameters:
    new_record: a pandas Series or similar iterable with the features of
the record
    model: pre-trained machine learning model
    scaler: pre-trained StandardScaler for normalizing the features
    feature_names: a list of feature names used in the trained model

    Returns:
    String: 'benign' if the prediction is 0 and 'malicious' otherwise
```

```
    '''

    # Convert new_record to pandas DataFrame
    new_record_df = pd.DataFrame([new_record], columns=feature_names)

    # Perform the same preprocessing steps as in training
    new_record_df = new_record_df.apply(pd.to_numeric, errors='coerce')

    # We drop NA columns only if they are also dropped in the main da-ta-
set
    new_record_df = new_record_df.loc[:, ~new_record_df.isna().any()]

    # Standardize the predictors
    new_record_scaled = scaler.transform(new_record_df)

    # Make a prediction
    pred = model.predict(new_record_scaled)

    return 'benign' if pred[0] == 0 else 'malicious'
```

Discussion of Data Science Case Study

In the case study, ChatGPT generated computer code to implement a rudimentary intrusion detection system. An actual system for real-time intrusion detection system would be far more complex. Nonetheless, this example demonstrates how, through a series of interactions with ChatGPT, a data scientist could build the system's components.

Figure 1 illustrates the manner of interaction. The developer gives ChatGPT high-level goals such as, "Create a Python function that takes a new record as input and uses the pre-trained Random Forest to classify it as benign or malicious." The developer may also ask ChatGPT questions like, "Should I normalize predictors before or after creating the train-test split?" In response, ChatGPT provides code and recommendations. The net effect is that the joint human-machine team can produce higher-quality software, documentation, and other supporting artifacts and do so in a more efficient manner.

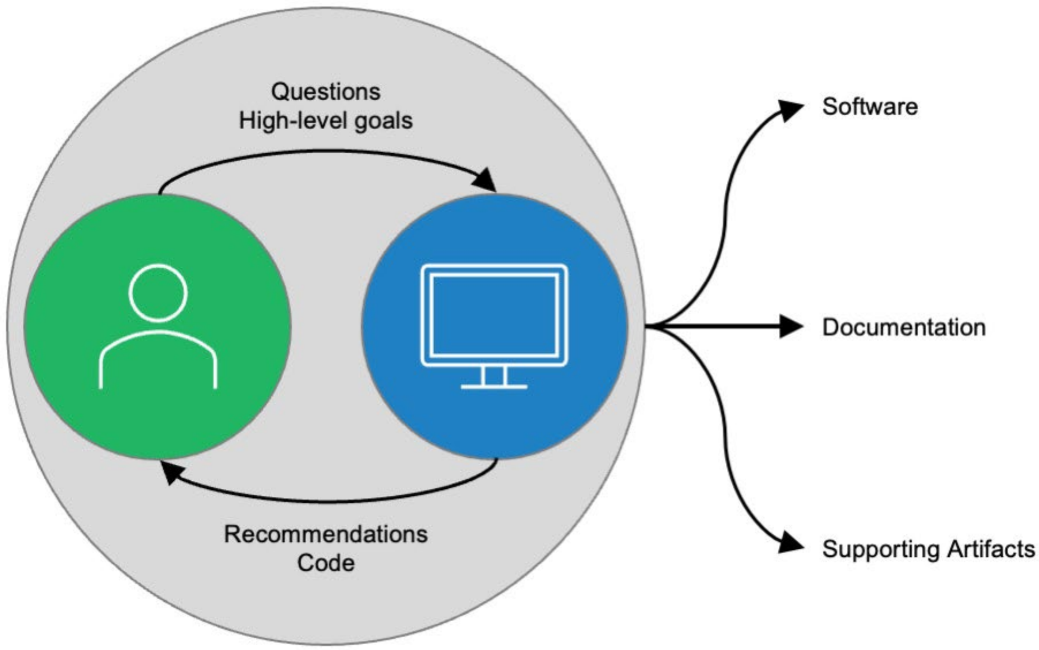

Figure 1: Using ChatGPT for Paired Programming

Three aspects of the case study are notable. First, ChatGPT demonstrates an impressive ability to generate a syntactically and functionally correct script from high-level instructions such as, "Standardize the predictors, and fit a logistic regression model." It chooses the appropriate Python libraries, programming idioms, and functions to fulfill the objectives outlined in the prompts. Furthermore, it accurately interprets the runtime error and modifies the script accordingly.

Second, ChatGPT adheres to style conventions. The variable names it opts for are informative, and it documents the purpose of each code block. Moreover, it incorporates print statements to enable the user to monitor the program's execution flow. ChatGPT also includes a document string in the function, elucidating its purpose.

Third, ChatGPT's programming knowledge extends beyond a single language. This example showcases its proficiency in two languages (Python and R), but ChatGPT's repertoire includes over 20 other languages. Although not demonstrated here, ChatGPT also has the ability to modify scripts if specific Python libraries and functions, such as scikit-learn, are unavailable.

However, ChatGPT does have limitations in the realm of programming. For instance, it does not execute code in real time, which means it cannot directly validate the functionality or correctness of the code. Additionally, as a pretrained model, ChatGPT does not have access to the most recent programming libraries or updates to existing ones. Furthermore, ChatGPT is not equipped to autonomously complete large programming tasks. For the creation of more complex software products, a human developer must first break down the task into simpler components.