IR Project Report

AmirMohammad Ebrahiminasab 610301101

February 12, 2025

List Report

1. Step 1
2. Step 2
3. Step 3
4. Step 4
5. Step 5
6. Step 6
7. Step 7
8. Step 8
9. Step 9
10. Step 10

1 Step 1: Load and Inspect the Dataset

In this step I loaded the data using the code snippet below and here is the overall structure of our dataset:

```python
import pandas as pd

df = pd.read_excel('sample_data/MeQSum_ACL2019_BenAbacha_Demner-Fushman.xlsx')

print(df.head())
df.shape
```

Listing 1: Loading the Dataset

Running the code above we realize that our dataset has 1000 records with 3 columns (File name: which we don't need, CHQ: which is the question and the text we need to summarize, Summary: which is our target variable).

2 Step 2: Preprocessing

In this step I used re library to preprocess my dataset, here is the code snippet:

```python
1  import re
2
3  def preprocess(text):
4          text = re.sub(r'<.*?>', '', text)
5          text = re.sub(r'[^a-zA-Z0-9\s]', '', text)
6          text = re.sub(r'\s+', ' ', text)
7          text = text.strip()
8          text = text.lower()
9
10         return text
11
12 df['CHQ'] = df['CHQ'].apply(preprocess)
13 df['Summary'] = df['Summary'].apply(preprocess)
```

Listing 2: Preprocessing the Dataset

Note that line 4 removes all HTML or XML tags from the text, line 5 removes all characters that are not letters (a-z, A-Z), digits (0-9), or whitespace, line 6 replaces multiple consecutive whitespace characters with a single space, line 7 removes leading and trailing whitespace from the text, and line 8 converts all characters in the text to lowercase.

We apply this function to both CHQ and Summary Column.

3 Step 3: Handling Missing or Irregular Data

We run the code below and do some EDA steps to handle any irregularity in our dataset:

```
print(df["CHQ"].isnull().sum())
print(df["Summary"].isnull().sum())
print(np.array([1 for text in df["CHQ"] if len(text) < 10]).sum
    ())
print(np.array([1 for text in df["Summary"] if len(text) < 10])
    .sum())
```

Listing 3: Checking missing data

when I ran the above code I realized that our dataset doesn't have any missing values.

Then I drew the Box Plot of these two columns and this was the result:

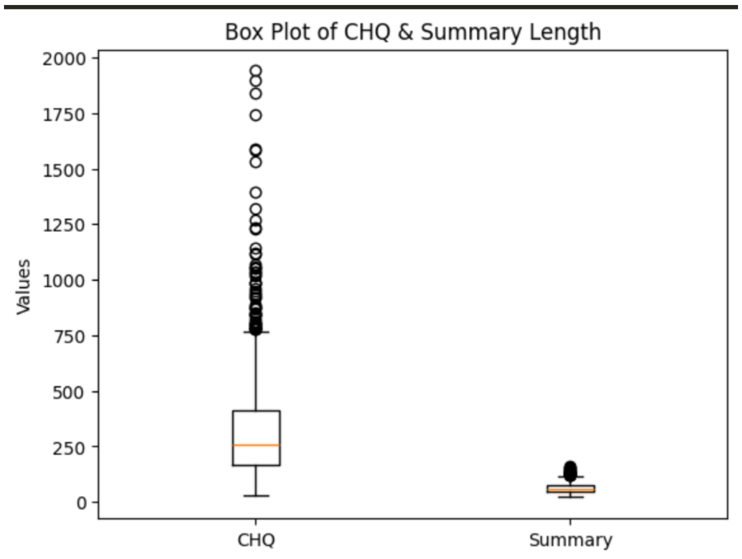


Figure 1: BoxPlot

As you can see above, there are a lot of outliers so we definitely need to trim or pad these texts!

To do that we need to determine what length is good enough, so I used the IQR method which finds the interquartile range of the data, and then finds the middle of the range, and uses the difference between the middle and the upper and lower quartiles to find the length of the outliers, then put aside the outliers calculate the mean length of the remaining data and realized the perfect length is 300 for CHQ texts and 60 for Summary texts, here is the overall formula of IQR:

$IQR = Q_3 - Q_1$ which Q_1 is the first quartile and Q_3 is the third quartile.

$$LowerBound = Q_1 - 1.5 \cdot IQR$$

$$UpperBound = Q_3 + 1.5 \cdot IQR$$

So any data outside of the [LowerBound, UpperBound] is considered an outlier.

The method I used for truncation is to trim the end, and for padding to pad to the right white space!

4 Step 4: Translate Questions to a Pivot Language

In this step I used MarianMTModel for translation and then used google translate API as well.

I used the code snippet below for this task:

```
def translate_texts(texts, lang, mx_len, all_model =
    models_and_tokenizers):
        model = all_model[lang]["model"]
        tokenizer = all_model[lang]["tokenizer"]

        tokenized_texts = tokenizer(texts, return_tensors="pt",
             truncation=True, padding=True, max_length=mx_len).
            to(device)
        translated = model.generate(**tokenized_texts)

        return [tokenizer.decode(t, skip_special_tokens=True)
            for t in translated]

batch_size = 16
for lang in tqdm(languages.keys()):
df[f'CHQ_{lang}'] = (
df['CHQ']
.groupby(df.index // batch_size)
.transform(lambda group: translate_texts(group.tolist(), lang,
    300))
)
```

Listing 4: MarianMTModel

One issue I encountered was that I had some problem with the speed of this process, so i searched a bit and found that for better efficiency, you can use batches since the models support that and it decreases the overhead of the process.

What we do in the above code is essentialy we first to tokenize the texts make trunc and padd them if necessary (which we did also ealier so it won't change much), make sure that is in the format the model needs(torch.tensor) and then we pass it to the model and get the output then decoded it using the tokenizer and we get the final output.

Here is an example of german translation:

Original Text (English)

Who makes bromocriptine? I am wondering what company makes the drug bromocriptine. I need it for a mass I have on my pituitary gland, and the cost just keeps raising. I cannot ever buy a full prescription because of the price, and I was told if I get a hold of the maker of the drug, sometimes they offer c...

Translated Text (German)

Wer macht Bromocriptin? Ich frage mich, welche Firma macht das Medikament Bromocriptin. Ich brauche es für eine Masse, die ich auf meiner Hypophyse habe, und die Kosten nur hält erhöhen. Ich kann nie kaufen ein volles Rezept wegen des Preises, und ich wurde gesagt, wenn ich einen Halt des Herstellers des Medikaments bekommen, manchmal bieten sie c...

For the API, I searched a lot and the original translate google API needed credit card to use it. I found the google trans library which supposedly does the same thing but it's free!

here is the code snippet i used for API translation:

```
import asyncio
import nest_asyncio
from googletrans import Translator
from tqdm import tqdm

nest_asyncio.apply()

async def google_trans_api(text, lang):
        async with Translator() as translator:
                result = await translator.translate(text, src='
                    en', dest=lang)

                return result.text
```

Listing 5: Google Translate API

And for translating them back to english, I just switch the src and dest and it's done!

5 Step 5: Translate Back to English

In this step we just reverse our models so that we have the english translation of each of those pivot language translations!

Here is an example of translating the german translation back to english using the reversed model:

Original Text (English)

Who makes bromocriptine? I am wondering what company makes the drug bromocriptine. I need it for a mass I have on my pituitary gland, and the cost just keeps raising. I cannot ever buy a full prescription because of the price, and I was told if I get a hold of the maker of the drug, sometimes they offer c...

Translated Text (German back to English)

who makes bromocriptin i wonder which company makes the drug bromocriptin i need it for a mass that i have on my pituitary and only keeps increasing the cost i can never buy a full prescription because of the price and i was told if i get a hold of the manufacturer of the drug sometimes offer it c...

6 Step 6: Using FQD to select a new Dataset

We use the following instructions that were provided in the task description:

FQD measures the distributional distance between the semantic representation of the gold question (original question) and the round-trip generated question. We assume that question semantic representations follow the multi-dimensional Gaussian distribution with first two moments: mean and covariance. The distance between these two Gaussian distributions is measured by the Fréchet distance.

Let h_Q be the semantic representation of the gold question and $\hat{h}_Q$ be the semantic representation of the round-trip question.

The Fréchet Question Distance between Q and $\hat{Q}$ is computed as follows:

$$d_{\text{FQD}}(Q, \hat{Q}) = 1 - \frac{h_Q \cdot \hat{h}_Q}{\|h_Q\| \|\hat{h}_Q\|}$$

To produce a uniform FQD score, we linearly scale the $d_{\text{FQD}}(Q, \hat{Q})$ in the range $[0, 1]$ using the following min-max normalization:

$$\text{FQD}(Q, \hat{Q}) = \frac{d_{\text{FQD}}(Q, \hat{Q}) - \min(d_{\text{FQD}})}{\max(d_{\text{FQD}}) - \min(d_{\text{FQD}})},$$

where $\min(d_{\text{FQD}})$ and $\max(d_{\text{FQD}})$ represent the minimum and maximum FQD in the dataset.

When the distribution of the gold question is close to the distribution of the round-trip generated question, the FQD score is close to zero. In order to have diverse, informative, and non-redundant samples in the training set, one does not need to include the round-trip generated questions whose FQD scores with gold questions are either low (near-same question) or high (entirely different questions). Toward this, we aim to select the round-trip generated questions such that the FQD score with gold questions is found to be in an optimal range. Given the round-trip generated questions $D^{\text{rtt}}_{\text{en}\leftrightarrow\text{xx}}$ with pivot language (xx), we select a subset of the questions as follows:

$$D^{\text{rtt+fqd}}_{\text{en}\leftrightarrow\text{xx}} = \{(\hat{Q}_i, S_i) \mid \mu_1 < \text{FQD}(Q_i, \hat{Q}_i) < \mu_2\},$$

where μ_1 and μ_2 are hyper-parameters (i.e., the optimal threshold).

And the following would be the code snippet of the description:

```
def calculate_pair_FQD(Q1, Q2):
        return 1 - np.dot(Q1, Q2) / (np.linalg.norm(Q1) * np.
            linalg.norm(Q2))

def normalize(x, min, max):
        return (x - min) / (max - min)

def embeding(texts, batch_size=16):
        embeddings = []
        for i in range(0, len(texts), batch_size):
                batch_texts = texts[i:i+batch_size]

                inputs = tokenizer(
                        batch_texts,
                        return_tensors="pt",
                        padding='max_length',
                        truncation=True,
                        max_length=512
                )
                inputs = {k: v.to(device) for k, v in inputs.
                    items()}

                with torch.no_grad():
                        outputs = model(**inputs)
                        batch_embeddings = outputs.
                            last_hidden_state.mean(dim=1).cpu().
                            numpy()
                embeddings.append(batch_embeddings)

        return np.concatenate(embeddings, axis=0)

columns = ["CHQ","CHQ_es","CHQ_it","CHQ_fr","CHQ_de","CHQ_zh"]
```

```

embd_ls = []

for text_col in tqdm(columns):
        texts = df[text_col].tolist()
        embeddings = embeding(texts, batch_size=16)
        embd_ls.append(np.array(list(embeddings)))

def fill(idx):
        embd = embd_ls[idx]
        og_embd = embd_ls[0]
        ans = []
        for i in range(len(og_embd)):
                ans.append(calculate_pair_FQD(og_embd[i], embd[
                    i]))

        df[f"FQD_scores_{columns[idx].split('_')[-1]}"] = np.
            array(ans)

for idx in range(1, len(columns)):
        fill(idx)
```

Listing 6: FQD

For embedding the texts I used BART language model which is is a transformer-based language model developed by Facebook AI that combines the strengths of both bidirectional (like BERT) and autoregressive (like GPT) architectures. It is pre-trained using a denoising autoencoder approach, meaning it learns to reconstruct original texts from corrupted inputs, making it highly effective for various NLP tasks such as text generation, summarization, and translation. BART is useful for text embeddings because it captures deep contextual relationships between words, allowing it to generate rich, meaningful representations of text. These embeddings are particularly beneficial for downstream applications like search, clustering, and semantic similarity tasks, where understanding nuanced textual relationships is crucial.

One issue I had was that I had some problem with the speed of this process, so i searched a bit and found that for better efficiency, you can use batches since the models support that and it decreases the overhead of the process.

7 Step 7: Using PRQD to select a new Dataset

I use the following instructions that were provided in the task description:

Similar to the FQD, it measures the distributional distance between semantic representations of the gold and round-trip generated questions; however, it does not require estimating the moments of the probability distributions.

Intuitively, precision measures how much of $\hat{h}_Q$ can be generated by a portion of h_Q. In contrast, recall measures how much of h_Q can be generated by a portion of $\hat{h}_Q$. Hence, the precision and recall should be high for approximately the same question distributions, whereas, if the question distributions are disjoint in nature, the precision and recall will be zero. Therefore, we aim to select the RTT questions whose precision and recall lie between the optimal range to ensure diversity.

To compute PRQD, we follow the algorithm below, which is based on the precision-recall distance (PRD) curve. Toward this, we compute pairs of precision prec(α) and recall rec(α) for an equiangular grid of values of α:

$$\mathrm{prec}(\alpha) = \sum_{v \in V} \min(\alpha h_Q(v), \hat{h}_Q(v))$$

$$\mathrm{rec}(\alpha) = \sum_{v \in V} \min\left(h_Q(v), \frac{\hat{h}_Q(v)}{\alpha}\right)$$

To compute a single-value metric, we calculate the F1-score corresponding to each α and select the maximum F1-score as the PRQD distance $d_{\mathrm{PRQD}}(Q, \hat{Q})$ as follows:

$$d_{\mathrm{PRQD}}(Q, \hat{Q}) = \max_{\alpha \in \Lambda} \left(\frac{2 \cdot \mathrm{prec}(\alpha) \cdot \mathrm{rec}(\alpha)}{\mathrm{prec}(\alpha) + \mathrm{rec}(\alpha)} \right)$$

Here is the code snippet that I implemented for this step:

```
def prec(Q1, Q2, alpha):
        return np.sum(np.minimum(alpha*Q1, Q2))

def rec(Q1, Q2, alpha):
        return np.sum(np.minimum(Q1, Q2/alpha))

def f1(Q1, Q2, alpha):
        precision = prec(Q1, Q2, alpha)
        recall = rec(Q1, Q2, alpha)

        return 2 * precision * recall / (precision + recall)

def PRQD_score(Q1, Q2):
        alphas = np.array([0.00000001, np.pi/4, np.pi/2, 3*np.
            pi/4, np.pi, 5*np.pi/4, 3*np.pi/2, 7*np.pi/4])

        return max([f1(Q1, Q2, alpha) for alpha in alphas])

columns = ["CHQ", "CHQ_es", "CHQ_it", "CHQ_fr", "CHQ_de", "
    CHQ_zh"]

def fill_pr(idx):
        embd = embd_ls[idx]
        og_embd = embd_ls[0]
        ans = []
        for i in range(len(og_embd)):
                ans.append(PRQD_score(og_embd[i], embd[i]))

        df[f"PRQD_scores_{columns[idx].split('_')[-1]}"] = np.
            array(ans)

for idx in range(1, len(columns)):
        fill_pr(idx)
```

Listing 7: PRQD

And then we normalize these scores and pick the best records same as before ($\mu_1 = 0.3$, $\mu_2 = 0.7$) using the code below:

```
columns = ['PRQD_scores_es', 'PRQD_scores_it', 'PRQD_scores_fr'
    , 'PRQD_scores_de', 'PRQD_scores_zh']

df_prqd = df[['CHQ', 'Summary']].copy()

for col in columns:
        suffix = col.split('_')[-1]
        filter_df = df.loc[(df[col] > 0.3) & (df[col] < 0.7), [
            f"CHQ_{suffix}", "Summary"]]
        filter_df.rename(columns={f"CHQ_{suffix}": 'CHQ'},
            inplace=True)
        df_prqd = pd.concat([df_prqd, filter_df], ignore_index=
            True)
```

Listing 8: Selecting best records

8 Step 8: Using QSV to select a new Dataset

I use the following instructions that were provided in the task description:

Studies show that sentences that maximize the semantic volume in a distributed semantic space are the most diverse and have the least redundant sentences. For the given gold question Q and a set of K RTT generated questions $\{\hat{Q}_1, \hat{Q}_2, \ldots, \hat{Q}_K\}$, first, we extract the semantic representation h_Q for the gold question and each RTT question $\{h_{\hat{Q}_1}, h_{\hat{Q}_2}, \ldots, h_{\hat{Q}_K}\}$ and form a data matrix $H \in \mathbb{R}^{(K+1)\times d}$. Later, we perform the linear dimensionality reduction using Principal Component Analysis (PCA) to project the data matrix H to a lower-dimensional space and obtain the transformed data matrix $H' \in \mathbb{R}^{(K+1)\times 2}$. In order to find and compare the most diverse round-trip candidate questions, we exclude the point corresponding to the gold question from H.

To find a convex maximum volume, we find the convex hull using the Quickhull algorithm as follows:

$$\{p_1, p_2, \ldots, p_C\} = \text{ConvexHull}(h'_1, h'_2, \ldots, h'_K)$$

The convex hull is the smallest convex set that includes all points $h'_1, h'_2, \ldots, h'_K$. The points $\{p_1, p_2, \ldots, p_C\}$ are the vertices of the convex hull. It also guarantees to obtain the maximum semantic area with the selected points. Intuitively, it selects the RTT questions which are diverse in nature.

Here is the code snippet for creating the H':

```
from sklearn.decomposition import PCA

pca = PCA(n_components=2)
H_prime = []
for i in range(1000):
        H = [embd_ls[0][i]] + [embd_ls[1][i]] + [embd_ls[2][i]]
             + [embd_ls[3][i]] + [embd_ls[4][i]] + [embd_ls[5][i
            ]]
        tmp = pca.fit_transform(H)
        H_prime.append(tmp)
```

Listing 9: H'

So H' is a list with length of 1000 and each index shape will be (6, 2), 6 being for 5 languages plus the original and 2 being the reduced dimension of each embedding representation of these texts!

And then we use the following code to select the best RTT's for each record and will add them to the dataframe, Please Note that ls is the list containing the selected texts and summ is its corresponding summary:

```
ls = []
summ = []
H_prime = np.array(H_prime)

for i in range(1000):
        rtt = H_prime[i, 1:, :]

        hull = ConvexHull(rtt)
        idx = hull.vertices
        tmp = df.iloc[i, 3+idx].tolist()
        for s in tmp:
                ls.append(s)

        tmp = df.iloc[2, 3+idx].tolist()
        for s in tmp:
                summ.append(s)
```

Listing 10: Selecting best record using Convex HUll

9 Step 9: Using pre-trained models to summarize questions

In this step I used 4 models:

- **BART (Bidirectional and Auto-Regressive Transformer):** BART is a transformer-based model designed for sequence-to-sequence tasks, trained as a denoising autoencoder. It excels in text generation, summarization, and translation by reconstructing original text from noisy input. Its bidirectional encoder captures deep contextual meaning, while the autoregressive decoder generates fluent text. BART is particularly effective for tasks requiring a mix of understanding and generation, such as dialogue systems and paraphrasing.

- **T5 (Text-To-Text Transfer Transformer):** T5 treats every NLP problem as a text-to-text task, meaning both input and output are formatted as natural language text. It is pre-trained on a diverse range of text transformations, including translation, summarization, and question answering. The model's flexibility allows it to generalize across multiple NLP tasks without task-specific architectures. Its ability to fine-tune efficiently makes it widely used in real-world applications requiring high adaptability.

- **ProphetNet:** ProphetNet is a sequence-to-sequence model optimized for text generation tasks, developed by Microsoft. Unlike traditional autoregressive models, it predicts multiple future tokens simultaneously, improving fluency and coherence in long-form text generation. This future n-gram prediction mechanism helps it perform well in summarization, story generation, and dialogue response tasks. It reduces exposure bias and enhances the quality of generated text, making it competitive with models like T5 and BART.

- **PEGASUS (Pre-training with Extracted Gap-sentences for Abstractive Summarization):** PEGASUS is specifically designed for abstractive text summarization, pre-trained using a novel gap-sentence masking strategy. During training, key sentences are removed from documents, and the model learns to predict them, making it highly proficient in understanding and generating summaries. It achieves state-of-the-art results in summarization tasks across multiple datasets, especially for long and complex documents. Its ability to generalize well to low-resource summarization tasks makes it a powerful tool for text compression and information extraction.

Here is the code snippet for evaluating each data frame created in the previous steps:

```
def generate_summary(model, tokenizer, batch_size=8, max_length
    =80):
        summaries = []
        model.eval()

        for i in range(0, len(texts), batch_size):
                batch_texts = texts[i:i+batch_size]
                print(f"proccessing batch {i}.")
                inputs = tokenizer(
                        batch_texts,
                        return_tensors="pt",
                        padding=True,
                        truncation=True,
                        max_length=512
                )
                inputs = {k: v.to(device) for k, v in inputs.
                    items()}

                with torch.no_grad():
                        summary_ids = model.generate(
                                **inputs,
                                max_length=max_length,
                                num_beams=3,
                                early_stopping=True,
                                use_cache=False
                        )

                batch_summaries = tokenizer.batch_decode(
                    summary_ids.cpu(), skip_special_tokens=True)
                summaries.extend(batch_summaries)

                del inputs, summary_ids
                torch.cuda.empty_cache()

        return summaries

models = [
("facebook/bart-large-cnn", "BART", 8),
("t5-small", "T5", 16),
("microsoft/prophetnet-large-uncased", "ProphetNet", 4),
("facebook/blenderbot-3B", "GPT", 2)
]

for model_name, model_label, batch_size in tqdm(models):
        tokenizer = AutoTokenizer.from_pretrained(model_name)
        model = AutoModelForSeq2SeqLM.from_pretrained(
            model_name).to(device)

        df_pred[f"Pred_Summaries_{model_label}"] =
            generate_summary(
                model,
                tokenizer,
                batch_size=batch_size
        )

        del model, tokenizer
        gc.collect()
        torch.cuda.empty_cache()
```

Listing 11: Evaluation

General Idea:

Firstly we loop over our models that we want to use to evaluate, we introduce a proper batch size for each model, after loading their wieghts and overall model and its tokenizer, we use generate_summary function to generate the summaries for each model and save them in their proper dataframe with proper column.

generate_summary function:

This function gets the model, tokenizer, batch_size, max_length to generate. then what it does is that it uses tokenizer to tokenize the text after being batched and then make it our input, then we move the inputs to cuda so we can utilize the gpu, then we use the model to generate the summaries and then we move the summaries to cpu and finally we return the summaries.

One issue I had was the gpu memory and the solution is that we release the memory after we have used each model or input that we won't need later, so we can use the gpu more efficiently. (That's why I added gc library!)

Another issue I encountered was that the GPT model had some issues, so I looked it up, apparently GPT is much larger and my gpu with the current memory it had couldn't handle it, I had two options either fix it and use GPT which would take god knows how much, or I could use a smaller model with good reported performance, so I decided to use the smaller model, which is the pegasus model.

10 Step 10: Using evaluation metrics and comparing the results

We have 3 functions for evaluations:

- **rouge:** This evaluates the ROUGE-1 and ROUGE-2 scores. ROUGE-1 evaluates the overlap of unigrams between the generated summary and the reference summary, while ROUGE-2 evaluates the overlap of bigrams. The formulas are as follows:

$$\text{ROUGE-1} = \frac{\text{Number of overlapping unigrams}}{\text{Number of reference unigrams}}$$

$$\text{ROUGE-2} = \frac{\text{Number of overlapping bigrams}}{\text{Number of reference bigrams}}$$

- **bleu:** This evaluates the BLEU score. It measures how similar the generated text matches the reference text by focusing on n-gram overlap (similar to ROUGE-N, like ROUGE-1 / ROUGE-2). It computes precision for multiple n-gram lengths and returns the average precision.

- **meteor:** This evaluates the METEOR score. METEOR was designed to address some of the limitations of BLEU, particularly its reliance on exact n-gram matching and lack of semantic awareness. Essentially, it uses synonymy, stemming, and other approaches to better capture meaning equivalence between the generated and reference texts.

And here is the code snippet for these functions:

```python
def rouge(ref, gen):
        scorer = rouge_scorer.RougeScorer(['rouge1', 'rouge2'],
             use_stemmer=True)
        scores = scorer.score(ref, gen)
        return scores["rouge1"].fmeasure, scores["rouge2"].
            fmeasure

def bleu(ref, gen):
        smoothie = SmoothingFunction().method1
        return sentence_bleu([ref.split()], gen.split(),
            smoothing_function=smoothie)

def meteor(ref, gen):
        ref_tokens = nltk.word_tokenize(ref.lower())
        gen_tokens = nltk.word_tokenize(gen.lower())
        return meteor_score([ref_tokens], gen_tokens)
```

Listing 12: Metrics

With the following code snippet we evaluate all of our models with all data frames with all metrics:

```python
evaluation_results_og = {}
evaluation_results_fqd = {}
evaluation_results_pqrd = {}

models = ["BART", "T5", "ProphetNet", "pegasus"]

for model_name in models:
        print(f"evaluating {model_name}:")

        rouge1_scores, rouge2_scores = [], []
        bleu_scores, meteor_scores = [], []

        for ref, gen in zip(df_og["Actual Summary"], df_og[f"
            Pred_Summaries_{model_name}"]):
                if pd.isna(gen) or len(gen) < 5:
                        continue

                r1, r2 = rouge(ref, gen)
                rouge1_scores.append(r1)
                rouge2_scores.append(r2)

                bleu_scores.append(bleu(ref, gen))
                meteor_scores.append(meteor(ref, gen))

        evaluation_results_og[model_name] = {
                "ROUGE-1": np.mean(rouge1_scores),
                "ROUGE-2": np.mean(rouge2_scores),
                "BLEU": np.mean(bleu_scores),
                "METEOR": np.mean(meteor_scores),
        }

        rouge1_scores, rouge2_scores = [], []
        bleu_scores, meteor_scores = [], []

        for ref, gen in zip(df_fqd["Actual Summary"], df_fqd[f"
            Pred_Summaries_{model_name}"]):
                if pd.isna(gen) or len(gen) < 5:
                        continue

                r1, r2 = rouge(ref, gen)
                rouge1_scores.append(r1)
                rouge2_scores.append(r2)

                bleu_scores.append(bleu(ref, gen))
                meteor_scores.append(meteor(ref, gen))

        evaluation_results_fqd[model_name] = {
                "ROUGE-1": np.mean(rouge1_scores),
                "ROUGE-2": np.mean(rouge2_scores),
                "BLEU": np.mean(bleu_scores),
                "METEOR": np.mean(meteor_scores),
        }

        rouge1_scores, rouge2_scores = [], []
        bleu_scores, meteor_scores = [], []

        for ref, gen in zip(df_pqrd["Actual Summary"], df_pqrd[f"Pred_Summaries_{model_name}"]):
                if pd.isna(gen) or len(gen) < 5:
                        continue

                r1, r2 = rouge(ref, gen)
                rouge1_scores.append(r1)
                rouge2_scores.append(r2)

                bleu_scores.append(bleu(ref, gen))
                meteor_scores.append(meteor(ref, gen))

        evaluation_results_pqrd[model_name] = {
                "ROUGE-1": np.mean(rouge1_scores),
                "ROUGE-2": np.mean(rouge2_scores),
                "BLEU": np.mean(bleu_scores),
                "METEOR": np.mean(meteor_scores),
        }
```

Listing 13: Final Evaluation

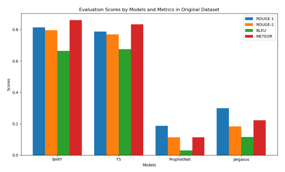
And finally I used the following code for visualization of their plots:

```
import matplotlib.pyplot as plt


metrics = ["ROUGE-1", "ROUGE-2", "BLEU", "METEOR"]

results = {metric: [] for metric in metrics}

for model in models:
        for metric in metrics:
                results[metric].append(evaluation_results_qsv[model][metric])

x = np.arange(len(models))
width = 0.2

fig, ax = plt.subplots(figsize=(10, 6))

```

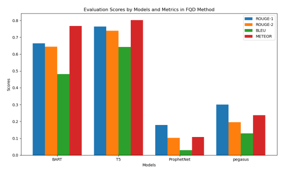
```
for i, metric in enumerate(metrics):
        ax.bar(x + i * width - width * (len(metrics) - 1) / 2, results[metric], width, label=metric)

ax.set_xlabel('Models')
ax.set_ylabel('Scores')
ax.set_title('Evaluation Scores by Models and Metrics in QSV Method')
ax.set_xticks(x)
ax.set_xticklabels(models)
ax.legend()

plt.tight_layout()
plt.show()
```

Listing 14: visualize

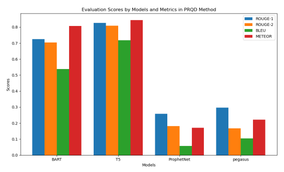
And here is the Plots:



(a) Original Dataset.

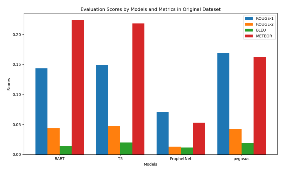


(b) FQD method.

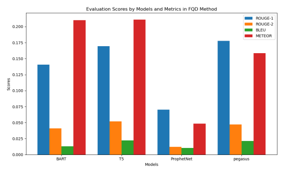


(c) PRQD method.

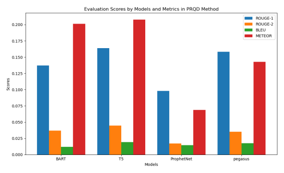
Figure 2: Results on CHQ vs the generated Summary.



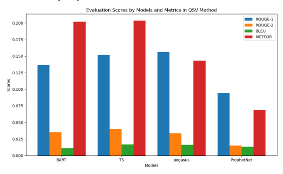
(a) Original Dataset.



(b) FQD method.



(c) PRQD method.



(d) QSV method.

Figure 3: Results on Actual Summary vs the generated Summary.

As you can see above, The overall result on CHQ are pretty good, so the models have been able to summarize the questions pretty well, but the human summarization is much more complex than the AI generated ones hence low accuracy.

But the overall order of methods and models is:

For models I would say: T5, BART, Pegasus and then ProphetNet in order.

For methods I would say: FQD, PRQD, QSV in that order.

Weirdly the original dataset performed if not better as well as the other methods!

Please note that some of these models are not meant to be used for summarization without any fine-tuning, so the results may not be a good representation of the performance of the model in general!