

# DEV-GPT PROJECT

**Finished Presentation**

Members:

Arambhumi Reddy(ar22br)

Lakshmi Harika G(lg22k)

Efe Ataman(ea22h)



# RESEARCH QUESTION 1

**Question:**

**What type of assistance are programmers seeking more from ChatGPT? Define a set of classes like Algorithm Design, Debugging, Syntax help, Code Optimization and General Programming Questions and categorize prompts into these classes.**

**Contributor: Lakshmi Harika G**

# METHODOLOGY

## INITIAL PROPOSAL – Topic Modelling

- Vectorization of words
- Passing the matrix of vectors to LDA model using wordnet lemmatizer and doc2bow functions
- Categorizing the topics of each prompt into one of the class labels using cosine similarity.

## LDA Results:


```
(0, '0.059*"break" + 0.040*"=" + 0.013*"#" + 0.010*"return"')
(1, '0.045*"{" + 0.040*"=" + 0.031*"}" + 0.020*"const"')
(2, '0.009*"-" + 0.005*"use" + 0.004*"want" + 0.004*"make"')
(3, '0.134*"<string" + 0.017*"|" + 0.005*"e" + 0.005*"</string-
array">')
(4, '0.011*"like" + 0.008*"file" + 0.006*"tell" +
0.005*"something"')
```

## OPTIMISED PROPOSAL - GPT2


- Used GPT-2 model and word vectorizer for contextualized embeddings.
- Created word embeddings for the list of classes
- Categorized the prompts into one of the classes using cosine similarity.

# CODE SNIPPETS

## Prompt Embeddings:

```
31s  from transformers import GPT2Model, GPT2Tokenizer


tokenizer = GPT2Tokenizer.from_pretrained("gpt2")
model = GPT2Model.from_pretrained("gpt2")
```



|                         |             |                                     |
|-------------------------|-------------|-------------------------------------|
| vocab.json: 100%        | <div></div> | 1.04M/1.04M [00:00<00:00, 2.63MB/s] |
| merges.txt: 100%        | <div></div> | 456k/456k [00:00<00:00, 4.06MB/s]   |
| tokenizer.json: 100%    | <div></div> | 1.36M/1.36M [00:00<00:00, 7.31MB/s] |
| config.json: 100%       | <div></div> | 665/665 [00:00<00:00, 10.6kB/s]     |
| model.safetensors: 100% | <div></div> | 548M/548M [00:06<00:00, 82.8MB/s]   |

+ Code

+ Text


```
1h  [5] tokenizer.pad_token = tokenizer.eos_token

import torch

def get_embeddings(text):
    inputs = tokenizer(text, return_tensors="pt", padding=True, truncation=True, max_length=512)
    outputs = model(**inputs)
    return outputs.last_hidden_state[:, 0, :].detach() # Taking the embedding of the first token

prompt_embeddings = [get_embeddings(prompt) for prompt in prompts]
```

## Category Embeddings:

```
1s  [7] list_of_tokens_for_similarity = {'Algorithm Design': ['Algorithms', 'create', 'Design', 'PseudoCode', 'Complexity', 'Efficie']
category_embeddings = {}
for category, strings in list_of_tokens_for_similarity.items():
    combined_string = ' '.join(strings)
    category_embeddings[category] = get_embeddings(combined_string)
```

# CODE SNIPPETS

## Cosine Similarity and Classification:

```
▶ from sklearn.metrics.pairwise import cosine_similarity

assigned_categories1 = {}

for i, prompt_embedding in enumerate(prompt_embeddings):
    # Compute cosine similarity between this prompt and all categories
    similarities = {cat: cosine_similarity(prompt_embedding, cat_emb.reshape(1, -1))[0][0] for cat, cat_emb in category_embeddings.items()}

    # Find the category with the highest similarity
    max_category = max(similarities, key=similarities.get)

    # Assign the category to the prompt
    prompt_number = f"Prompt {i}"
    assigned_categories1[prompt_number] = max_category

# Now 'assigned_categories' is the desired dictionary
print(assigned_categories1)
```

```
➞ {'Prompt 0': 'Debugging', 'Prompt 1': 'Debugging', 'Prompt 2': 'Code completion', 'Prompt 3': 'General Programming Questions', 'Prompt 4':
```

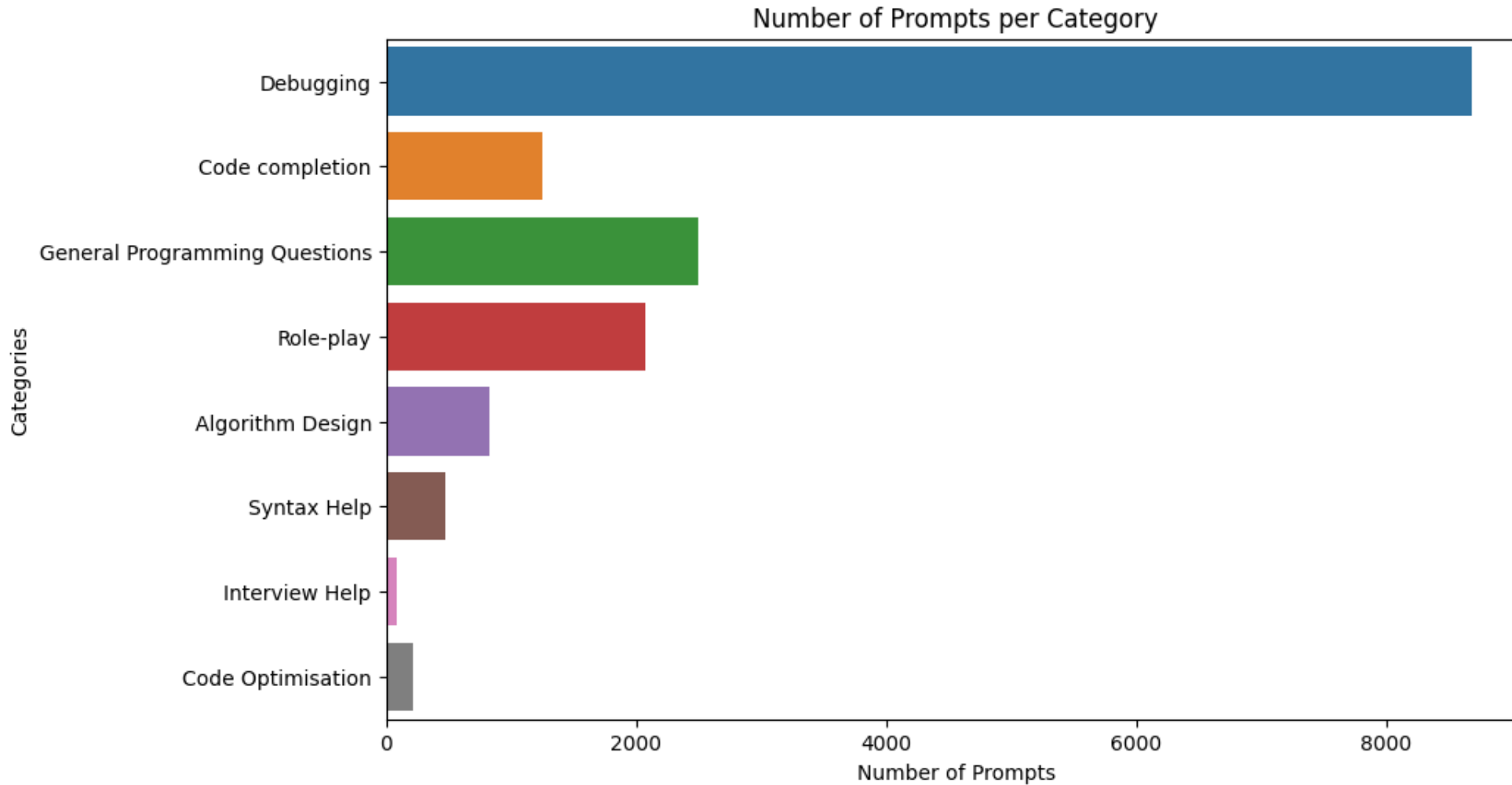
## Visualization:

```
▶ import matplotlib.pyplot as plt
import seaborn as sns
from collections import Counter

category_counts1 = Counter(assigned_categories1.values())

# Creating a bar plot
plt.figure(figsize=(10, 6))
sns.barplot(x=list(category_counts1.values()), y=list(category_counts1.keys()))
plt.xlabel('Number of Prompts')
plt.ylabel('Categories')
plt.title('Number of Prompts per Category')
plt.show()
```

# RESULTS



# CLASSIFICATION MODEL AND ACCURACY

## Training a Linear SVM model:

```
# Split into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

# Initialize and train the SVM model
model = SVC(kernel='linear')
model.fit(X_train, y_train)

# Predict on the test set
y_pred = model.predict(X_test)

# Evaluate the performance
print("Accuracy:", accuracy_score(y_test, y_pred))
print("\nClassification Report:\n", classification_report(y_test,
y_pred, target_names=encoder.classes_))
```

Accuracy: 0.7465880893300249

### Classification Report:

|                               | precision | recall | f1-score | support |
|-------------------------------|-----------|--------|----------|---------|
| Algorithm Design              | 0.44      | 0.16   | 0.23     | 165     |
| Code Optimisation             | 0.64      | 0.26   | 0.37     | 34      |
| Code completion               | 0.87      | 0.66   | 0.75     | 245     |
| Debugging                     | 0.73      | 0.95   | 0.82     | 1719    |
| General Programming Questions | 0.78      | 0.50   | 0.61     | 533     |
| Interview Help                | 0.00      | 0.00   | 0.00     | 20      |
| Role-play                     | 0.82      | 0.66   | 0.73     | 422     |
| Syntax Help                   | 0.81      | 0.40   | 0.53     | 86      |
| accuracy                      |           |        | 0.75     | 3224    |
| macro avg                     | 0.64      | 0.45   | 0.51     | 3224    |
| weighted avg                  | 0.74      | 0.75   | 0.72     | 3224    |



# FINAL UPDATE

Previous Accuracy: 74.65%

Trained a XGBoost algorithm on the prompts and prompt categories and obtained an accuracy of 77.5%

Improved the accuracy further by tuning the hyperparameters of the LinearSVC using gridSearch and achieved an accuracy of 78.16%

```
[CV] END .....C=100, gamma=0.001, kernel=poly; total time= 23.0s
[CV] END .....C=100, gamma=0.001, kernel=poly; total time= 22.7s
[CV] END .....C=100, gamma=0.001, kernel=sigmoid; total time= 26.3s
[CV] END .....C=100, gamma=0.001, kernel=sigmoid; total time= 26.2s
[CV] END .....C=100, gamma=0.001, kernel=sigmoid; total time= 26.5s
[CV] END .....C=100, gamma=0.001, kernel=sigmoid; total time= 26.6s
[CV] END .....C=100, gamma=0.001, kernel=sigmoid; total time= 26.3s
Best SVC Params: {'C': 100, 'gamma': 1, 'kernel': 'rbf'}
Best SVC Accuracy: 0.7816639197480636
```

# METHODOLOGY

Define a function to preprocess the text to:

- Convert text to lower case
- Remove Punctuation
- Remove Numbers
- Tokenization
- Remove stop words
- Stemming
- Join words into single string

```
def preprocess(text):  
    text = text.lower()  
    text = text.translate(str.maketrans('', '', string.punctuation))  
    text = re.sub(r'\d+', '', text)  
    words = word_tokenize(text)  
    words = [word for word in words if word not in stopwords.words('english')]  
    stemmer = PorterStemmer()  
    words = [stemmer.stem(word) for word in words]  
    text = ' '.join(words)  
    return text
```

# RESEARCH QUESTION 2

**Question:** Do Developers Receive Comprehensive Answers from ChatGPT on their Initial queries, or do they often require follow-up interactions for clarity?

**Contributor:** Efe Ataman

# METHODOLOGY

-Vectorize the text of the initial prompts using TF-IDF (Term Frequency – Inverse Document Frequency)

```
vectorizer = TfidfVectorizer(preprocessor=preprocess)
initial_prompts = [conversation[0]['prompt'] for conversation in all_conversations]
initial_vectors = vectorizer.fit_transform(initial_prompts)
```

✓ 19.5s

# METHODOLOGY

-Analyze each conversation with cosine similarity

-Threshold determines if a follow-up is related to the initial prompt

- This score ranges from 0 to 1, where 0 indicates no similarity and 1 indicates identical text. It measures how similar the follow-up query is to the initial query in terms of the presence and frequency of words.
- If the score is below 0.2, the follow-up is classified as a 'new query'

not related to initial query

```
for i, conversation in enumerate(all_conversations):
    initial_vector = initial_vectors[i:i+1]
    follow_up_texts = [interaction['prompt'] for interaction in conversation[1:]]
    related_follow_ups = 0
    new_queries = 0

    if follow_up_texts:
        follow_up_vectors = vectorizer.transform(follow_up_texts)
        similarities = cosine_similarity(follow_up_vectors, initial_vector).flatten()

        threshold = 0.5
        related_follow_ups = (similarities >= threshold).sum()
        new_queries = (similarities < threshold).sum()

    conversation[0]['related_follow_ups'] = related_follow_ups
    conversation[0]['new_queries'] = new_queries
```

# METHODOLOGY

-Create a data frame for Statistical analysis

```
data = {  
    'initial_query': [],  
    'sentiment': [],  
    'subjectivity': [],  
    'related_follow_ups': [],  
    'new_queries': []  
}  
  
for conversation in all_conversations:  
    data['initial_query'].append(conversation[0]['prompt'])  
    data['sentiment'].append(conversation[0]['sentiment'])  
    data['subjectivity'].append(conversation[0]['subjectivity'])  
    data['related_follow_ups'].append(conversation[0]['related_follow_ups'])  
    data['new_queries'].append(conversation[0]['new_queries'])  
  
df = pd.DataFrame(data)
```

# METHODOLOGY

- Sentiment and subjectivity analysis of user's initial prompts using Text Blob

```
✓ for conversation in all_conversations:  
    initial_prompt = conversation[0]['prompt']  
    analysis = TextBlob(initial_prompt)  
    sentiment = analysis.sentiment.polarity  
    subjectivity = analysis.sentiment.subjectivity  
    conversation[0]['sentiment'] = sentiment  
    conversation[0]['subjectivity'] = subjectivity
```

# METHODOLOGY

-Calculate and print statistical results such as; average sentiment, subjectivity, and the number of follow-ups and new queries

```
|  
average_sentiment = df['sentiment'].mean()  
average_subjectivity = df['subjectivity'].mean()  
average_related_follow_ups = df['related_follow_ups'].mean()  
average_new_queries = df['new_queries'].mean()  
  
print(f"Average Sentiment: {average_sentiment}")  
print(f"Average Subjectivity: {average_subjectivity}")  
print(f"Average Number of Related Follow-Ups: {average_related_follow_ups}")  
print(f"Average Number of New Queries: {average_new_queries}")
```



# RESULTS

- Average Sentiment: 0.0412

(This suggests that developers' initial queries typically exhibit a neutral tone, hinting at a balanced, objective approach)

- Average Subjectivity: 0.312

(This score suggests that developers' queries are more objective, focusing more on factual information than personal opinions or emotions)

# RESULTS

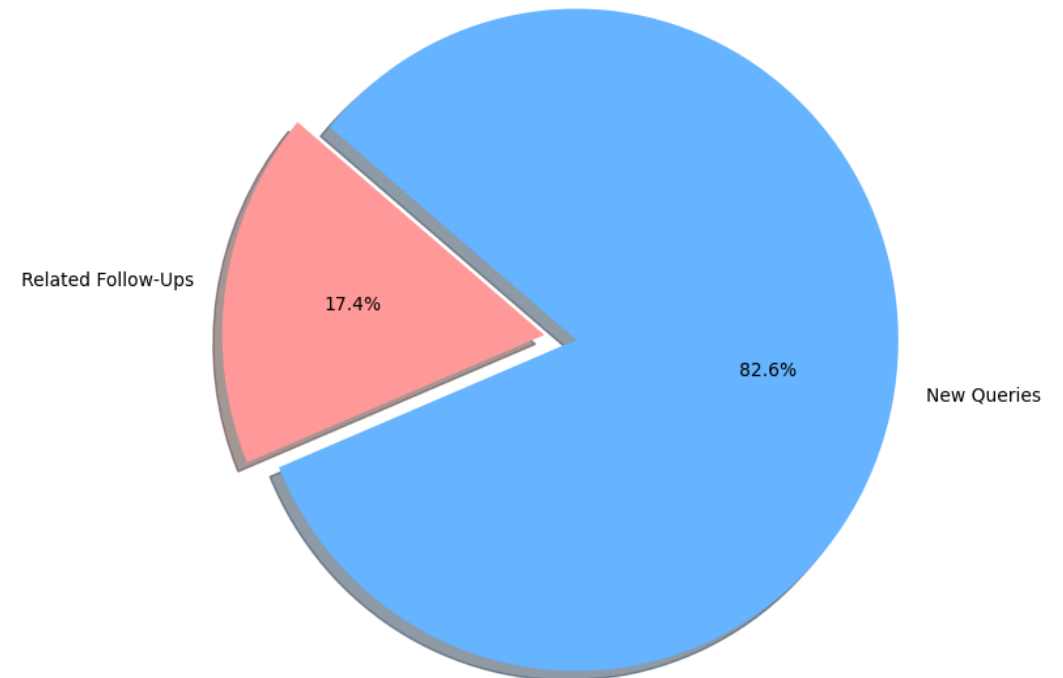
**Total Number of Related Follow-Ups: 12711**

**Total Number of New Queries: 60325**

**Threshold: 0.2**

**-Developers do receive comprehensive answers from ChatGPT on their initial query.**

Proportion of Related Follow-Ups vs. New Queries



# TESTING

- Manually go through randomly selected 50 prompt, answer and follow-up pairs 3 times.
- Results:
- 1: 25 New queries, 25 related follow-ups (new query: 50%, follow-up: 50%)
- 2: 32 New queries, 18 related follow-ups (new query: 84%, follow-up: 16%)
- 3: 34 New queries, 16 related follow-ups ( new query: 68%, follow up: 32%)

# HOW CAN IT BE IMPROVED

- Utilizing BERT or GPT to enhance contextual understanding
- Improved accuracy
- Dynamic word embeddings
- Better at Handling Longer Texts
- Computationally Expensive

# RESEARCH QUESTION 3

**Question:**

**How accurately can we predict the length of a conversation with ChatGPT based on the initial prompt and context provided?**

**Contributor: Arambhumi Reddy**

# METHODOLOGY

Incorporating additional features to capture contextual factors that may affect conversation length.

- The average language complexity of the prompts and responses, as calculated using the Flesch-Kincaid Grade Level, is approximately 7.24.
- The Flesch-Kincaid Grade Level is a measure of readability, with higher values indicating more complex language. A grade level of 7.24 suggests that the language complexity is roughly at the level of a seventh-grade student.

```
In [7]: import textstat

# Analyze language complexity using Flesch-Kincaid
language_complexity_scores = [textstat.flesch_kincaid_grade(prompt) for prompt in prompts]

# Print average language complexity score
average_language_complexity = sum(language_complexity_scores) / len(language_complexity_scores)
print("Average Language Complexity:", average_language_complexity)
```

Average Language Complexity: 7.242318938897568

# MODEL SELECTION AND COMPLEXITY

## INITIAL PROPOSAL – GPT-3 for feature extraction

- Transformer-based model “GPT-3” introduces a level of complexity due to its sophisticated architecture and large parameter count.

### Results:

```
array([[ -1.2587640e-01,   7.1513459e-02, -  
1.1070192e+00, ...,   
        1.7822880e-01,   1.2592289e-01,   
        [ 4.2092096e-02, -2.3145759e-01, -  
4.8710462e-01, ...,   
        3.3670199e-01]], dtype=float32)
```

## OPTIMISED PROPOSAL - GPT-2 for feature extraction (Lack of computational power to use GPT-3)

- The function that is defined in the project converts a piece of text into a feature vector that captures the contextual information encoded by a transformer model.

# MODEL SELECTION AND COMPLEXITY

## INITIAL PROPOSAL – RNN

- Feed the extracted features into an RNN for further sequence modeling.
- Use recurrent layers like LSTM to capture dependencies across the sequence.
- **REJECTED** due to lack of computational power.

OPTIMISED PROPOSAL – Tried Linear regression, Gradient Boosting and finally decided to go with Random Forest.

- Gradient Boosting took approx. 3 days to train so I decided not to use it for this project
- Linear regression and Random Forest took 7-9 hours each to train.



# RESULTS OF MODEL SELECTION

## Gradient Boosting with grid search for hyperparameters

### Linear Regression

```
# Step 1: Split data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(transformer_features, actual_lengths, test_size=0.2, random_state=42)

# Step 2: Train a linear regression model
regression_model = LinearRegression()
regression_model.fit(X_train, y_train)

# Step 3: Predict conversation lengths on the testing set
predictions = regression_model.predict(X_test)

# Step 4: Evaluate the model
mse = mean_squared_error(y_test, predictions)
mae = mean_absolute_error(y_test, predictions)

print("Mean Squared Error:", mse)
print("Mean Absolute Error:", mae)

# Step 5: Visualize predicted vs. actual lengths
plt.scatter(y_test, predictions)
plt.xlabel("Actual Conversation Lengths")
plt.ylabel("Predicted Conversation Lengths")
plt.title("Actual vs. Predicted Conversation Lengths")
plt.show()
```

Mean Squared Error: 182002.38117906003  
Mean Absolute Error: 196.0876155884503

```
# Define the parameter grid to search
param_grid = {
    'n_estimators': [50, 100, 150],
    'learning_rate': [0.01, 0.1, 0.2],
    'max_depth': [3, 4, 5]
}

# Initialize the Gradient Boosting Regressor
gb_regressor = GradientBoostingRegressor(random_state=42)

# Initialize GridSearchCV
grid_search = GridSearchCV(gb_regressor, param_grid, cv=5, scoring='neg_mean_squared_error')

# Fit the model
grid_search.fit(X_train, y_train)

# Get the best parameters
best_params = grid_search.best_params_

# Make predictions on the test set using the best model
best_model = grid_search.best_estimator_
predictions = best_model.predict(X_test)

# Evaluate the model
mse = mean_squared_error(y_test, predictions)
mae = mean_absolute_error(y_test, predictions)

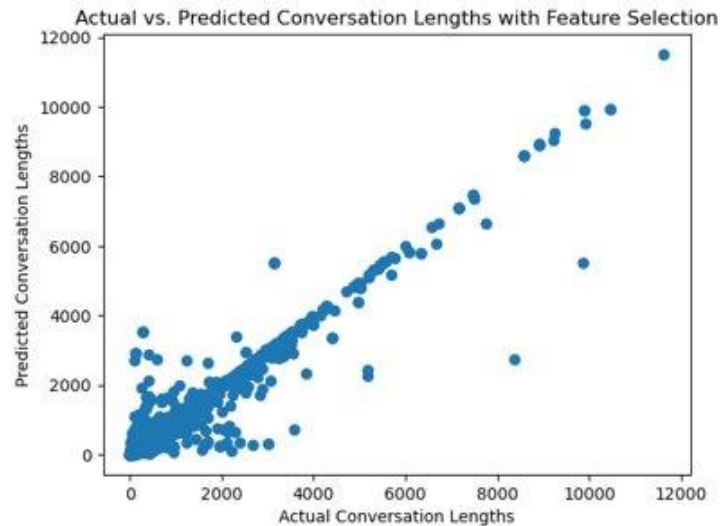
print("Best Hyperparameters:", best_params)
print("Mean Squared Error:", mse)
print("Mean Absolute Error:", mae)
```

Best Hyperparameters: {'learning\_rate': 0.2, 'max\_depth': 5, 'n\_estimators': 150}  
Mean Squared Error: 8087.1581393962315  
Mean Absolute Error: 29.044136561836254

# RESULTS OF MODEL SELECTION

## Random Forest

Mean Squared Error with Feature Selection: 20046.576784556666  
Mean Absolute Error with Feature Selection: 19.38681510048589



MAE of all  
models

Linear Regression : 196.08  
Gradient Boosting : 29.04  
Random Forest : 19.38

```
In [22]: #feature selection Testing phase

# Step 1: Split data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(transformer_features, actual_lengths, test_size=0.2, random_state=42)

# Step 2: Train a Random Forest Regressor model
rf_model = RandomForestRegressor(n_estimators=100, random_state=42)
rf_model.fit(X_train, y_train)

# Step 3: Perform feature selection
sfm = SelectFromModel(rf_model, threshold=0.3) # You can adjust the threshold
sfm.fit(X_train, y_train)
X_train_selected = sfm.transform(X_train)
X_test_selected = sfm.transform(X_test)

# Step 4: Train a model on the selected features
rf_model_selected = RandomForestRegressor(n_estimators=100, random_state=42)
rf_model_selected.fit(X_train_selected, y_train)

# Step 5: Predict conversation lengths on the testing set
predictions_selected = rf_model_selected.predict(X_test_selected)

# Step 6: Evaluate the model with selected features
mse_selected = mean_squared_error(y_test, predictions_selected)
mae_selected = mean_absolute_error(y_test, predictions_selected)

print("Mean Squared Error with Feature Selection:", mse_selected)
print("Mean Absolute Error with Feature Selection:", mae_selected)

# Visualization: Visualize predicted vs. actual lengths
plt.scatter(y_test, predictions_selected)
plt.xlabel("Actual Conversation Lengths")
plt.ylabel("Predicted Conversation Lengths")
plt.title("Actual vs. Predicted Conversation Lengths with Feature Selection")
plt.show()

Mean Squared Error with Feature Selection: 20046.576784556666
Mean Absolute Error with Feature Selection: 19.38681510048589
```

# MODEL FINALIZATION

**Random Forest** is the final model selected for my RQ.

## Feature Selection

Select from model : Assigns importance scores to each feature. Features with importance scores greater than or equal to the specified threshold are deemed important and retained, while those below the threshold are discarded. This feature selection approach aims to enhance model efficiency, mitigate overfitting, and potentially improve the interpretability of the Random Forest Regressor by focusing on the most influential features for the given predictive task.

```
# Step 3: Perform feature selection  
sfm = SelectFromModel(rf_model, threshold=0.3) # You can adjust the threshold  
sfm.fit(X_train, y_train)  
X_train_selected = sfm.transform(X_train)  
X_test_selected = sfm.transform(X_test)
```

# MODEL EVALUATION AND INTERPRETATION

## CUSTOM LOSS FUNCTION ( REJECTED )

- While designing custom loss functions can be crucial for certain models like Recurrent Neural Networks (RNNs) where the network architecture allows for such fine-tuning, it is not typically a common practice or a primary focus when working with Random Forest models.

## CROSS-VALIDATION

- we employ a randomized search with cross-validation to systematically tests different combinations of hyperparameter values, evaluating each with a 5-fold cross-validation setup.
- Goal is to identify the hyperparameter configuration that minimizes the mean squared error during cross-validation, ensuring the model's generalization across different subsets of the training data.

# MODEL EVALUATION AND HYPERPARAMETER TUNING

RandomizedSearchCV is a form of cross-validation. Specifically, it is a method of hyperparameter tuning that combines a randomized search over specified hyperparameter distributions with cross-validation.

It reduced the MSE and MAE values significantly.

|     | From     | TO      |
|-----|----------|---------|
| MSE | 20046.57 | 7224.86 |
| MAE | 19.38    | 12.70   |

R-squared: 0.9816188039140639

```
# Step 1: Split data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(transformer_features, actual_lengths, test_size=0.2, random_state=42)

# Step 2: Define the parameter distribution for random search
param_dist = {
    'n_estimators': [int(x) for x in np.linspace(start=50, stop=200, num=10)],
    'max_depth': [None] + [int(x) for x in np.linspace(10, 50, num=5)],
    'min_samples_split': [2, 5, 10, 20],
    'min_samples_leaf': [1, 2, 4, 8]
}

# Step 3: Create a Random Forest Regressor
rf_model = RandomForestRegressor(random_state=42)

# Step 4: Perform random search
random_search = RandomizedSearchCV(rf_model, param_distributions=param_dist, n_iter=10, cv=5, scoring='neg_mean_squared_error',
random_search.fit(X_train, y_train)

# Step 5: Get the best hyperparameters
best_params = random_search.best_params_
print("Best Hyperparameters:", best_params)

# Step 6: Train a model with the best hyperparameters
best_rf_model = RandomForestRegressor(**best_params, random_state=42)
best_rf_model.fit(X_train, y_train)

# Step 7: Predict conversation lengths on the testing set
predictions = best_rf_model.predict(X_test)

# Step 8: Evaluate the model
mse = mean_squared_error(y_test, predictions)
mae = mean_absolute_error(y_test, predictions)
r2 = r2_score(y_test, predictions)

print("Mean Squared Error:", mse)
print("Mean Absolute Error:", mae)
print("R-squared:", r2)

# Visualization: Visualize predicted vs. actual lengths
plt.scatter(y_test, predictions)
plt.xlabel("Actual Conversation Lengths")
plt.ylabel("Predicted Conversation Lengths")
plt.title("Actual vs. Predicted Conversation Lengths")
plt.show()

Best Hyperparameters: {'n_estimators': 116, 'min_samples_split': 5, 'min_samples_leaf': 1, 'max_depth': 20}
Mean Squared Error: 7224.860154068646
Mean Absolute Error: 12.702339654455734
R-squared: 0.9816188039140639
```

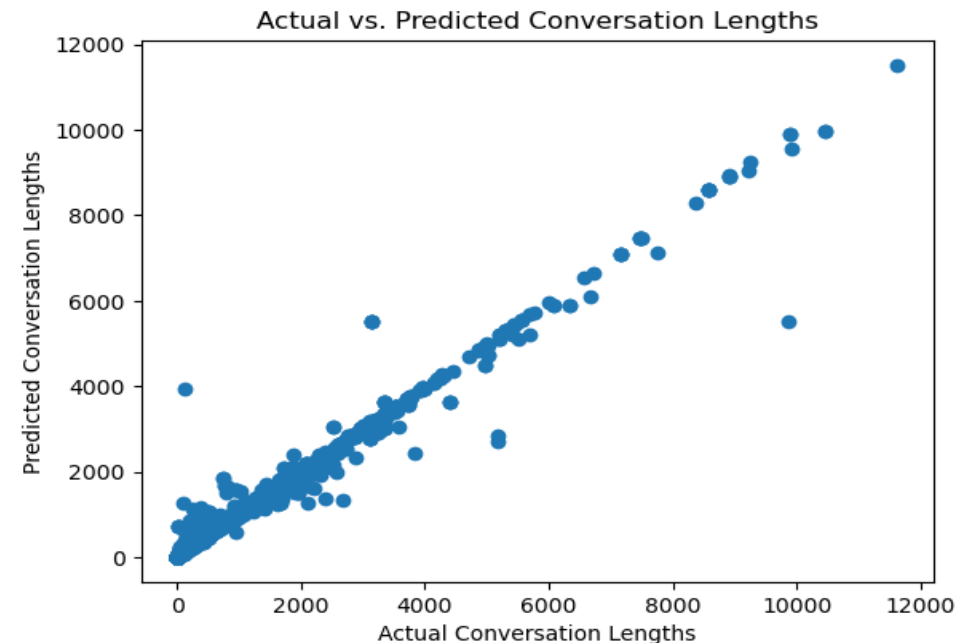
# FINAL TUNING

```
rf_model = RandomForestRegressor(n_estimators=113, max_depth=15, min_samples_split=2, min_samples_leaf=1, random_state=42)
rf_model.fit(X_train, y_train)
```

These metrics suggest that the model is performing well, as the R-squared value is close to 1, indicating a good fit. The scatter plot also helps visualize how well the predicted lengths align with the actual lengths.

An R-squared value of 0.984 indicates that our Random Forest model is explaining approximately 98.4% of the variance in the target variable (conversation lengths). This is a high R-squared value, suggesting that our model is fitting the data well and capturing most of the variability in the conversation lengths.

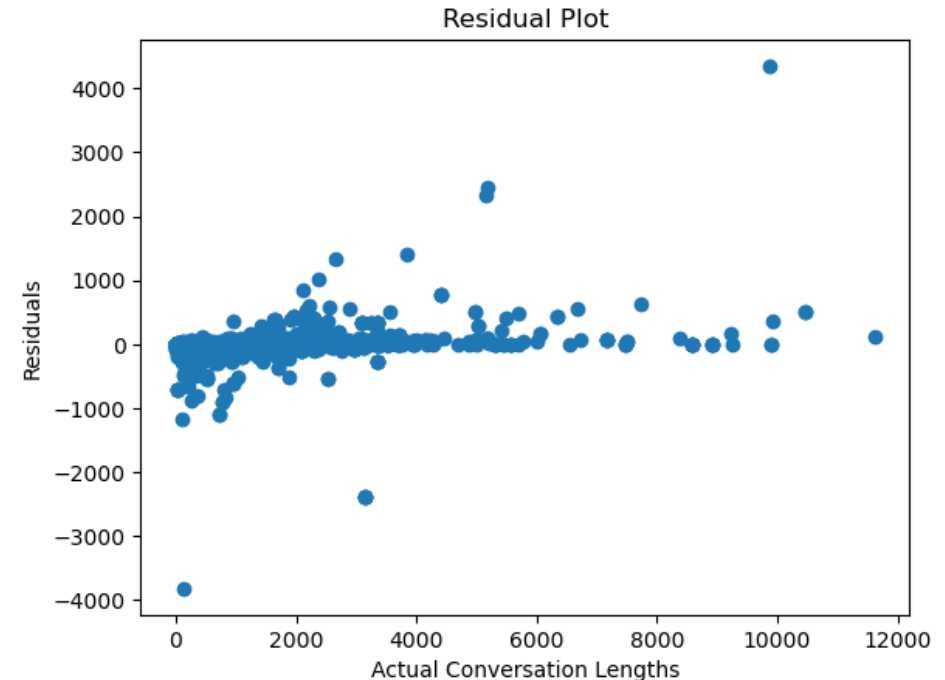
Mean Squared Error: 6255.77172229077  
Mean Absolute Error: 8.387659187717423  
R-squared: 0.9840843193855419



# RESULTS

The residual plot shows a random and even spread of points around the horizontal axis, it indicates that our model is capturing the underlying patterns in the data well. A good fit is characterized by residuals that do not exhibit any systematic patterns or trends.

```
# Create residual plot
residuals = y_test - predictions
plt.scatter(y_test, residuals)
plt.xlabel("Actual Conversation Lengths")
plt.ylabel("Residuals")
plt.title("Residual Plot")
plt.show()
```



# CONCLUSION

The model is capturing almost 98.4% of the patterns in conversation lengths. The predictions are very close to the actual conversation lengths.

This means we can trust the model to estimate conversation lengths accurately.

While the model is performing well, it's important to note that it's based on the data it was trained on. We need to be cautious about applying it to scenarios outside that scope.



**THANK  
YOU**