Creation of dataset based on specific scenarios that can be happening in SurveySparrow

```
import csv
import random

# More specific categories
categories = [
    "Survey Creation",
    "Data Collection",
    "Results Analysis",
    "Account Management",
    "Pricing and Billing",
    "Technical Issues",
    "Integrations",
    "Survey Distribution"
]

# More realistic queries with varying complexity
queries = {
    "Survey Creation": [
        "How do I add branching logic to my survey?",
        "Can I use custom CSS in my survey design?",
        "Is there a way to randomize question order?",
        "How many question types does SurveySparrow offer?",
        "Can I create a multi-language survey?",
    ],
    "Data Collection": [
        "What's the maximum number of responses I can collect?",
        "How can I prevent duplicate responses?",
        "Is it possible to collect responses offline?",
        "Can I set an expiry date for my survey?",
        "How do I enable partial response saving?",
    ],
    "Results Analysis": [
        "How can I create custom reports for specific question
types?",
        "Is there a way to filter responses based on specific
criteria?",
        "Can I generate word clouds from open-ended responses?",
        "How do I export my survey data to SPSS format?",
        "Is it possible to set up automated report generation?",
    ],
    "Account Management": [
        "How do I add team members to my account?",
        "Can I transfer ownership of a survey to another user?",
        "What's the process for upgrading from a free to a paid
plan?",
        "How can I enable two-factor authentication for my account?",
        "I need to close my account, what steps should I take?",
```

```
    ],
    "Pricing and Billing": [
        "Can you explain the difference between your pricing tiers?",
        "Is there a discount for annual billing?",
        "How do I update my credit card information?",
        "Do you offer any special pricing for non-profit
organizations?",
        "I was charged twice this month, can you help me understand
why?",
    ],
    "Technical Issues": [
        "The survey embed code isn't working on my website",
        "I'm getting a 404 error when trying to access my results",
        "The email invitations aren't being delivered to some
respondents",
        "My custom domain isn't resolving correctly for my surveys",
        "The survey is loading very slowly for respondents, how can I
optimize it?",
    ],
    "Integrations": [
        "How do I set up the Zapier integration?",
        "Can SurveySparrow integrate directly with our CRM system?",
        "Is there an API available for custom integrations?",
        "How do I connect my Google Analytics account to track survey
performance?",
        "Can I use webhooks to send survey data to our internal
systems?",
    ],
    "Survey Distribution": [
        "What's the best way to share my survey on social media?",
        "How can I embed the survey in an email newsletter?",
        "Is there a QR code option for sharing surveys?",
        "Can I schedule automated reminder emails for incomplete
responses?",
        "How do I create a custom URL for my survey?",
    ]
}

# Refined escalation and sentiment options
escalation_options = ["Escalation needed", "No escalation needed"]
sentiment_options = ["Positive", "Negative", "Neutral"]

# Generate variations with more context
def generate_variations(query, category, n=3):
    variations = [query]
    prefixes = [
        f"I'm having trouble with {category.lower()}: ",
        f"Can you help me understand how to ",
        f"I'm confused about {category.lower()}: ",
```

```
        f"I need assistance with {category.lower()}: ",
        f"Could you explain how to "
    ]
    for _ in range(n-1):
        variations.append(random.choice(prefixes) + query.lower())
    return variations

# Create the dataset
dataset = []
for category, category_queries in queries.items():
    for query in category_queries:
        variations = generate_variations(query, category)
        for variation in variations:
            # Assign escalation need based on query complexity
            escalation = "Escalation needed" if len(variation.split())
> 10 or "error" in variation.lower() or "isn't working" in
variation.lower() else "No escalation needed"

            # Assign sentiment based on query content
            if any(word in variation.lower() for word in ["error",
"trouble", "confused", "isn't working"]):
                sentiment = "Negative"
            elif any(word in variation.lower() for word in ["help",
"explain", "understand"]):
                sentiment = "Neutral"
            else:
                sentiment = "Positive"

            dataset.append([variation, escalation, sentiment,
category])

# Shuffle the dataset
random.shuffle(dataset)

# Write to CSV
with open('Agent_escalation.csv', 'w', newline='', encoding='utf-8')
as file:
    writer = csv.writer(file)
    writer.writerow(["Query", "Escalation", "Sentiment", "Category"])
    writer.writerows(dataset)

print(f"Dataset created with {len(dataset)} entries.")
```

```
Dataset created with 120 entries.
```

Using BERT model for Agent escalation

```
#Import libraries

import pandas as pd
```

```
import torch
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report
from transformers import BertTokenizer, BertForSequenceClassification,
AdamW
from torch.utils.data import TensorDataset, DataLoader, RandomSampler,
SequentialSampler

#Load and prepare the data

df = pd.read_csv('/content/Agent_escalation.csv')


sentences = df.Query.values
labels = (df.Escalation == "Escalation needed").astype(int).values

#Load the BERT tokenizer

tokenizer = BertTokenizer.from_pretrained('bert-base-uncased',
do_lower_case=True)
```

```
/usr/local/lib/python3.10/dist-packages/huggingface_hub/utils/
_token.py:89: UserWarning:
The secret `HF_TOKEN` does not exist in your Colab secrets.
To authenticate with the Hugging Face Hub, create a token in your
settings tab (https://huggingface.co/settings/tokens), set it as
secret in your Google Colab and restart your session.
You will be able to reuse this secret in all of your notebooks.
Please note that authentication is recommended but still optional to
access public models or datasets.
  warnings.warn(
```


```
{"model_id":"a554c5e24ca54c769ce306fc3a4a1423","version_major":2,"vers
ion_minor":0}

{"model_id":"21215b0c9a3c4a50b4e80e8bf9206790","version_major":2,"vers
ion_minor":0}

{"model_id":"d3823f1c7c85401abb8828d0a2d919e5","version_major":2,"vers
ion_minor":0}
```


```
/usr/local/lib/python3.10/dist-packages/huggingface_hub/
file_download.py:1132: FutureWarning: `resume_download` is deprecated
and will be removed in version 1.0.0. Downloads always resume when
possible. If you want to force a new download, use
`force_download=True`.
  warnings.warn(
```


```
{"model_id":"b5f51fe2687a4300805d2a450b50a040","version_major":2,"vers
ion_minor":0}
```

```
# Tokenize all of the sentences and map the tokens to their word IDs
input_ids = []
attention_masks = []

for sent in sentences:
    encoded_dict = tokenizer.encode_plus(
                        sent,
                        add_special_tokens = True,
                        max_length = 64,
                        pad_to_max_length = True,
                        return_attention_mask = True,
                        return_tensors = 'pt',
                   )
    input_ids.append(encoded_dict['input_ids'])
    attention_masks.append(encoded_dict['attention_mask'])

# Convert to tensors
input_ids = torch.cat(input_ids, dim=0)
attention_masks = torch.cat(attention_masks, dim=0)
labels = torch.tensor(labels)
```

```
Truncation was not explicitly activated but `max_length` is provided a
specific value, please use `truncation=True` to explicitly truncate
examples to max length. Defaulting to 'longest_first' truncation
strategy. If you encode pairs of sequences (GLUE-style) with the
tokenizer you can select this strategy more precisely by providing a
specific strategy to `truncation`.
/usr/local/lib/python3.10/dist-packages/transformers/tokenization_util
s_base.py:2699: FutureWarning: The `pad_to_max_length` argument is
deprecated and will be removed in a future version, use `padding=True`
or `padding='longest'` to pad to the longest sequence in the batch, or
use `padding='max_length'` to pad to a max length. In this case, you
can give a specific length with `max_length` (e.g. `max_length=45`) or
leave max_length to None to pad to the maximal input size of the model
(e.g. 512 for Bert).
  warnings.warn(
```

```
# Split into training and testing sets
x_train, x_test, y_train, y_test = train_test_split(input_ids, labels,
random_state=42, test_size=0.1)
train_masks, validation_masks, _, _ =
train_test_split(attention_masks, labels, random_state=42,
test_size=0.1)

#Batch Size
batch_size = 32

#Create dataloaders
train_data = TensorDataset(x_train, train_masks, y_train)
train_sampler = RandomSampler(train_data)
```

```
train_dataloader = DataLoader(train_data, sampler=train_sampler,
batch_size=batch_size)

validation_data = TensorDataset(x_test, validation_masks,y_test)
validation_sampler = SequentialSampler(validation_data)
validation_dataloader = DataLoader(validation_data,
sampler=validation_sampler, batch_size=batch_size)

# Load BertForSequenceClassification
model = BertForSequenceClassification.from_pretrained(
    "bert-base-uncased",
    num_labels = 2,
    output_attentions = False,
    output_hidden_states = False,
)

# Set up the optimizer
optimizer = AdamW(model.parameters(), lr = 2e-5,eps = 1e-8)
```

{"model_id":"cabfbe9572544d70b966bb66575238da","version_major":2,"vers
ion_minor":0}

```
Some weights of BertForSequenceClassification were not initialized
from the model checkpoint at bert-base-uncased and are newly
initialized: ['classifier.bias', 'classifier.weight']
You should probably TRAIN this model on a down-stream task to be able
to use it for predictions and inference.
/usr/local/lib/python3.10/dist-packages/transformers/optimization.py:5
88: FutureWarning: This implementation of AdamW is deprecated and will
be removed in a future version. Use the PyTorch implementation
torch.optim.AdamW instead, or set `no_deprecation_warning=True` to
disable this warning
  warnings.warn(
```

```
# Training loop
device = torch.device('cpu')
model.to(device)
epochs = 4

for epoch in range(epochs):
    model.train()
    for batch in train_dataloader:
        b_input_ids = batch[0].to(device)
        b_input_mask = batch[1].to(device)
        b_labels = batch[2].to(device)
        model.zero_grad()
        outputs = model(b_input_ids,
                        token_type_ids=None,
                        attention_mask=b_input_mask,
                        labels=b_labels)
```

```
        loss = outputs[0]
        loss.backward()
        optimizer.step()

# Validation
model.eval()
eval_loss, eval_accuracy = 0, 0
for batch in validation_dataloader:
        batch = tuple(t.to(device) for t in batch)
        b_input_ids, b_input_mask, b_labels = batch
        with torch.no_grad():
            outputs = model(b_input_ids,
                            token_type_ids=None,
                            attention_mask=b_input_mask)
        logits = outputs[0]
        logits = logits.detach().cpu().numpy()
        label_ids = b_labels.to('cpu').numpy()
        eval_accuracy += (logits.argmax(axis=1) == label_ids).mean()
print(f"Epoch {epoch+1}, Validation Accuracy:
{eval_accuracy/len(validation_dataloader)}")
```

```
Epoch 4, Validation Accuracy: 0.9166666666666666
```

```
# Test the model
model.eval()
predictions = []
true_labels = []
for batch in validation_dataloader:
    batch = tuple(t.to(device) for t in batch)
    b_input_ids, b_input_mask, b_labels = batch
    with torch.no_grad():
        outputs = model(b_input_ids, token_type_ids=None,
attention_mask=b_input_mask)
    logits = outputs[0]
    logits = logits.detach().cpu().numpy()
    label_ids = b_labels.to('cpu').numpy()
    predictions.extend(logits.argmax(axis=1))
    true_labels.extend(label_ids)

print(classification_report(true_labels, predictions,
target_names=['No escalation needed', 'Escalation needed']))
```

```
                     precision    recall  f1-score   support

No escalation needed       0.67      1.00      0.80         2
   Escalation needed       1.00      0.90      0.95        10

            accuracy                           0.92        12
           macro avg       0.83      0.95      0.87        12
        weighted avg       0.94      0.92      0.92        12
```

Saving the developed model

```
import os
from google.colab import drive
import torch

# Mount Google Drive
drive.mount('/content/drive')

# Define the path where you want to save the model in your Google
Drive
save_path = '/content/drive/My Drive/BERT_SurveySparrow_Model'

# Create the directory if it doesn't exist
os.makedirs(save_path, exist_ok=True)

# Save the model
model.save_pretrained(save_path)

# Save the tokenizer
tokenizer.save_pretrained(save_path)

print(f"Model and tokenizer saved to Google Drive at: {save_path}")

# Verify that the files are saved
!ls "{save_path}"
```

```
Drive already mounted at /content/drive; to attempt to forcibly
remount, call drive.mount("/content/drive", force_remount=True).
Model and tokenizer saved to Google Drive at: /content/drive/My
Drive/BERT_SurveySparrow_Model
config.json         README.txt             tokenizer_config.json
model.safetensors  special_tokens_map.json  vocab.txt
```

Testing out the Agent escalation model by providing user inputs.

```
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

```
import os
import torch
from transformers import BertTokenizer, BertForSequenceClassification,
BertConfig

# Define the path where the model is saved
model_path = '/content/drive/My Drive/BERT_SurveySparrow_Model'

# Load the config
config = BertConfig.from_pretrained(model_path)

# Load the model
model = BertForSequenceClassification.from_pretrained(model_path,
config=config)

# Load the tokenizer
tokenizer = BertTokenizer.from_pretrained(model_path)

# Move the model to the appropriate device
device = torch.device('cpu')
model.to(device)

print("Model loaded successfully")
```

```
Model loaded successfully
```

```
# Test the model
def predict_escalation(query):
    inputs = tokenizer(query, return_tensors="pt", truncation=True,
padding=True, max_length=64)
    inputs = {k: v.to(device) for k, v in inputs.items()}

    with torch.no_grad():
        outputs = model(**inputs)

    prediction = torch.argmax(outputs.logits, dim=1).item()
    return "Escalation needed" if prediction == 1 else "No escalation
needed"

# Example usage
query = "How to make this work"
result = predict_escalation(query)
print(f"Query: {query}")
print(f"Prediction: {result}")
```

```
Query: How to make this work
Prediction: No escalation needed
```

Using the Agent escalation model created using BERT sentimental analysis is done in the dataset to analyse the model efficiency.

```
import nltk
nltk.download('punkt')
nltk.download('stopwords')

import pandas as pd
import torch
from transformers import BertTokenizer, BertForSequenceClassification
from nltk.tokenize import word_tokenize
from nltk.corpus import stopwords
from nltk.probability import FreqDist
```

```
[nltk_data] Downloading package punkt to /root/nltk_data...
[nltk_data]   Unzipping tokenizers/punkt.zip.
[nltk_data] Downloading package stopwords to /root/nltk_data...
[nltk_data]   Unzipping corpora/stopwords.zip.
```

```
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

```
# Load the saved model and tokenizer
import os
import torch
from transformers import BertTokenizer, BertForSequenceClassification,
BertConfig
model_path = '/content/drive/My Drive/BERT_SurveySparrow_Model'
config = BertConfig.from_pretrained(model_path)
model = BertForSequenceClassification.from_pretrained(model_path,
config=config)
tokenizer = BertTokenizer.from_pretrained(model_path)
device = torch.device('cpu')
model.to(device)

def predict_sentiment(query):
    inputs = tokenizer(query, return_tensors="pt", truncation=True,
padding=True, max_length=64)
    inputs = {k: v.to(device) for k, v in inputs.items()}

    with torch.no_grad():
        outputs = model(**inputs)

    logits = outputs.logits
    probabilities = torch.softmax(logits, dim=1)
    sentiment_score = probabilities[0][1].item()  # Probability of
positive sentiment

    if sentiment_score > 0.6:
        return "Positive"
    elif sentiment_score < 0.4:
        return "Negative"
```

```
    else:
        return "Neutral"

def extract_keywords(query, num_keywords=5):
    # Tokenize the query
    tokens = word_tokenize(query.lower())

    # Remove stopwords
    stop_words = set(stopwords.words('english'))
    tokens = [token for token in tokens if token not in stop_words and
token.isalnum()]

    # Get frequency distribution
    fdist = FreqDist(tokens)

    # Return the most common words
    return [word for word, _ in fdist.most_common(num_keywords)]

# Function to analyze a query
def analyze_query(query):
    sentiment = predict_sentiment(query)
    keywords = extract_keywords(query)
    return {
        "query": query,
        "sentiment": sentiment,
        "keywords": keywords
    }

# Load the dataset
df = pd.read_csv('/content/Agent_escalation.csv')

# Analyze each query in the dataset
results = []
for query in df['Query']:
    results.append(analyze_query(query))

# Create a new dataframe with the results
results_df = pd.DataFrame(results)

# Merge with the original dataset
final_df = pd.concat([df, results_df[['sentiment', 'keywords']]],
axis=1)

# Display the first few rows of the final dataframe
print(final_df.head())

# Save the results
final_df.to_csv('sentimental_analyzed_dataset.csv', index=False)
print("Analysis complete. Results saved to
'sentimental_analyzed_dataset.csv'")
```

```
                                                Query
Escalation  \
0            Can I set an expiry date for my survey?  No escalation
needed
1  Could you explain how to how do i connect my g...     Escalation
needed
2  I need assistance with account management: wha...     Escalation
needed
3  I'm having trouble with pricing and billing: i...     Escalation
needed
4  Can you help me understand how to the survey i...     Escalation
needed

  Sentiment             Category sentiment  \
0  Positive      Data Collection  Negative
1   Neutral         Integrations  Positive
2  Positive   Account Management  Positive
3  Negative  Pricing and Billing  Positive
4   Neutral     Technical Issues  Positive

                                           keywords
0                       [set, expiry, date, survey]
1      [could, explain, connect, google, analytics]
2  [need, assistance, account, management, process]
3     [billing, trouble, pricing, discount, annual]
4       [help, understand, survey, loading, slowly]
Analysis complete. Results saved to 'sentimental_analyzed_dataset.csv'
```

Developing a small IVR system using our developed Agent escalating and Empathetic response model to test it.

```
from google.colab import drive
drive.mount('/content/drive')
```

```
Drive already mounted at /content/drive; to attempt to forcibly
remount, call drive.mount("/content/drive", force_remount=True).
```

```
import torch
from transformers import BertTokenizer, BertForSequenceClassification
import nltk
from nltk.tokenize import word_tokenize
from nltk.corpus import stopwords
from nltk.probability import FreqDist

nltk.download('punkt')
nltk.download('stopwords')

# Load the model (adjust the path as necessary)
model_path = '/content/drive/My Drive/BERT_SurveySparrow_Model'
model = BertForSequenceClassification.from_pretrained(model_path)
tokenizer = BertTokenizer.from_pretrained(model_path)
device = torch.device('cpu')
model.to(device)
```

```
[nltk_data] Downloading package punkt to /root/nltk_data...
[nltk_data]   Package punkt is already up-to-date!
[nltk_data] Downloading package stopwords to /root/nltk_data...
[nltk_data]   Package stopwords is already up-to-date!
```

```
BertForSequenceClassification(
  (bert): BertModel(
    (embeddings): BertEmbeddings(
      (word_embeddings): Embedding(30522, 768, padding_idx=0)
      (position_embeddings): Embedding(512, 768)
      (token_type_embeddings): Embedding(2, 768)
      (LayerNorm): LayerNorm((768,), eps=1e-12,
elementwise_affine=True)
      (dropout): Dropout(p=0.1, inplace=False)
    )
    (encoder): BertEncoder(
      (layer): ModuleList(
        (0-11): 12 x BertLayer(
          (attention): BertAttention(
            (self): BertSdpaSelfAttention(
              (query): Linear(in_features=768, out_features=768,
bias=True)
              (key): Linear(in_features=768, out_features=768,
bias=True)
              (value): Linear(in_features=768, out_features=768,
bias=True)
              (dropout): Dropout(p=0.1, inplace=False)
```

```
            )
            (output): BertSelfOutput(
              (dense): Linear(in_features=768, out_features=768,
bias=True)
              (LayerNorm): LayerNorm((768,), eps=1e-12,
elementwise_affine=True)
              (dropout): Dropout(p=0.1, inplace=False)
            )
          )
          (intermediate): BertIntermediate(
            (dense): Linear(in_features=768, out_features=3072,
bias=True)
            (intermediate_act_fn): GELUActivation()
          )
          (output): BertOutput(
            (dense): Linear(in_features=3072, out_features=768,
bias=True)
            (LayerNorm): LayerNorm((768,), eps=1e-12,
elementwise_affine=True)
            (dropout): Dropout(p=0.1, inplace=False)
          )
        )
      )
    )
    (pooler): BertPooler(
      (dense): Linear(in_features=768, out_features=768, bias=True)
      (activation): Tanh()
    )
  )
  (dropout): Dropout(p=0.1, inplace=False)
  (classifier): Linear(in_features=768, out_features=2, bias=True)
)

def predict_sentiment(query):
    inputs = tokenizer(query, return_tensors="pt", truncation=True,
padding=True, max_length=64)
    inputs = {k: v.to(device) for k, v in inputs.items()}

    with torch.no_grad():
        outputs = model(**inputs)

    logits = outputs.logits
    probabilities = torch.softmax(logits, dim=1)
    sentiment_score = probabilities[0][1].item()  # Probability of
positive sentiment

    if sentiment_score > 0.6:
        return "Positive"
    elif sentiment_score < 0.4:
        return "Negative"
```

```
    else:
        return "Neutral"

def extract_keywords(query, num_keywords=5):
    tokens = word_tokenize(query.lower())
    stop_words = set(stopwords.words('english'))
    tokens = [token for token in tokens if token not in stop_words and
token.isalnum()]
    fdist = FreqDist(tokens)
    return [word for word, _ in fdist.most_common(num_keywords)]

def analyze_query(query):
    sentiment = predict_sentiment(query)
    keywords = extract_keywords(query)
    return {
        "query": query,
        "sentiment": sentiment,
        "keywords": keywords
    }

def generate_response(analysis_result):
    sentiment = analysis_result['sentiment']
    keywords = analysis_result['keywords']

    if sentiment == "Negative":
        return "I'm sorry to hear that you're having difficulties. Let
me connect you with a specialist who can help you right away.", True
    elif "password" in keywords:
        return "To reset your password, please go to the login page
and click on 'Forgot Password'.", False
    elif "survey" in keywords and "create" in keywords:
        return "Creating a new survey is easy! Just log in to your
account and click on 'Create New Survey' on the dashboard.", False
    else:
        return "Thank you for your query. How else can I assist you
today?", False

def main():
    print("Welcome to Simple Workspace!")
    print("Type your query or type 'exit' to quit.")

    while True:
        query = input("Your query: ")

        if query.lower() == 'exit':
            print("Exiting...")
            break

        analysis_result = analyze_query(query)
        response, should_escalate = generate_response(analysis_result)
```

```
        print("Response:", response)
        if should_escalate:
            print("Escalating call...")
        else:
            print("Not escalating call.")
        print()

if __name__ == "__main__":
    main()
```

```
Welcome to Simple Workspace!
Type your query or type 'exit' to quit.
Your query: I forgot my password. is this bad?
Response: To reset your password, please go to the login page and
click on 'Forgot Password'.
Not escalating call.

Your query: exit
Exiting...
```

IVR using Speech to text

```
import speech_recognition as sr

def audio_to_text(file_path):
    recognizer = sr.Recognizer()

    with sr.AudioFile(file_path) as source:
        audio_data = recognizer.record(source)
        text = recognizer.recognize_google(audio_data)

from pydub import AudioSegment

def process_audio_file(file_path):
    text = audio_to_text(file_path)
    sentiment = predict_sentiment(text)

    if sentiment == "Negative":
        return "I'm sorry to hear that you're having difficulties. Let
me connect you with a specialist who can help you right away.", True
    else:
        return "Thank you for your query. How else can I assist you
today?", False

def main():
    audio_file_path = '/content/record_out.wav'
    recognized_text = audio_to_text(audio_file_path)
    print("Recognized Text:", recognized_text)
```

```
    # Further processing with sentiment analysis or other logic
    sentiment = predict_sentiment(recognized_text)

    if sentiment == "Negative":
        print("Sentiment: Negative")
        print("I'm sorry to hear that you're having difficulties. Let
me connect you with a specialist who can help you right away.")
        print("Escalating call...")
    else:
        print("Sentiment: Positive or Neutral")
        print("Thank you for your query. How else can I assist you
today?")
        print("Not escalating call.")

if __name__ == "__main__":
    main()
```

```
Converted text: the service is too bad
Recognized Text: the service is too bad
Sentiment: Positive or Neutral
Thank you for your query. How else can I assist you today?
Not escalating call.
```