Answers

(you can find code below)

Question 3.1 : how would you measure diversity? E.g., entropy of each feature

For each answer, we calculate an entropy score by comparing its value in each column to the overall distribution of values in that column (which we computed earlier). This gives us a sense of how unique or common a particular row is for each feature. For Task 3.3, we can average the entropies of the answers for previously achieved max sematic similarity

Question 3.2 : what is the max readability and semantic similarity independent of the diversity?

To identify the best sets of answers based on high semantic similarity to the ground truth, we build a decision tree built on SPICE score as a custom split criterion.

- At each step, we attempt all possible feature splits to partition the data. -For each potential split, we summarize the explanations in the left split.
- We then compute the SPICE similarity between the summarized explanations and the ground truth.
- The split that maximizes the SPICE similarity is selected.
- The decision tree will have a maximum depth of 4 to limit the number of recursive steps.
- The tree will stop splitting when no valid splits remain (i.e., no further improvement in SPICE similarity).
- The process recursively splits the data at each node, selecting the best split based on SPICE similarity.
- At each level, we build tree nodes using the split that results in the highest SPICE similarity, focusing on sets of answers that are most semantically similar to the ground truth. This approach ensures that we choose sets of answers that exhibit high semantic similarity to the ground truth, optimizing for the most relevant and accurate explanations.

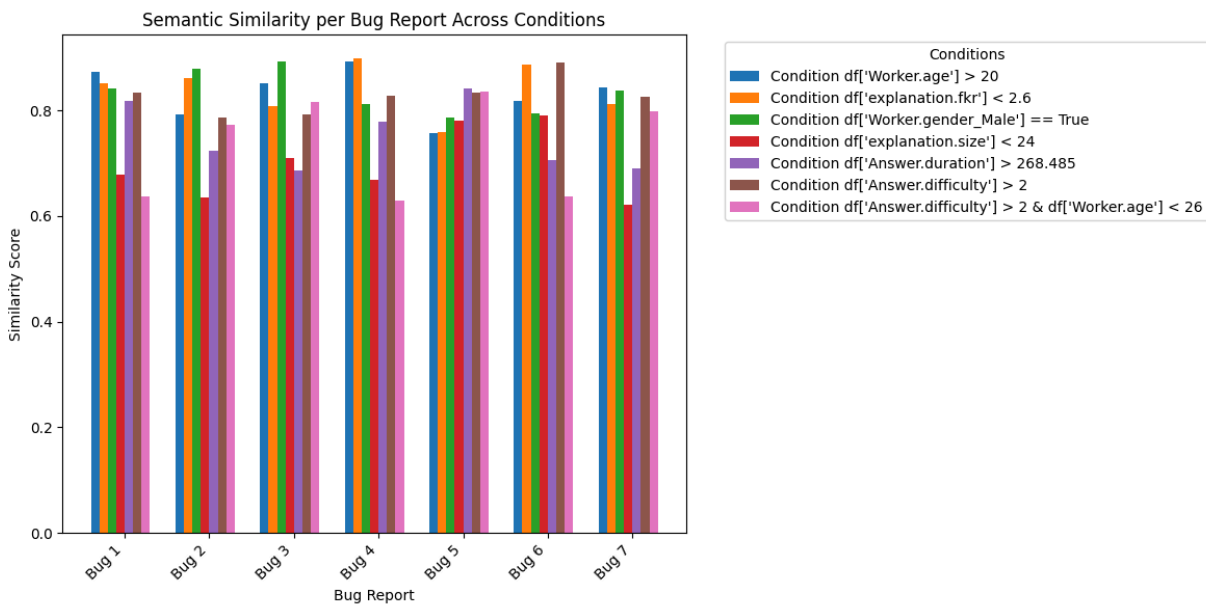
We get the following splits:

- Split on Answer.duration at 88.588

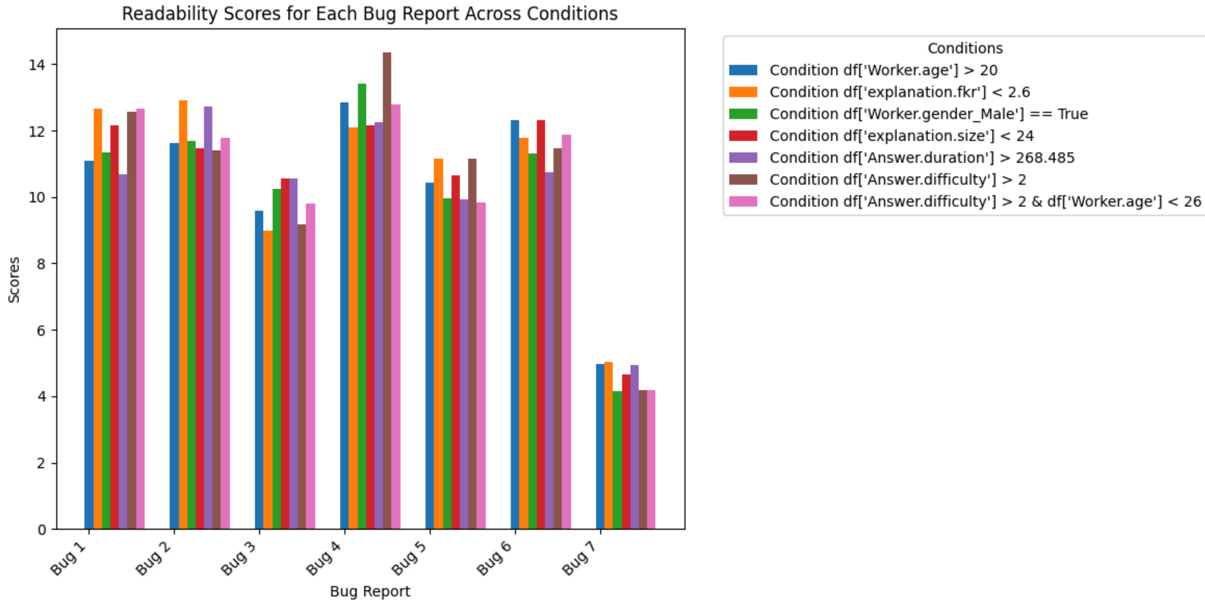
- Split on Answer.difficulty at 3
- Split on Worker.age at 20
- Split on explanation.fkr at 2.6
- Split on Worker.gender_Male at True
- plit on explanation.size at 24
- Split on Answer.duration at 268.485

The answers from people over 20 and the answers with high readibility generate more similar results to our ground truth than the other conditions. They are also more readable. Overall, there aren't huge differences among these splits, we chose based on the Decision Tree Strategy using the spice score, which also captures semantic similarity

In [ ]:



In [ ]:

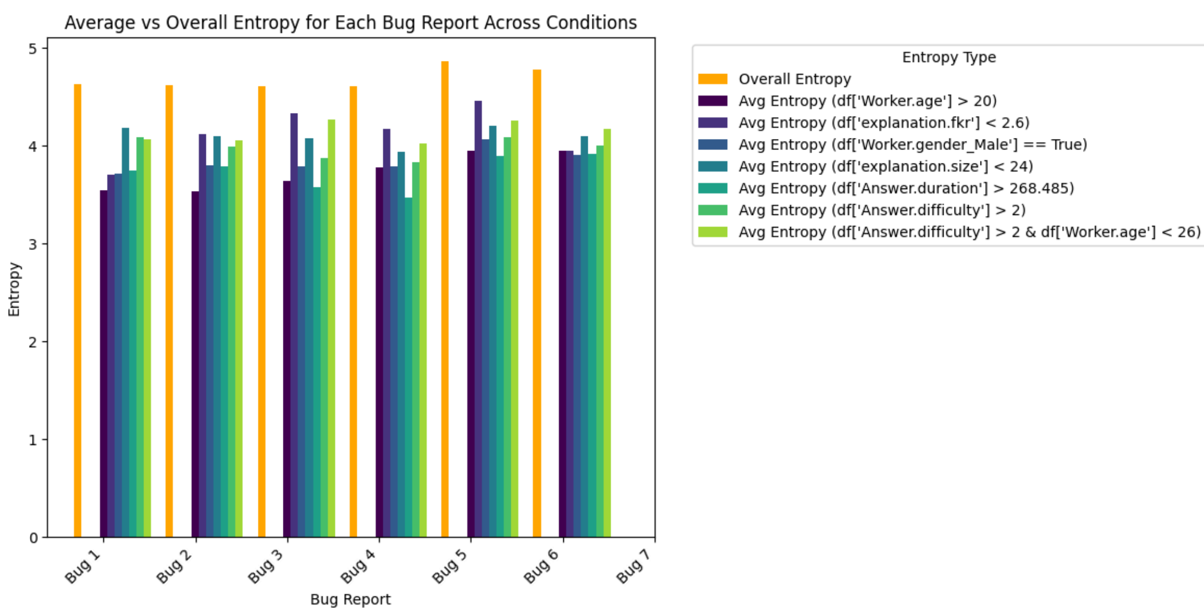


Question 3.3 : what is the max diversity for (previously achieved) max semantic similarity ?

Overall

The max diversity is for the split of workers over 20 , but the second best diversity answers with a diffulty greater than 2. It has a also a high semantic similarity, but less readibilty (see visualizations above). Overall, the differences are still small.

In [ ]:



Code

Data preparation

We transform the categroical data into numerical values through binarizations. Also we map the Worker.origin to a new feature, which shows whether the worker is a native speaker. Also, we introduce a new feature entropy, which shows the diversity of the features within a row.

Read Raw Data

In [ ]:
```
import pandas as pd
from nltk.tokenize import word_tokenize
import nltk
from sklearn.preprocessing import LabelEncoder
from sklearn.preprocessing import MultiLabelBinarizer

nltk.download("punkt_tab")
```

In [ ]:
```
import pandas as pd

df = pd.read_csv("answerList_data.csv")
df = df[df["GroundTruth"] == 1].copy()
```

ground truth consolidated explanation from task 2

In [4]:
```
import json

with open("ground_truths.json", "r") as file:
    gt_expl = json.load(file)
```

Complexity scores

- Type token ratio
- Flesh kincaid readability

Explanation size

- word length

In [5]:
```
explanations = list(df["Answer.explanation"])
explanations = [str(e) for e in explanations]
ttr_scores = [
    len(list(set(word_tokenize(e)))) / len(word_tokenize(e)) for e in explar
]

df["explanation.ttr"] = ttr_scores
```

In [6]:
```
import textstat

df["explanation.fkr"] = df["Answer.explanation"].apply(
    lambda a: textstat.flesch_kincaid_grade(str(a))
)
```

```
In [7]: df["explanation.size"] = df["Answer.explanation"].apply(
            lambda a: len(word_tokenize(str(a)))
        )
```

Clean Data

```
In [8]: country_mapping = {
            "USA": "United States",
            "US": "United States",
            "U.S.": "United States",
            "U.S.A.": "United States",
            "U.S": "United States",
            "usa": "United States",
            "Usa": "United States",
            "united states": "United States",
            "United States of America": "United States",
            "Unites States of America": "United States",
            "Canada": "Canada",
            "canada": "Canada",
            "India": "India",
            "INDIA": "India",
            "india": "India",
            "indian": "India",
            "uk": "United Kingdom",
        }
        df["Worker.country"] = (
            df["Worker.country"].map(country_mapping).fillna(df["Worker.country"])
        )
        native_speaking_countries = ["United States", "India", "Canada", "United Kir
        df["Worker.native_speaker"] = df["Worker.country"].apply(
            lambda x: 1 if x in native_speaking_countries else 0
        )
```

```
In [ ]: columns = [
            "Worker.score",
            "Worker.profession",
            "Worker.yearsOfExperience",
            "Worker.age",
            "Worker.gender",
            "Worker.native_speaker",
            "Answer.duration",
            "Code.complexity",
            "explanation.size",
            "Answer.confidence",
            "Answer.difficulty",
            "explanation.ttr",
            "explanation.fkr",
            "Answer.explanation",
            "FailingMethod",
        ]

        df_selected = df[columns].copy()

        label_cols = ["Answer.explanation", "FailingMethod"]
```

```
feature_cols = list(set(df_selected.columns) - set(label_cols))
```

In [10]:
```
gt_columns = ["Answer.explanation", "FailingMethod"]
df_gt = df[gt_columns].copy()
```

In [11]:
```
df_selected = df[columns].copy()
```

calculate diversity for features

In [ ]:
```
import numpy as np
import pandas as pd


def calculate_entropy(series):
    value_counts = series.value_counts(normalize=True)
    entropy = -np.sum(value_counts * np.log2(value_counts))
    return entropy


entropy_values = df_selected[feature_cols].apply(calculate_entropy)
```

In [14]:
```
def calculate_row_entropy(row, feature_entropy):
    row_entropy = 0
    for feature, value in row.items():
        column_entropy = feature_entropy.get(feature, 0)

        value_frequency = (
            df_selected[feature].value_counts(normalize=True).get(value, 0)
        )

        if value_frequency > 0:
            row_entropy += -value_frequency * np.log2(value_frequency)

    return row_entropy


df_selected["entropy"] = df_selected.apply(
    lambda row: calculate_row_entropy(row, entropy_values), axis=1
)
```

In [15]:
```
entropy_range = df_selected["entropy"].min(), df_selected["entropy"].max()
entropy_range
```

Out[15]:
```
(2.9924320968187095, 4.945120081851421)
```

Choose for each Bug report diverse answers

For each row, we calculate a "row entropy" score by comparing its value in each column to the overall distribution of values in that column (which we computed earlier). This gives us a sense of how unique or common a particular row is for each feature. For each feature in the row, we look up the frequency of that value in the dataset (value_frequency) and calculate the entropy contribution based on how "spread out" the values are in that feature.

transform categorical data

In [ ]:
```
df_categorical = df_selected[categorical_columns]

df_encoded = pd.get_dummies(df_categorical)

df_transformed = pd.concat(
    [df_selected.drop(columns=df_categorical.columns), df_encoded], axis=1
)

df_transformed.columns
```

LLM for consolidating explanations

Assistant for LLama

The answers consolidated with GPT are retreived with ChatGPT

In [ ]:
```
import os
from typing import Dict, List
from huggingface_hub import login
from openai import OpenAI
from transformers import AutoTokenizer, PreTrainedTokenizerFast
from omegaconf import DictConfig

MODEL_NAME = "neuralmagic/Meta-Llama-3.1-405B-Instruct-quantized.w4a16"
ANSWER_TOKEN_LENGTH = 2048
MODEL_TEMPERATURE = 0.2


class Assistant:
    model_name: str
    temperature: float
    _client: OpenAI
    _system_message: Dict[str, str]
    _messages: List[Dict[str, str]]
    _tokenizer: PreTrainedTokenizerFast

    def __init__(self, model_name: str, model_temperature: float):
        login(token="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")
        self._client = OpenAI(base_url=os.getenv("VLLM_BASE_URL"))
        self._system_message = {
            "role": "system",
            "content": (
                "You are an AI assistant helping to summarize explanations f
                "Provide concise, clear explanations based on input."
            ),
        }
        self._messages = []
        self.temperature = model_temperature
```

```python
        self.model_name = model_name
        self._tokenizer = AutoTokenizer.from_pretrained(model_name)
        assert (
            self._tokenizer != False
        ), f"Something went wrong when fetching the default tokenizer for mo

    def _all_messages(self):
        return [self._system_message] + self._messages

    def _tokenized_messages(self):
        return self._tokenizer.apply_chat_template(self._all_messages(), tok

    def generate_answer(self, prompt: str) -> str:
        self._messages += [
            {"role": "user", "content": prompt},
        ]
        num_tokens = len(self._tokenized_messages())
        while num_tokens > ANSWER_TOKEN_LENGTH:
            self._messages.pop(0)
            num_tokens = len(self._tokenized_messages())

        completion = self._client.completions.create(
            model=self.model_name,
            max_tokens=ANSWER_TOKEN_LENGTH,
            prompt=self._tokenized_messages(),
            temperature=self.temperature,
        )
        answer = completion.choices.pop().text
        self._messages += [
            {
                "role": "assistant",
                "content": answer,
            }
        ]

        return answer
```

In [19]:

```python
import pandas as pd


def summarize_explanations_with_assistant(
    explanations, assistant: Assistant, prompt_template: str
):
    """
    Summarizes explanations using the Assistant class.

    Args:
        grouped_explanations (dict): Grouped explanations by bug ID.
        assistant (Assistant): An instance of the Assistant class.
        prompt_template (str): Prompt template with a placeholder for explar

    Returns:
        dict: Summarized explanations for each bug ID.
    """
    summarized_answers = {}
    combined_text = "\n".join(explanations)
```

```
    prompt = prompt_template.format(combined_text=combined_text)

    summary = assistant.generate_answer(prompt)

    return summary
```

Prompt

we chose the generated Chain of Thought Prompt, as it is most promising (see mini project 2)

In [20]:
```
CHAT_COT_PROMPT = (
    "You are a highly skilled AI assistant designed to analyze and consolida
    "Your task is to generate a single explanation that minimizes redundancy
    "necessary for a developer to fix the issue. Below are some explanations
    "Explanations:\n{combined_text}\n\n"
    "Requirements for the Explanation:\n"
    "1. Describe how the program works in the context of the bug.\n"
    "2. Explain how the failure occurs, including specific conditions or ste
    "3. Identify and describe the root cause of the problem in the code.\n"
    "4. Make the explanation concise and free of unnecessary repetition. Ens
    "Step-by-step reasoning to consolidate the explanation:\n"
    "1. Analyze each explanation to identify overlapping information and uni
    "2. Determine the most relevant points about how the program works, why
    "3. Synthesize these points into a cohesive explanation that is concise
    "Build only a summarizing answer from that"
)
```

Choose Answers that have high semantic similarity to ground truth

- Custom Split Criterion: At each step:
- Try all feature splits.
- Summarize left-split explanations.
- Compute SPICE similarity with ground truth.
- Select the split that maximizes SPICE similarity.
- Stopping Criteria:
- Max Depth = 4.
- Stop if no valid splits exist.
- Recursive Splitting: Build tree nodes using the best split.

In [ ]:
```
import numpy as np
import pandas as pd
from sklearn.tree import DecisionTreeClassifier
from pycocoevalcap.spice.spice import Spice


# Custom Decision Tree using SPICE similarity
class SpiceDecisionTree(DecisionTreeClassifier):
    def __init__(self, max_depth=4, assistant=None, ground_truth=None):
```

```python
        """
        Custom Decision Tree using SPICE similarity as the splitting criteri

        Parameters:
        - max_depth: Maximum depth of the tree (default = 4)
        - assistant: AI assistant for summarizing explanations
        - ground_truth: The ground truth explanation for similarity comparis
        """
        super().__init__(max_depth=max_depth)
        self.assistant = assistant
        self.ground_truth = ground_truth
        self.tree_ = {}  # Custom tree structure
        self.spice_evaluator = Spice()
        self.max_depth = 4

    def _find_best_split(self, data, feature_columns):
        """Find the best feature to split on based on SPICE similarity."""
        best_feature = None
        best_score = -1
        best_split = None
        best_val = None

        for feature in feature_columns:
            unique_values = data[feature].unique()

            for val in unique_values:
                left_split = data[data[feature] <= val]
                right_split = data[data[feature] > val]

                if len(left_split) == 0 or len(right_split) == 0:
                    continue  # Skip invalid splits

                # Summarize left-side explanations
                summary = summarize_explanations_with_assistant(
                    list(left_split["Answer.explanation"]),
                    self.assistant,
                    CHAT_COT_PROMPT,
                )

                # Compute SPICE similarity
                spice_score, _ = self.spice_evaluator.compute_score(
                    {0: [self.ground_truth]}, {0: [summary]}
                )

                summary = summarize_explanations_with_assistant(
                    list(right_split["Answer.explanation"]),
                    self.assistant,
                    CHAT_COT_PROMPT,
                )

                # Compute SPICE similarity
                second_spice_score, _ = self.spice_evaluator.compute_score(
                    {0: [self.ground_truth]}, {0: [summary]}
                )

                # Choose the best split based on highest SPICE similarity
```

```python
                if spice_score > best_score or second_spice_score > best_sco
                    best_val = val
                    best_score = spice_score
                    best_feature = feature
                    best_split = (left_split, right_split)

        return best_feature, best_score, best_split, best_val

    def _build_tree(self, data, feature_columns, depth=0):
        """Recursively build the decision tree."""
        if depth >= self.max_depth or len(data) < 2:
            return {
                "summary": summarize_explanations_with_assistant(
                    data["Answer.explanation"], self.assistant, CHAT_COT_PRO
                )
            }

        # Find the best feature split
        best_feature, best_score, best_split, best_val = self._find_best_spl
            data, feature_columns
        )

        if best_feature is None:
            return {
                "summary": summarize_explanations_with_assistant(
                    data["Answer.explanation"], self.assistant, CHAT_COT_PRO
                )
            }

        left_data, right_data = best_split

        return {
            "feature": best_feature,
            "spice_score": best_score,
            "left": self._build_tree(left_data, feature_columns, depth + 1),
            "right": self._build_tree(right_data, feature_columns, depth + 1
            "val": best_val,
        }

    def fit(self, X, y=None):
        """Train the decision tree on filtered data."""
        feature_columns = [
            col
            for col in X.columns
            if col not in ["Answer.explanation", "FailingMethod", "entropy"]
        ]
        self.tree_ = self._build_tree(X, feature_columns)
        return self

    def print_tree(self, tree=None, depth=0):
        """Print the decision tree structure."""
        if tree is None:
            tree = self.tree_

        if "feature" in tree:
            print(
```

```python
                f"{'  ' * depth}- Split on {tree['feature']}, at {tree['val'
            )
            self.print_tree(tree["left"], depth + 1)
            self.print_tree(tree["right"], depth + 1)
        else:
            print(f"{'  ' * depth}- Leaf: {tree['summary']}")


# Load and filter dataset for `FailingMethod == "HIT04_7"`
df_filtered = df_transformed[df_transformed["FailingMethod"] == "HIT04_7"].c
df_filtered = df_filtered.drop(columns=["FailingMethod"])

df_expl_filtered = df_gt[df_transformed["FailingMethod"] == "HIT04_7"].copy(

# Initialize and train the custom SPICE-based decision tree
assistant = Assistant(model_name=MODEL_NAME, model_temperature=MODEL_TEMPERA
ground_truth = gt_expl["HIT04_7"]  # Replace with actual GT

tree = SpiceDecisionTree(max_depth=4, assistant=assistant, ground_truth=grou
print(df_filtered.columns)
tree.fit(df_filtered)

# Print the resulting tree
# tree.print_tree()
```

In [29]:
```python
tree.print_tree()
```

```
- Split on Answer.duration, at 88.588,  (SPICE Score: 0.2105263157894737)
  - Split on Answer.difficulty, at 3,  (SPICE Score: 0.2)
    - Split on Worker.age, at 20,  (SPICE Score: 0.20408163265306126)
      - Leaf: The provided explanation only confirms that the variable is de
fined correctly as a parameter of the getPaint method, but lacks details on
the program's behavior, failure occurrence, and root cause of the problem, r
equiring additional information to diagnose and fix the issue.
      - Leaf: No information is provided to describe the program's behavior,
failure occurrence, or root cause of the problem, making it impossible to ge
nerate a meaningful explanation without further investigation.
    - Leaf: The program calls the Color constructor with three float paramet
ers, allowing values between 0.0 and 1.0. However, the explanation lacks det
ails on the failure occurrence and root cause of the problem, requiring addi
tional information to understand the issue and provide a comprehensive solut
ion.
  - Split on explanation.fkr, at 2.6,  (SPICE Score: 0.1724137931034483)
    - Leaf: The provided explanation is incomplete, as it only mentions the
data type of the variable "value" and the usage of Math.max() and Math.min()
functions. There is no information about the bug, the failure, or the root c
ause of the problem. Therefore, it is not possible to generate a comprehensi
ve explanation that meets the requirements. Additional information is needed
to provide a clear and concise explanation of the issue.
    - Split on Worker.gender_Male, at False,  (SPICE Score: 0.13333333333333
333)
      - Split on explanation.size, at 24,  (SPICE Score: 0.1276595744680851)
        - Leaf: The program throws an `IllegalArgumentException` when a meth
od receives an invalid argument, indicating a failure in input validation. T
his exception occurs when the input is outside the expected range or is othe
rwise inappropriate. The root cause is the passing of an illegal argument, w
hich needs to be corrected to resolve the issue.
        - Leaf: The program's behavior is unclear when handling negative num
bers due to undefined input range. The issue cannot be resolved without revi
ewing the definitions of `this.lowerBound` and `this.upperBound`, which are
necessary to determine the root cause of the problem.
      - Split on Answer.duration, at 268.485,  (SPICE Score: 0.15625)
        - Leaf: The program attempts to create a color with a value that sho
uld be between 0.0 and 1.0. However, the code fails to sanitize negative num
bers, and the value -0.5 is not checked properly. Although the argument valu
e is checked against the lowerBound and upperBound variables, the resulting
variable "v" is not used, allowing the invalid value to pass through. As a r
esult, a negative integer value is produced, which is outside the expected 0
to 255 range for a color. The root cause is the failure to use the sanitized
variable "v" in the code, allowing invalid values to cause the error.
        - Leaf: The program attempts to create a Color object with an intege
r value "g" calculated from the "value" variable, which should be between 0
and 255. However, the calculation can result in a negative value for "g" whe
n "value" is outside the valid range, such as the provided -0.5. The failure
occurs because the variable "v", which is defined to ensure the value is wit
hin the valid range, is not used on line 117. Instead, the unsanitized "valu
e" is used, resulting in a negative integer value for "g", which is outside
the expected 0 to 255 range for a color. The root cause is the incorrect use
of "value" instead of "v", which fails to sanitize the input and causes the
error.
```

we chose the set of programers with the following , since the Decision tree based on the spice score is highest for this group we now want to compare the text similarity between gt and this subset of answers for each bug report based on cosine similarity of the embedded sequences

Similarity

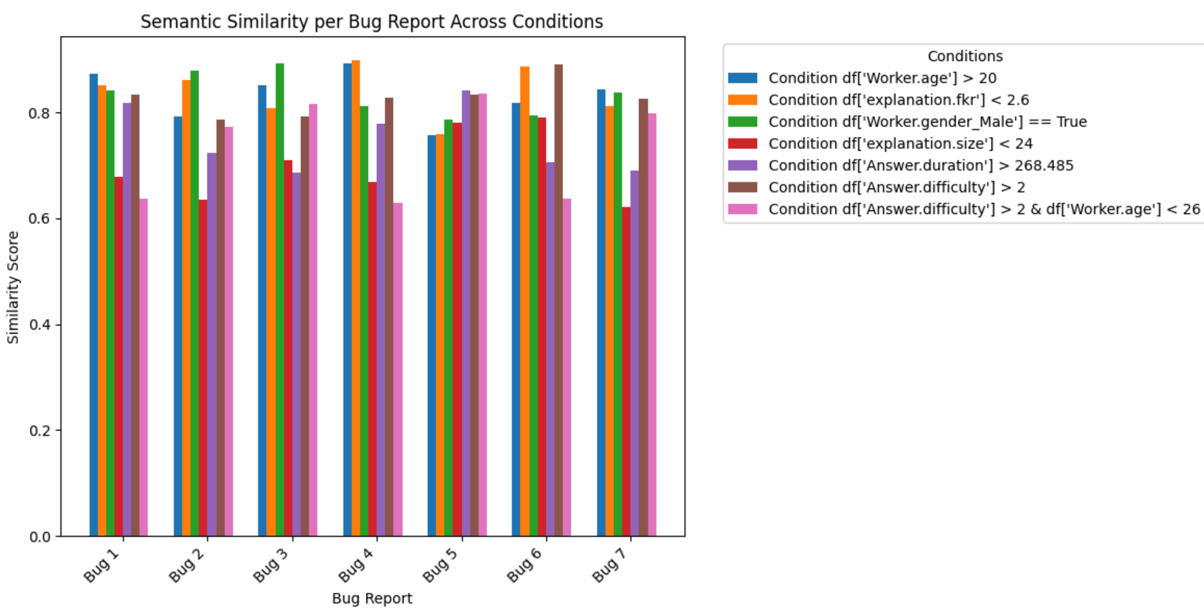
In [88]:
```
from sentence_transformers import SentenceTransformer
from sklearn.metrics.pairwise import cosine_similarity

model = SentenceTransformer("sentence-transformers/all-MiniLM-L6-v2")


def sem_similarity(ground_truth, new):
    embeddings1 = model.encode(ground_truth)
    embeddings2 = model.encode(new)
    return cosine_similarity(embeddings1.reshape(1, -1), embeddings2.reshape
```

In [ ]:
```
from collections import defaultdict

# Step 1: Group explanations by bug report ID
bug_explanations = defaultdict(list)


def get_summary_for_split(df, condition):
    # Apply the condition to filter the DataFrame
    df_split = df[condition(df)]

    for _, row in df_split.iterrows():
        bug_explanations[row["FailingMethod"]].append(row["Answer.explanatic

    # Step 2: Summarize explanations for each bug report
    summarized_explanations = {
        bug_id: summarize_explanations_with_assistant(
            explanations, assistant, CHAT_COT_PROMPT
        )
        for bug_id, explanations in bug_explanations.items()
    }

    return summarized_explanations


df_filtered = df_transformed.copy()
print(df_filtered.columns)

# conditions derived from decision tree
conditions = [
    (
        "df['Worker.age'] > 20",
        lambda df: (df["Worker.age"] > 20)
        & (df["Answer.difficulty"] < 3)
        & (df["Answer.duration"] > 88.588),
    ),
    (
```

```
        "df['explanation.fkr'] < 2.6",
        lambda df: df["explanation.fkr"] < 2.6,
    ),
    (
        "df['Worker.gender_Male'] == True",
        lambda df: (df["Worker.gender_Male"] == True),
    ),
    (
        "df['explanation.size'] < 24",
        lambda df: (df["explanation.size"] < 24),
    ),
    (
        "df['Answer.duration'] > 268.485",
        lambda df: (df["Answer.duration"] > 268.485),
    ),
    (
        "df['Answer.difficulty'] > 2",
        lambda df: df["Answer.difficulty"] > 2,
    ),
    (
        "df['Answer.difficulty'] > 2 & df['Worker.age'] < 26",
        lambda df: (df["Answer.difficulty"] > 2) & (df["Worker.age"] < 26),
    ),
]


import matplotlib.pyplot as plt
import seaborn as sns
import random

# Define colors for each condition
condition_colors = {
    "df['Worker.age'] > 20": "blue",
    "df['explanation.fkr'] < 2.6": "red",
    "df['Worker.gender_Male'] == True": "green",
    "df['explanation.size'] < 24": "purple",
    "df['Answer.duration'] > 268.485": "orange",
    "df['Answer.difficulty'] > 2": "brown",
    "df['Answer.difficulty'] > 2 & df['Worker.age'] < 26": "pink",
}

bug_reports = []
scores = []
colors = []

for condition in conditions:
    condition_label = condition[0]
    summarized_explanations = get_summary_for_split(df_filtered, condition[1

    for bug_id, summarized_explanation in summarized_explanations.items():
        gt_explanation = gt_expl.get(bug_id, "No ground truth available")
        similarity = sem_similarity(summarized_explanation, gt_explanation)
        bug_reports.append(bug_id)
        scores.append(similarity)
        colors.append(condition_colors[condition_label])
num_conditions = 7
```

```
num_bug_reports = len(scores) // num_conditions
scores = np.array(scores).reshape(num_conditions, num_bug_reports)


bug_reports = [f"Bug {i+1}" for i in range(num_bug_reports)]

condition_labels = [f"Condition {c[0]}" for c in conditions]

bar_width = 0.1
x = np.arange(num_bug_reports)
```

In [58]:
```
plt.figure(figsize=(12, 6))

for i in range(num_conditions):
    plt.bar(x + i * bar_width, scores[i], width=bar_width, label=condition_l

plt.xticks(x + (num_conditions / 2) * bar_width, bug_reports, rotation=45, h

plt.xlabel("Bug Report")
plt.ylabel("Similarity Score")
plt.title("Semantic Similarity per Bug Report Across Conditions")

plt.legend(title="Conditions", bbox_to_anchor=(1.05, 1), loc="upper left")

plt.tight_layout()

plt.show()
```

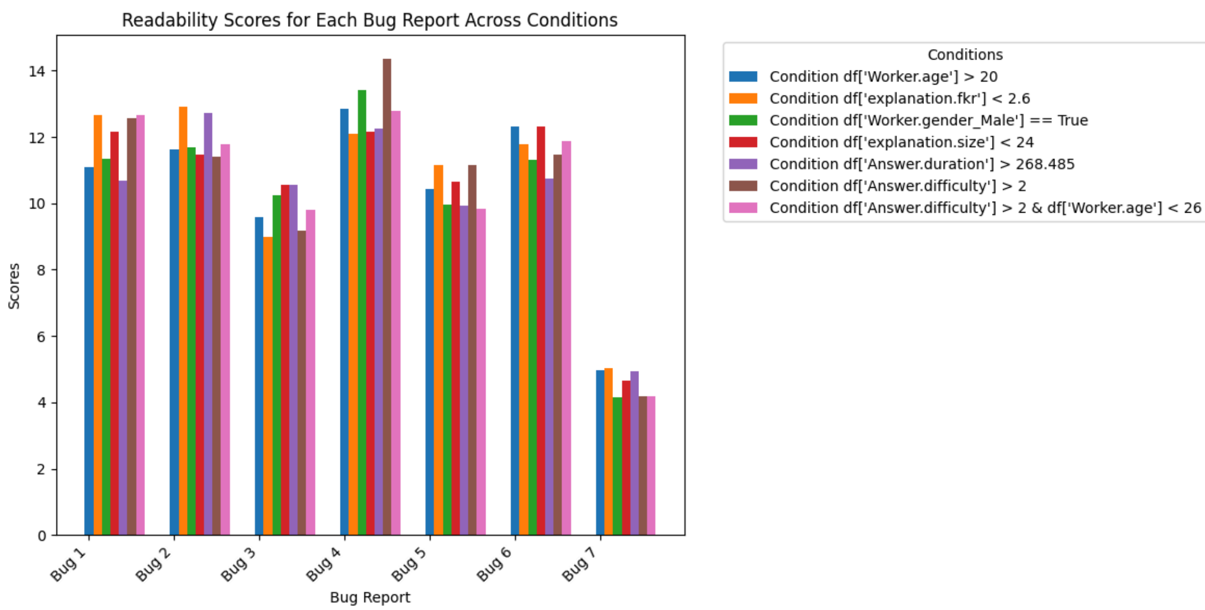


In [ ]:
```
condition_scores = defaultdict(list)

for bug_id, explanation in summarized_explanations.items():
    for condition in conditions:
        condition_name, condition_func = condition
        fkr_score = textstat.flesch_kincaid_grade(explanation)

        # Store results in a dictionary with condition name
        condition_scores[condition_name].append(fkr_score)
```

```
print(condition_scores)

num_conditions = len(condition_scores)
bug_reports = [f"Bug {i+1}" for i in range(num_bug_reports)]
x = np.arange(num_bug_reports)
```

In [79]:

```
plt.figure(figsize=(12, 6))

bar_width = 0.1

for i, (condition, scores) in enumerate(condition_scores.items()):
    readability_scores = scores
    plt.bar(
        x + i * bar_width,
        readability_scores,
        width=bar_width,
        label=condition_labels[i],
    )

plt.xlabel("Bug Report")
plt.ylabel("Scores")
plt.title("Readability Scores for Each Bug Report Across Conditions")
plt.xticks(x, bug_reports, rotation=45, ha="right")
plt.legend(title="Conditions", bbox_to_anchor=(1.05, 1), loc="upper left")
plt.tight_layout()
plt.show()
```



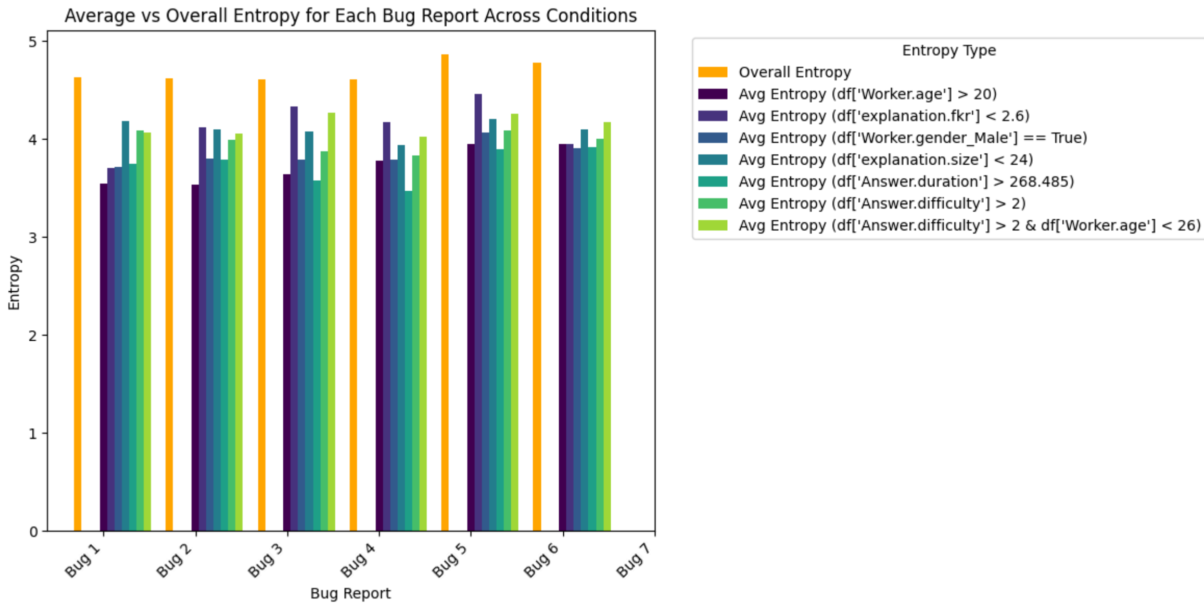
The answers from people over 20 and the answers with high readibility generate more similar results to our gt than the other conditions. They are also more readable.

Diversity of Answers coming from the Decisiontree splits

In [ ]:

```python
import matplotlib.pyplot as plt
import numpy as np

# Initialize variables
average_entropy = defaultdict(list)
overall_entropy = []

# Iterate through each bug report
for bug_id, explanations in bug_explanations.items():
    # Filter the DataFrame based on the condition for this bug
    filtered_df = df_filtered[df_filtered["FailingMethod"] == bug_id]

    # Calculate the overall entropy (unfiltered) for each bug report
    overall_entropy_value = filtered_df["entropy"].mean()
    overall_entropy.append(overall_entropy_value)

    # Iterate over the conditions and calculate average entropy for each cor
    for condition_name, condition_func in conditions:
        filtered_df_condition = filtered_df[condition_func(filtered_df)]

        # Calculate the average entropy under this condition
        avg_entropy = filtered_df_condition["entropy"].mean()
        average_entropy[condition_name].append(avg_entropy)

# Prepare for plotting
num_bug_reports = len(bug_explanations)
x = np.arange(num_bug_reports)

# Set bar width for each condition
width = 0.08
num_conditions = len(average_entropy)

# Create a bar chart for both average and overall entropy
plt.figure(figsize=(12, 6))

# Plot overall entropy
plt.bar(
    x - (width * num_conditions / 2),
    overall_entropy,
    width=width,
    label="Overall Entropy",
    color="orange",
)

# Plot average entropy for each condition
for i, (condition_name, avg_entropies) in enumerate(average_entropy.items())
    plt.bar(
        x + (i * width),
        avg_entropies,
        width=width,
        label=f"Avg Entropy ({condition_name})",
        color=plt.cm.viridis(i / num_conditions),
    )

# Labels and Title
```

```python
plt.xlabel("Bug Report")
plt.ylabel("Entropy")
plt.title("Average vs Overall Entropy for Each Bug Report Across Conditions"
plt.xticks(x, bug_reports, rotation=45, ha="right")
plt.legend(title="Entropy Type", bbox_to_anchor=(1.05, 1), loc="upper left")
plt.tight_layout()

# Show the plot
plt.show()
```



The answers from people over 20 have the most diverse values for the feature